Automatic Replication of WSRF-based Grid Services via Operation Providers ⋆

G. Moltó *, V. Hernández, J.M. Alonso

Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia. Camino de Vera S/N, 46022 Valencia, Spain

Abstract

The advent of service-oriented architectures in Grid environments has fostered the development of applications in distributed deployments. The Globus Toolkit 4 (GT4) and its implementation of stateful Web services, via the WS-Resource Framework (WSRF), is a suitable platform to develop these Grid services. This way, its increased usage in many scientific areas reveals new scenarios where fault-tolerance and high availability should be considered. This paper describes a library that manages the automatic replication of WSRF-based Grid services. This functionality can be plugged to existing Grid services, by means of minimal changes in its source code, to achieve state replication through WS-Resources. The architecture of the library and its performance evaluation are described. In particular, two different replica topologies are addressed: ring-based and leaf-to-root complete binary tree, in order to achieve resource state update in logarithmic time with respect to the number of replicas. Finally, the paper describes the integration of the replication library into a service-oriented metascheduler to enhance fault-tolerance and to guarantee service availability.

Key words: Grid computing, Replication, Globus Toolkit, WSRF, Service-Oriented Architectures

⋆ The authors wish to thank the financial support received from the Spanish Ministry of Science and Technology to develop the project ngGrid: New Generation Components for the Efficient Exploitation of eScience Infrastructures (TIN2006-12890). This work has been partially supported by the Structural Funds of the European Regional Development Fund (ERDF).

* Corresponding author: Tel. +34963877356. Fax.: +3496877359
Email address: gmolto@dsic.upv.es (G. Moltó).

1 Introduction

During the last years, Grid Computing has proved to be a suitable technology to support the execution of computationally intensive applications in distributed deployments. The Globus Toolkit (GT) version 2 [1] was considered to be the de facto standard in Grid middlewares. It provided a set of services and protocols to ease sharing both computational power and storage capacity within the context of virtual organizations. With the release of GT version 4 (GT4) [2], the move towards service-oriented architectures enlightened a new computing paradigm. Machines in a Grid were no longer dedicated exclusively to computing but they could also implement and offer a whole new catalog of services through standard web-based interfaces. These functionality required stateful Web services and the Grid service term was thus coined.

Since then, the development of Grid services has been growing in order to enrich the functionality of Grid deployments. As Grid services become a key factor in distributed infrastructures, topics such as fault-tolerance and high availability turn into requirements in order to maintain service availability regardless of software and hardware failures. This way, replication allows to improve reliability, fault-tolerance and accessibility by providing the user with different replicas of the same Grid service, all of them with a coherent state.

The fault-tolerance topic in Grid computing has been extensively addressed in the literature, where it typically focuses on reliable job management and execution on distributed deployments [3,4]. Achieving high availability through replication has also been studied, but mainly related to data Grids [5,6]. However, the area of building reliable replicated Grid services is still very new. In [7], the problem of fault-tolerant management of a set of Grid services is addressed. For that, a hierarchical system comprised of statically configured bootstrap services is proposed. These are responsible of reinstantiating the failed components. There also exist service managers attached to the replicated services responsible for managing their state. It is a generic approach for the scalable management of Grid services, but loosely related to WSRF-based Grid services.

In [8], the problem of building highly available Grid services is addressed by using primary-backup replication implemented in GT3. With this approach, a single replica processes all the requests and keeps the backup services consistent in case the primary fails. Therefore, the main bottleneck resides at the primary replica. In [9], fault tolerant concepts for stateful Web services are applied to a specific Grid middleware designed for monitoring and migrating high-performance applications. Implemented in GT4, the authors developed a ring replication protocol to provide total message ordering and group membership.

This paper describes a library for the replication of GT4-based Grid services through WS-Resource state management. This functionality can be plugged to already existing Grid services by using the operation providers functionality of GT4. The library manages the update of the replicas to guarantee their consistent state. In addition, different topologies are explored to achieve efficient Grid service reliability and sustained service availability.

The remainder of the paper is structured as follows. First, section 2 briefly describes the underlying technologies employed. Next, section 3 introduces the main goals of the replication library detailing its architecture and the proposed topologies. Later, section 4 details the inclusion of replication in a service-oriented metascheduler and evaluates the performance of the library under different conditions. Finally, section 5 summarizes the paper and points to future work.

2 Grid Services, WSRF and GT4

GT4 provides the implementation of a set of Grid services which conform to OGSA (Open Grid Services Architecture) [10]. OGSA represents an evolution towards a Grid system architecture based on the concepts and technologies provided by Web services. Being developed by the Open Grid Forum (OGF), OGSA defines a common, standard and open architecture for all the services that can be found in a Grid system (job management, resource management, security, etc.) [11]. This way, different vendor tools can cooperate together by means of standard interfaces. Therefore, OGSA defines its underlying architecture to be based on special Web services, which maintain their state from one invocation to another.

This is where WSRF (Web Services Resource Framework) [12] comes into play, specifying how Web services can be stateful. This is achieved by coupling a data container, which stores the stateful data, to a Web service, thus obtaining a WS-Resource. OGSA uses this new concept to specify the underlying architecture of the Grid services. Therefore, GT4 includes an implementation of WSRF as well as a set of Grid services developed on top of WSRF, which are compliant with OGSA requirements.

Figure 1 describes the architecture of a typical WSRF-based Grid service implemented under GT4, as described in [11]. Notice that the Grid service is composed by a Factory service, which creates the WS-Resources through the Resource Home, and the Instance service, which actually implements the operations of the service and modifies the state of the WS-Resource. Grid services expose their operations via the Web Services Definition Language (WSDL). They are typically implemented using the Java language and deployed in the

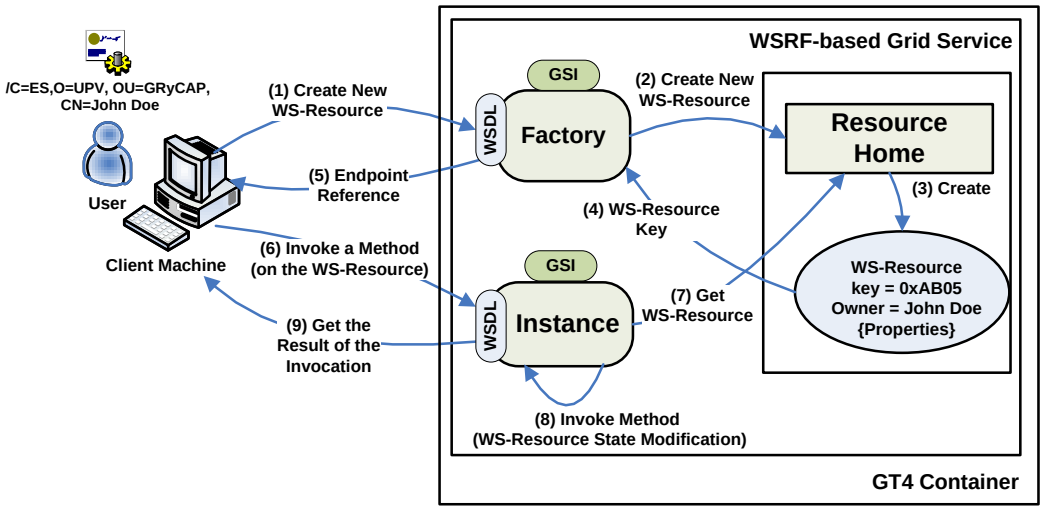


Fig. 1. Architecture of a WSRF-based Grid Service implemented under GT4.

GT4 container. The GT4 container includes a Java Web services application server, providing the Grid services with an appropriate runtime environment.

One of the important features provided by GT4 is the Grid Security Infrastructure (GSI) to provide authentication and authorization mechanisms based on public-key cryptography. This allows controlled access to the service functionality (methods) as well as privacy and data integrity.

Following the diagram in Figure 1, first of all, the client contacts the Factory service to create a new WS-Resource (step 1). At this point, the client-side credentials are checked for authentication and authorization through GSI-based mechanisms. Then, the WS-Resource is created by the Resource Home (steps 2,3) and it receives, as a unique identifier, the WS-Resource key (step 4). This identifier is combined with the Uniform Resource Identifier (URI) of the Instance service to produce an Endpoint reference (EPR) (step 5). The EPR uniquely identifies the location of both the Instance service and the corresponding WS-Resource which holds the state. At a later stage, the client can invoke a method (step 6) on the Instance service using the EPR. This may modify the WS-Resource state (steps 7,8) to produce some results that are returned to the client (step 9).

If we consider a stateless Grid service (i.e., a traditional Web service), then replication becomes a trivial task that can be solved with multiple deployments of the service on different GT4 containers. As no shared data among replicas exist, no coordination is ever required. However, replicating Grid services requires, at least, managing these problems:

- Group membership: The Grid services that share the same state through replication form a group of replicas, where all of its members agree on a common topology. As this group could possibly be dynamic, there must

exist mechanisms to enter and leave the group at any time.
- Consistency: There must be a coordination protocol to maintain a coherent state of the replicas (i.e. the same WS-Resource state).

So, a Grid service *per se* is a passive entity. Replicating a Grid service means replicating its state, that is, its WS-Resources. Therefore, multiple WS-Resources will exist with the same state on the different Grid services deployed in a distributed infrastructure.

3 The Grid Service Replication Library

The main aim of the library is to provide an existing WSRF-based Grid service with automatic replication of WS-Resources in a transparent manner to the users and integrated with its GSI-based security mechanisms. It can operate both in a primary-backup approach, where a master replica is responsible for processing all the requests, and in a multi-primary scheme, where any replica can process a request.

The library uses passive replication, where a request is processed on a single replica and then the new state of the WS-Resource is transferred to the other replicas. To ensure sequential consistency, that is, all the replicas traverse the same state changes in the same total order, we have implemented a semaphore or lock-based approach based on Grid services. This guarantees mutual exclusion to a common resource (the state of the WS-Resource).

One important aspect of this library is that it does not introduce any central software component to manage replication. This avoids both introducing a bottleneck in the system and a Single Point of Failure which could bring the whole system to a halt. Instead, all the replicas have the same functionality implemented and distributed algorithms running on all the replica nodes have been employed. As the proposed library mainly relies on distributed algorithms with no central agents, its scalability is not limited by any software component.

Figure 2 describes the main functionality of the replication library. The scenario starts with the deployment of the same Grid service, modified to include the functionality of the replication library, in different GT4 container instances. This is commonly done among different machines to cope with the hardware failures. However, the replication library also operates on multiple instances of the Grid service within the same container, each one attached to a different port. This enables the Grid administrator to save hardware but still introducing high availability facing possible container and Grid service failures.

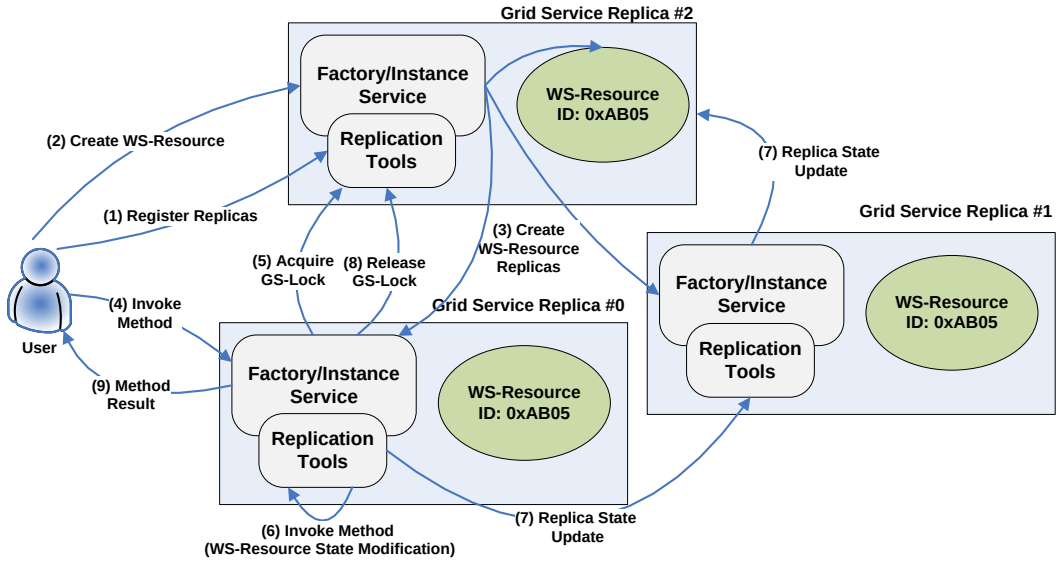


Fig. 2. Principal functionality of the replication library.

First of all, the user chooses any of the replica Grid services and registers the other replicas by specifying their Endpoints (step 1). Next, the user requests this replica to create a WS-Resource (step 2), which stores the state. This WS-Resource is then created on the other replicas with exactly the same identifier (key) and the same ownership (step 3). This requires credential delegation among the different replicas and appropriate configuration of stubs to comply with the GSI-based security already existing in the Grid service. Sharing the same identifier simplifies the client-side access to the different replicas, which can be achieved with a simple modification of the Instance service URI in the EPR. In addition, all the replicas agree on a common topology so that they have a full and shared view of the group. At the end of this process, the same WS-Resource exists on the different container instances. The library is able to replicate multiple WS-Resources using the same approach.

If the methods on the Grid service only read the state of the WS-Resource then synchronization and updating protocols are not required among the replicas. However, consider the case depicted in Figure 2, where the user invokes a method that results in a state modification (step 4). To ensure coordinated access to the WS-Resource state we have implemented a lock-based approach, via Grid services, called GS-Lock. Initially, all the replicas agree on the one with the largest index to be the GS-Lock controller. Before a Grid service executes an operation that modifies the WS-Resource, it must acquire the GS-Lock. This is a blocking operation that returns immediately if the GS-Lock is available but it hangs the caller until it becomes available and access is granted to it (step 5). This may result in a small client-side delay, but the client is completely unaware of the synchronization and replication process.

Once the GS-Lock has been acquired, it is time to perform the operation that

modifies the state of the WS-Resource. When a modification of this state is detected, the replication library is in charge of propagating the state update to the other replicas using a coordinated distributed algorithm that depends on the topology (step 7). Once all the replicas have been successfully updated, the GS-Lock is released (step 8) and the result of the method is returned to the user (step 9). This allows another operation to gain access to the GS-Lock and proceed with the WS-Resource state modification. The GS-Lock is acquired only for a specified amount of time, large enough to perform the whole state update among the replicas. This timeout allows the GS-Lock controller to release it in case the replica that acquired it fails. If the GS-Lock controller replica fails, the other ones will notice it when they try to acquire the GS-Lock and, therefore, will choose the new replica with the largest index acting as a controller.

3.1 Efficient state update: Considering replica topologies

Updating the state among all the replicas is the most time-consuming operation of the replication process. These depends mainly on three factors: Firstly, the amount of data involved in each state update operation, which exclusively depends on the replicated Grid service. For comprehensive states, this could result in large messages being exchanged among the replicas. Secondly, the network capacity (i.e., bandwidth and latency) largely determines the state update speed. Finally, the state update time is also influenced by the number of replicas.

We have implemented two different replica topologies and evaluated its benefits and caveats.

3.1.1 Ring-based Topology

The ring topology has already been depicted in Figure 2. All the replicas have a complete view of the topology, implemented by means of a replicated array with information about all of them (Factory and Instance service URIs). Their location in the array determines their topology identifier. This allows to efficiently build distributed algorithms for different topologies.

The distributed algorithm only requires that each replica node has the aforementioned array with as many entries as the size of the group. Each replica knows its corresponding location in the array and, therefore, it is trivial to find out the information about its neighbor node. The ith replica is considered to be adjacent to the $(i-1)$th replica. The array is considered a circular structure where the 0th replica is adjacent to the $(n-1)$th replica, for a group of n nodes. The logic of the distributed algorithm is confined to a single method

that is called whenever a replica node receives a state update request. Therefore, all the replicas share the same replication code which implements the following behavior.

The Grid service replica on which the state changes, named the source replica (replica 0 in Figure 2), sends the state update to its neighbor replica, according to the ring topology and its topology identifier. This one applies the new state and forwards it to its neighbor replica. The notifications of state update among the replicas are delivered via method invocations which include the content of the new state and additional information such as the identifier of the source replica. This process iterates until the state update reaches the source replica again (omitted in Figure 2). At this point the source replica knows that all of them are in a consistent state. Therefore, the GS-Lock can be released for other operations to proceed. Notice that the state update is a distributed algorithm running on the Grid service replicas.

Handling the failure of a replica during the state update process is also achieved via a distributed algorithm. When a replica tries to update the state of another one and it detects that it has failed, even if it is because of network errors, it is responsible for notifying the other replicas about the failed one. All of them discard the failed replica in the topology array, and rearrange the ring so that the state update bypasses the failed replica. If the source replica fails, then the replica that detects the failure is also in charge of releasing the GS-Lock to avoid waiting until the timeout of the GS-Lock expires.

Joining the ring topology can be achieved by registering a new replica to be part of the ring. This operation can be performed by either the user or a Grid administrator. Being a distributed algorithm, this operation can be executed on any of the existing replicas. This causes a topology rearrangement and the new replica receives the latest state of the WS-Resource, from the replica that executes the operation, so that they all keep consistent.

Notice that a ring modification is not an expensive operation. It only requires a lightweight method invocation, specifying the failed replica or the new one, in every node of the ring and then each replica locally updates its own array of replicas to represent the new ring topology. The usage of distributed algorithms and the lack of a master replica simplifies the operations.

The ring-based topology results in a replication time that is linear with the number of replicas. In order to reduce the time required for the state update, we have also implemented a hierarchical topology based on modified Complete Binary Trees [13].

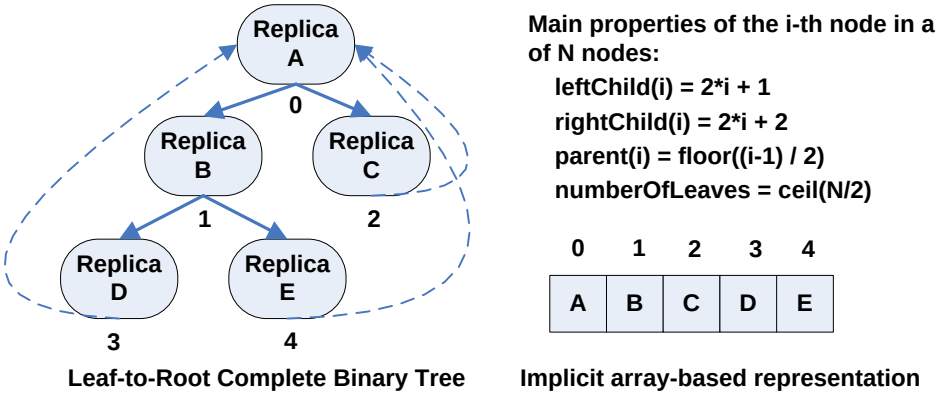


Fig. 3. A Leaf-to-Root Complete Binary Tree (CBT) topology of Grid service replicas. The implicit array-based representation represents a level-order traversal of the tree.

3.1.2 Leaf-to-Root Complete Binary Tree Topology

As the number of replicas grows, the amount of time dedicated to state propagation increases linearly. This may impose a serious limitation for replicated Grid services with a large number of replicas and/or a huge amount of data to be transferred during each state update.

In order to reduce this time, a linear relation among the replicas must be abandoned in favor to a hierarchical approach. This way, we have employed a Leaf-to-Root Complete Binary Tree (CBT) topology. In a CBT (see Figure 3) all the levels, except perhaps the last, are filled (i.e., full of nodes). In the last level, all the nodes must be located at the leftmost side of the tree. These special trees can be implemented using an implicit array-based representation. These allows one to compute several properties using simple formulae, such as obtaining the right and the left child of the *i-th* node ($2 * i + 1$, $2 * i + 2$) or computing the number of leaves of a tree with N nodes ($\lceil \frac{N}{2} \rceil$). This allows to implement efficient distributed algorithms, running on the different replicas, based on simple array-based operations.

Therefore, each replica has an array representing the tree-based topology (see Figure 3). This array represents the level-order (a.k.a. breadth-first order) traversal of the complete binary tree. In our implementation, the leaves are conceptually linked to the root replica to implement the state update process. Consider how this protocol works under a typical operation:

(1) The user modifies the WS-Resource state located at Grid service replica B in Figure 3. The GS-Lock must be acquired so that no other operation can proceed before the state update propagation finishes.
(2) The source replica (the one which originates the resource update (B)) notifies the root replica (A) about the state update.
(3) The state update proceeds downwards from the root replica until it

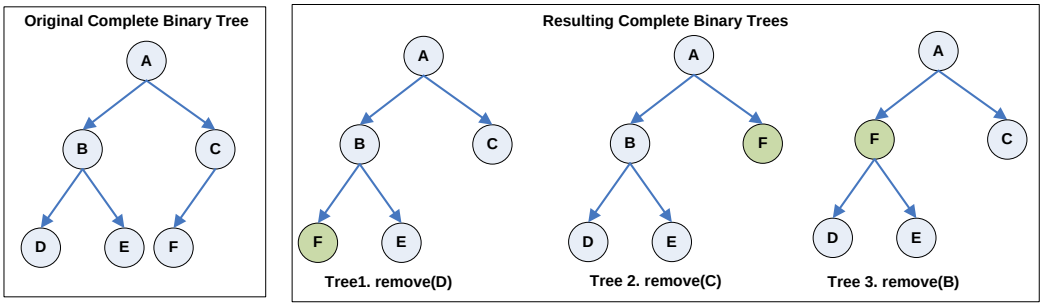


Fig. 4. Structural changes in a Complete Binary Tree after erasing a replica with 0, 1 and 2 children.

reaches the leaf nodes. This is done with parallel update streams which proceed simultaneously for the different branches of the tree. Therefore, replica A notifies the WS-Resource state update to replicas B and C while replica B informs replicas D and E. The exchanged message among replicas includes information about the source replica. This allows the source replica to skip its own state update.

(4) As replicas C, D and E are leaf nodes, they notify the root replica that they have been updated. As the root replica can easily compute the number of leaf nodes, it certainly knows when the whole tree has been updated.

(5) Once the whole tree of replicas share the same WS-Resource state, the GS-Lock can be released by the root replica and another operation can be executed.

The root replica may seem a partial bottleneck as the leaf nodes must notify their update. However, these notifications do not include resource state information so this turns out into a lightweight method invocation.

The state update process in a complete binary tree with N replicas requires a time in the order of $\theta(log_2(N))$, which is the upper bound of the height of the tree. This results in a significantly faster approach than using a ring-based topology for a moderate number of replicas.

Handling either replica or network errors has again be treated using a distributed algorithm. When one replica invokes the state update method of another one and the operation fails, then it is responsible of notifying the others about the failed node. This is achieved via a lightweight method invocation on each replica, which locally modifies the array of replicas stored at every node to create the new topology. Being a distributed algorithm, where all the replica nodes execute the same code, the resulting topology is shared by all replica nodes. Next, the state update process starts again from the (possibly new) root replica. Removing a replica from this topology requires the structural integrity of the CBT to be preserved. Then, this reduces to a problem of removing a node in a CBT, depicted in Figure 4:

- If the failed replica is a leaf node (D in the original tree), then the last node in the implicit array representation (F in the original tree) takes its place, thus obtaining Tree 1. If the failed replica is actually the last node, then it simply disappears from the topology.
- If the failed replica has only one child (C in the original tree), its child is adopted by its parent node (A in the original tree), thus obtaining Tree 2. Notice that due to the CBT properties, the only node with a single children is always the parent of the last node.
- If the failed replica has two children (B in the original tree), its place is occupied by the last node, thus resulting in Tree 3.

Therefore, to satisfy the structural integrity of the CBT, the last replica node should take the place of the failed one. As such, using the properties of the CBT, an efficient distributed algorithm can be developed to handle the failure of replicas by altering the topology at runtime.

Joining the group can be achieved by registering a new replica at any other one. This one contacts the root replica and the topology update proceeds downwards in the tree until it reaches the leaf nodes which again notify the root replica. Every replica updates their topology using the same rules so that, in the end, they all share a common topology. The new replica will always appear as a new leaf in the CBT topology.

4 Application to a Service-Oriented Metascheduler

In order to test the functionality of the replication library in a real Grid service, we have included replication capabilities into GMarteGS [14]. This is a service-oriented metascheduler which uses the resource brokering functionality of GMarte [15] to achieve efficient task allocation on computational Grids. This section first briefly describes GMarteGS, then explains how to plug the replication library into a WSRF-based Grid service and, finally, details the specific adaptations performed to suit this application.

Notice that that replication library can be applied to generic WSRF-based Grid services and, therefore, this is just an application example. The results obtained can be easily extrapolated to other Grid services.

4.1 A brief introduction to GMarteGS

GMarteGS is implemented using GT4 and WSRF to enable internet-based, multi-user support to achieve successful execution of high performance com-

puting applications in computational Grids based on the Globus Toolkit. It uses GSI-based security to offer user authentication and authorization, privacy among different users and data integrity in client-service communications. It also handles the data transfers performed between the client machine and the remote computational resources where the executions actually take place.

In GMarteGS, each WS-Resource hosts information about a different metascheduling session. This information includes the description of a set of computational tasks (i.e. their executable files, the arguments, the requirements for execution, etc.), a computational infrastructure represented by a set of computing elements to be employed for the execution of the tasks, and a specific configuration of the metascheduler (i.e. task allocation policy, number of threads, etc.).

This service-oriented metascheduler is being employed for the execution support of high performance applications in the area of cardiac electrical activity simulation [16]. These applications require large computing requirements and long execution times, in the order of days. Therefore, fault-tolerant mechanisms need to be employed to guarantee successful execution. The usage of a service-oriented metascheduler enables the users to submit executions via graphical applications and monitor their evolution while they are being executed on a Grid infrastructure.

Fault-tolerance should be faced at multiple levels. At a higher level, GMarteGS is isolated from task execution problems, which are handled by the GMarte metascheduler. This component manages task and computational resource failures by means of rescheduling and also retries the failed data transfers a certain number of times (see [15] for details). GMarteGS also implements fault-tolerant techniques. The client and the Grid service are largely decoupled so that failures in the client do not affect the service. In addition, the service uses WS-Resource persistence so that, in the case of container failure, its state can be automatically recovered after booting a new container (see [14] for details).

However, high availability and sustained service availability can only be achieved through replication. This allows a client to connect to a different replica, should the one he is using fails, finding a consistent state, and without requiring the intervention of the Grid administrator.

4.2 Plugging the replication library into a Grid service

Introducing the replication capabilities into an existing WSRF-based Grid service should have minimal impact in its coding. To achieve this feature, we have relied on two features. Firstly, object-oriented programming allows to confine

the replication code in the developed library and inherit that functionality in the current service. Secondly, the usage of the Operation Providers [11] functionality of the GT allows to easily use the replication capabilities of the library from an existing service. This development approach allows the programmer to implement some generic functionality which can then be plugged into other Grid services with minimal developer intervention.

These are the main modifications required in the Grid service to have available the replication functionality:

(1) The WSDL of the Factory and the Instance service must extend (via *wsdlpp:extends*) those provided by the replication library. This enables both services to automatically publish the required methods for replication (resource state update, topology update, group joining, etc.).
(2) The deployment descriptor of the Grid service (*deploy-server.wsdd*) has to specify the operation providers available in the replication library, for the Instance and the Factory services. This enables both services to automatically receive the implementation of the replication methods.
(3) The Factory service, the Instance service, the ResourceHome and the Resource main Java classes should inherit from the counterparts provided by the replication library. This enables both services to automatically receive the implementation of other support methods required for replication. The developer may also choose the replication topology to be employed, but this can also be specified at runtime.
(4) After the state of the WS-Resource changes, the developer must invoke the method that starts the distributed state update process according to the specified topology. The state of the WS-Resource exclusively depends on the Grid service and it is currently represented in the shape of a String.

4.2.1 Actions performed in GMarteGS

GMarteGS is aware of the metascheduling process by means of a callback method that uses GMarte to indicate the changes in the state of tasks as they are being processed (i.e. unassigned, allocated, staged in, active, staged out, completed, failed, etc.). A change in the state of a task means a change in the metascheduling session and, as a consequence, a change in the WS-Resource state which requires to be reconciled with the other replicas. During the first moments of a metascheduling session the tasks change very often their state until they are actually running. Therefore, we have implemented basic contention mechanisms to reduce the replication overhead. For example, it is possible to use a replication period to wait a minimum amount of time between two replica state updates. Of course, this approach depends on the Grid service, but it results in a trade-off. As this time increases so it does the chance of accessing an inconsistent replica by different users.

The state of the WS-Resource can be represented by an XML document that captures the state of the metascheduling session, that is, the state of the tasks, the computational resources employed and the metascheduling configuration. The description of a task includes, among other attributes, the location of the executable file, its command-line arguments, the input files that it requires for execution and its computational requirements. This is dynamically generated whenever a state change is detected by GMarteGS and it represents the current state of the WS-Resource. This state is exchanged with the other replicas in the shape of a String and later unmarshalled to recreate an appropriate Java object on which to execute the user operations regarding the metascheduling session. Although XML Data Binding tools exist to simplify the actions, this approach typically requires fine-tuning the XML documents whenever changes are introduced in the representation of tasks or computational resources.

To solve this problem, we have alternatively used a different approach. Whenever a state change is detected, a binary duplicate of the metascheduling session is in-memory generated via the serialization mechanisms provided by Java. This information is then encoded in Base 64 so that it allows a String representation that later is compressed using the standard Zip utilities provided by Java. We have measured that this compression step allows reducing the information up to a factor of 3. This procedure allows to have a binary duplicated object in another Grid service replica that is shielded against changes in the representation of tasks and resources.

Concerning the data files required by the metascheduling process, mainly the executable files and dependent input archives, these can also be replicated using the Data Replication Service [17] provided by GT. This service uses the Reliable File Transfer service to coordinate data transfer among different GridFTP servers. These servers could be available in the machines where the replicas exist or reside in a specialized Data Grid. Its usage is out of the scope of this paper.

4.2.2 Performance evaluation

To evaluate the performance of the library, we have considered a scenario composed of five machines with computing capabilities enough (at least Xeon 2.0 GHz with 1 GByte of RAM), linked to a Gigabit Ethernet network. Each machine has the GT version 4.0.8 deployed with the GMarteGS metascheduler linked with the replication library. Another machine acts as the client submitting a certain number of jobs to one of the replicas. We have disabled the usage of GSI Secure Conversation to reduce the overhead during the state update process among the replicas.

As the metascheduling session progresses, changes in the state of the session

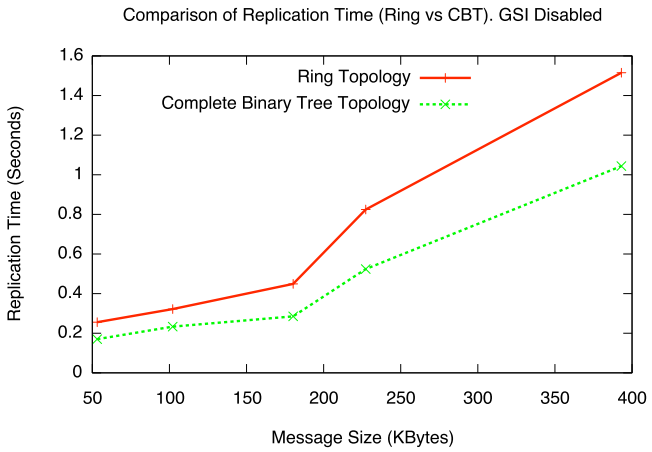
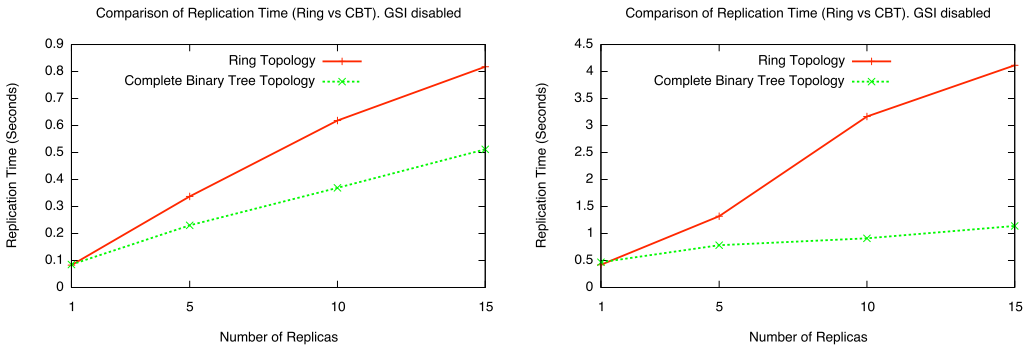


Fig. 5. Comparison of the average replication time among five replicas with two different topologies.

(i.e., the state of its tasks) are updated at the other replicas. The time since the source replica detects a new state until all of them are in coherent state is considered to be the replication time. For these executions we have not used a contention mechanism for the state update. Therefore, changes in the state of the jobs from the session trigger a state update process. However, as a precautionary measure, only the latest state of the session is exchanged if several update requests are received while an update is still on course. This mainly causes to stress test the replication library by performing almost one replication after another during the metascheduling process.

Figure 5 compares the average time required to update the state of five replicas when using a ring topology compared to a Leaf-to-Root CBT topology during a metascheduling session of computational tasks. The message size represents the amount of information exchanged among the replicas which represents the state of the WS-Resource, that is, the metascheduling session. In particular, the figure depicts the amount of information corresponding to 1 task (53.16 Kbytes), 10 tasks (102.14 Kbytes), 50 tasks (227.32 Kbytes) and 100 tasks (393.10 Kbytes). As this information uses a Base-64 representation and is compressed there is no longer a linear relation between the number of tasks and the message size.

It can be seen that the CBT topology is able to reduce the replication time among the five replicas up to a 32%. In fact, this topology allows for concurrent update streams through the tree of replicas which no longer require all updates to be performed sequentially, as in the case of the ring topology. It is important to point out that CBT topology is expected to even increase its advantages over the ring topology as the number of replicas grows. However, for the particular case of the GMarteGS metascheduler, it not necessary to have a large number of replicas to gain high availability.



(a) Local Area Network (b) Wide Area Network

Fig. 6. Comparison of the average replication time among different replicas with two different topologies using a message size of 102.14 KBytes (10 tasks).

The CBT topology can achieve global state update in logarithmic time with respect to the number of replicas while the ring topology can only achieve it in linear time. Finally, note that GMarteGS uses GSI security for data protection, but the replication library has been specially configured in this tests to disable GSI security. This reduces the performance penalties involved in each method invocation of the replication library due to the establishments of the security contexts.

Figure 6.a compares the average replication time for a different number of replicas using the two topologies. The nodes exchange state messages corresponding to 10 computational tasks. With only two replicas, the amount of time for replication is very similar, as no performance can be gained by taking advantage of the topology. These slight changes are due to the implemented distributed protocols. However, as the number of replicas grow, so it does the difference in the replication time between the CBT and the ring topology.

To investigate the effects of a Wide Area Network (WAN) in the performance of the replication library we have simulated a high latency environment among the replicas. A delay in each state update message between two replicas has been introduced with a pseudorandom uniformly distributed value between 0 and 300 milliseconds. The results are shown in Figure 6.b. It can be noticed that the CBT topology outperforms the Ring topology. As the CBT topology is able to use parallel state update streams, the influence of the high latency environment has a much lower impact than with the ring topology.

The use case employed does not affect the ability of the replication library. It has only a direct influence in the amount of replications performed and the size of each state update message. Therefore, this functionality can be included into other kind of Grid Services. According to the results, the Leaf-to-Root Complete Binary Tree topology stands out as a candidate topology to be employed for the efficient replication of WSRF-based Grid services.

5 Conclusion

This paper has described a library that replicates WSRF-based Grid services in order to achieve service availability by means of fault-tolerance and high availability. The library allows to seamlessly replicate WS-Resources among different GT4 containers and maintain a coherent and shared state. This is achieved by an implementation of Grid-based locks, to avoid concurrent access to the shared WS-Resource state, and the usage of topologies to allow efficient state update propagation. In particular, the ring-based and the Leaf-to-Root Complete Binary Tree topologies have been studied. The latter accomplishes the global state update in logarithmic time with respect to the number of replicas.

The future works involve incorporating and evaluating SOAP with Attachments (SwA) as a transport mechanism to exchange replica state. As the amount of state data increases, this could certainly enable to reduce the size of the messages exchanged [18]. Also, we plan to focus on automatic deployment of Grid services. These would enable the system to automatically deploy new replicas on demand to better cope with failures without any user or Grid administrator intervention.

References

[1] I. Foster, C. Kesselman, Globus : A metacomputing infrastructure toolkit, International Journal of Supercomputer Applications and High Performance Computing 11 (2) (1997) 115-128.

[2] I. Foster, Globus Toolkit Version 4: Software for Service-Oriented Systems., in: LNCS (Ed.), IFIP International Conference on Network and Parallel Computing, Vol. 3779, 2005, pp. 2-13.

[3] E. Huedo, R. S. Montero, I. M. Llorente, Evaluating the reliability of computational grids from the end user's point of view, J. Syst. Archit. 52 (12) (2006) 727-736.

[4] E. Byun, S. Choi, M. Baik, J. Gil, C. Park, C. Hwang, MJSA: Markov job scheduler based on availability in desktop grid computing environment, Future Generation Computer Systems 23 (4) (2007) 616-622.

[5] P. Kunszt, E. Laure, H. Stockinger, K. Stockinger, File-based replica management, Future Generation Computer Systems 21 (1) (2005) 115-123.

[6] M. Lei, S. V. Vrbsky, X. Hong, An on-line replication strategy to increase availability in data grids, Future Generation Computer Systems 24 (2) (2008) 85-98.

[7] H. Gadgil, G. Fox, S. Pallickara, M. Pierce, Scalable, fault-tolerant management in a service oriented architecture, in: HPDC'07: Proceedings of the 16th international symposium on High performance distributed computing, ACM, New York, NY, USA, 2007, pp. 235-236.

[8] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, R. Schlichting, Fault-tolerant grid services using primary-backup: Feasibility and performance, in: IEEE International Conference on Cluster Computing, 2004, 2004, pp. 105-114.

[9] A. Luckow, B. Schnor, Service replication in grids: Ensuring consistency in a dynamic, failure-prone environment, in: IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008., 2008.

[10] I. Foster, et al., The Open Grid Services Architecture, version 1.5 (March 2006).

[11] B. Sotomayor, L. Childers, Globus Toolkit 4: Programming Java Services, Morgan Kaufmann, 2005.

[12] WSRF - The WS-Resource Framework, URL: `http://www.globus.org/wsrf`.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, 2001.

[14] G. Moltó, V. Hernández, J. M. Alonso, A service-oriented WSRF-based architecture for metascheduling on computational grids, Future Generation Computer Systems (International Journal of Grid Computing) 24 (4) (2008) 317-328.

[15] J. M. Alonso, V. Hernández, G. Moltó, GMarte: Grid middleware to abstract remote task execution, Concurrency and Computation: Practice and Experience 18 (15) (2006) 2021-2036.

[16] J. M. Alonso, J. M. Ferrero (Jr.), V. Hernández, G. Moltó, J. Saiz, B. Trenor, A grid computing-based approach for the acceleration of simulations in cardiology, IEEE Transactions on Information Technology in Biomedicine 12 (2) (2008) 138-144.

[17] Data replication service, The Globus Toolkit Documentation, `http://www.globus.org/toolkit/docs/4.2/4.2.0/data/datarep/index.html`.

[18] D. Zhang, P. Coddington, A. Wendelborn, Binary data transfer performance over high-latency networks using web service attachments, in: E-SCIENCE'07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing, IEEE Computer Society, Washington, DC, USA, 2007, pp. 261-269.