Implementing a Distributed Execution Service for a Grid Broker

Flavio V. D. de Figueiredo, Francisco V. Brasileiro, Andrey E. M. Brito

Universidade Federal de Campina Grande
Departamento de Sistemas e Computação
Laboratório de Sistemas Distribuídos
Av. Aprígio Veloso, 882 - Bloco CO, Bodocongó
CEP 58109-970, Campina Grande - PB, Brasil

flaviov@lsd.ufcg.edu.br, {fubica, andrey}@dsc.ufcg.edu.br

Abstract. *Grid middleware such as OurGrid offer solutions for executing parallel tasks on a grid system. In such systems, users submit their applications for executions through a client broker. MyGrid is the client broker used for the OurGrid system; it is in charge of managing task executions that a user has submitted. Although the broker is able to detect task failures and reschedule them, MyGrid itself constitutes a single point of failure from the user perspective. If it fails, all knowledge of task executions is lost. Moreover, MyGrid is also a bottleneck, since hundreds, or even thousands, of executions could potentially be spawned by an application and need to be managed at the same time by a single broker. In this paper we present the design and implementation of a fault-tolerant distributed execution service that allows for load balancing and improves MyGrid performance. A checkpointing mechanism is used to ease the implementation of the service and to further increase system reliability.*

1. Introduction

A grid system comprises a set of distributed, heterogeneous resources, such as: personal computers, storage space, clusters, servers, etc., connected through a network. Grid computing has been proposed to cater for the high computational demand in different areas of research, such as physics, biology, astronomy and computer science itself. A grid user gains access to a grid system by using a grid middleware; with it the user can access a variety of services, such as: resource management, security services, monitoring services and execution services [Foster et. al., 2001][Foster and Iamnitchi, 2003].

The OurGrid system [Cirne et. al., 2006] is a peer-to-peer grid middleware in which sites lend their idle resources to gain access to resources from other sites when they need. The goal behind the use of the system is the execution of Bag of Tasks (BoT) applications, those parallel applications in which tasks do not need to communicate among each other. The OurGrid middleware has been used to support OurGrid's free-to-join community, which is in production since December 2004 (see http://status.ourgrid.org/ for a fresh snapshot of the running system).

Three main services compose the OurGrid system; these are the brokers, resources and peers. Peers are in charge of delivering resources, known as Grid Machines or simply GuMs, to the broker, known as MyGrid [Cirne et. al., 2003]. MyGrid runs in the user machine, named the *home machine*. GuMs offer an environment for the execution of tasks, computation requested by the broker is done on these . A machine is available as a GuM if the User Agent service is running at it, this service is available for Linux, Windows or using OurGrid´s sandboxing solution called SWAN[Cirne et. al., 2006]. Peers also exchange resources amongst themselves, using the Network of Favours incentive mechanism [Cirne et. al., 2006]. An OurGrid site is a Local Area Network with Peers, Brokers and GuMs running on the machines that compose it.

Current fault-tolerance in OurGrid is only available for the GuMs, such failures are treated with checkpointing mechanisms [OurGrid Team, 2006] or with task replication [Cirne et.

al., 2006][Cirne et. al., 2003]. Home machine failures are not dealt with in any way, though proposals have been made to deal with such [Silva and Chao, 2004]. If it fails, all information on replica executions is lost. The fact that execution information is lost makes such failures have a great cost, since a MyGrid user will have to restart all pending executions, even if those have not failed. There is also a scalability problem with the current architecture; the home machine has to manage all pending executions, limiting the maximum number of executions that can be executed at a time.

In this paper we present the design and implementation of a distributed execution service for MyGrid, which is able to recover from failures and also load balance the management of pending tasks. We have decoupled the service from MyGrid and implemented a checkpointing mechanism that is able to tolerate faults in the home machine, at the same time that allows disconnected operation, ie. a user may submit an application, log off, and later log in the system to access the results of the computation. With these mechanisms in place we hope to bring scalability and reliability solutions to MyGrid. Though implemented in the OurGrid system, this solution can be applied on other grid architectures that have a similar execution service as MyGrid´s.

The rest of this paper is structured in the following way. Section 2 describes MyGrid's current architecture, showing the life-cycle of a BoT application execution. Section 3 presents the design of a distributed execution service for MyGrid, named the Replicated Replica Executor, RRES. In section 4 implementation details for each of the RRES modules are described. Section 5 concludes the paper with our final remarks in directions for future work.

2. MyGrid's Current Architecture

The MyGrid broker is the user interface for the OurGrid system. Whenever an application (or job) is submitted by the user, the broker contacts one of the peers in the peer-to-peer grid, named the *local* peer, and submits a request for the number of GuMs required to executing the job. The local peer routes the query to other peers and returns any available GuM to the broker. As the local peer starts providing the broker with GuMs, the broker initiates the scheduling of tasks in these GuMs. To improve performance tasks may be replicated and executed in parallel in different GuMs. When one of the replicas finishes its execution, the results are collected and the other replicas of the same task are aborted. Soon after the job is completed, ie. all tasks have finished their execution, the broker returns the received GuMs to the local peer, which in turn returns each GuM to the peer that manages it. In summary, the broker encapsulates a scheduling service (Scheduler) that matches replicas of a task to GuMs and a replica execution service (Replica Executor or RE), which manages the execution of these replicas on the allocated machines.

The execution of a task is performed in three phases: the INIT phase in which file transfers occur from the home machine to the GuM on which the task will execute; the REMOTE phase in which computations of the task are executed on the remote GuM; and, the FINAL phase in which files are transferred from the GuM to the home machine.

The RE creates a thread for each new replica that it needs to execute. These threads contact the remote GuM to perform the actual task execution. Each replica execution has to run the three phases of a task to conclude properly. To avoid replicas from overwriting each others outputs, when one of these replicas finishes the REMOTE phase, MyGrid aborts the rest and concludes the task, executing its FINAL phase. Also, to provide fault tolerance, a replica that fails is executed again a few times. If the maximum number of executions is reached, MyGrid will declare the task as failed; also, if a user cancels a job then its tasks and replicas are set as cancelled. After the replica execution is concluded RE will send a status to the Scheduler, according to the way the replica ended its execution (finished, failed, aborted or cancelled) [Cirne et. al., 2003]. The scheduler will then treat the result, according to its status, if necessary it also contact's the local peer to ask for more resources or to cancel requests for resources. Figure 1 presents a graphical representation of the elements that constitute MyGrid and the interactions

amongst them.

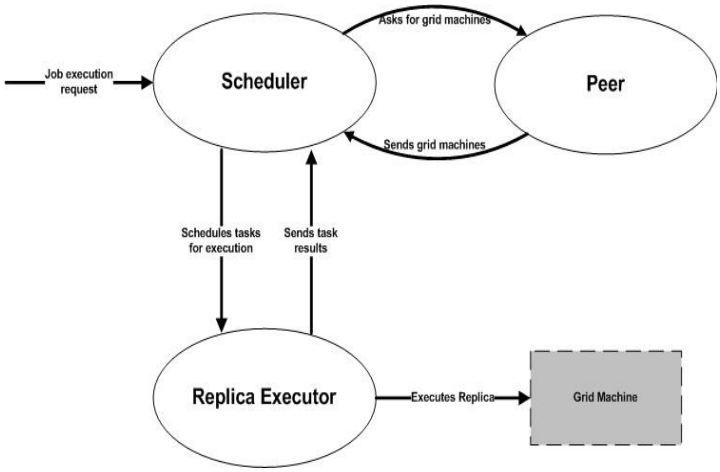

Figure 1. MyGrid's Current Architecture

3. A Distributed Execution Service

The Replicated Replica Executor Service (RRES) is a distributed execution service for MyGrid. It is composed of various REs that work individually and execute in an address space that is different from that of the broker. Each of these REs will run on a different machine, named Replica Executor Machine (REM). By decoupling the broker from the RRES we allow several brokers to share a single RRES. The REES is an execution service for brokers running on the same site, being used for a single site we can assume that REMs will run on machines that are located on the same local area network (the same one that composes the site).

The design of the RRES uses a master-slave replication model. The Master's job is to schedule executions to Slaves and communicate with MyGrid, while the Slave's job is simply to execute replicas. There is only one active Master process running inside the system, only this Master is known by the MyGrid brokers. Slaves will be able to contact the Master but not each other. Slaves will also be able to communicate with MyGrid, but such communication is unilateral, Slave to MyGrid. Masters and Slaves are independent processes; they can be started separately, but in the RRES each REM will start both of them. The Slave process will be ready to execute replicas, but the Master process starts in *sleeping mode* and does nothing until it has to assume Master functionalities of the system. Being this way, if a Master fails another machine can promptly assume its position. Also, if there is only one machine running the server, than it can perform both functionalities.

Both the broker and joining Slaves discover the Master by reading a configuration file stored in the Network File System, NFS. This configuration file is created by the first REM that is initiated, being the first, this REM will immediately wake up its Master process. The file will contain only information necessary to lookup this REM's host name and port. When another REM starts up it will not alter the configuration file. From now on we shall refer to Master as the REM who is performing the Master's functionalities within the RRES and as Slaves every other REM. The Scheduler module will be mentioned as MyGrid only.

The RRES will, from the MyGrid's perspective, work the same way as it does in the design discussed in the previous section. Obviously, since they run in different address spaces, the actual mechanism used for them to communicate should allow for appropriate inter-process communication. For instance, in our implementation we use Remote Method Invocations (RMI). The other difference is that a different REM, most likely unknown to MyGrid will probably execute the replica being sent for execution.

An execution on the new architecture is as follows: MyGrid brokers will submit jobs to the Scheduler the same way as they do in the previous design. The scheduler will now contact the

Master, sending it information about this new request. When the Master receives new execution requests, information about the execution is sent to one Slave in the system, this Slave is chosen based on a simple load-balancing function. The chosen Slave will then execute the Replica as it does in the previously discussed architecture and then send the result back to the Master, who will then send back the result to MyGrid (see Figure 2).

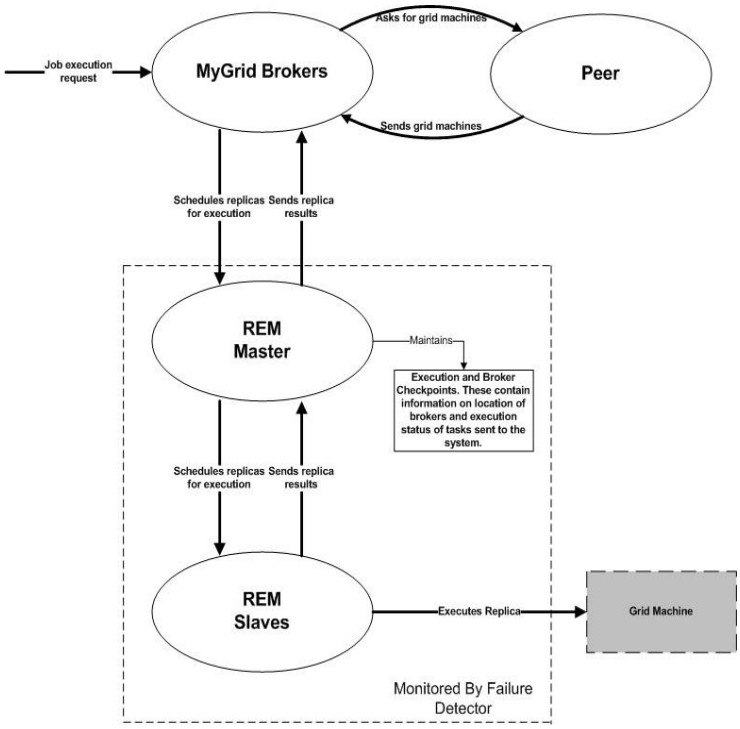

Figure 2. RRES Architecture

Failures in MyGrid will no longer affect the RE, and vice-versa, since they now run independently. However, connection problems now rise, in the previous architecture they ran in the same address space and such a problem was impossible. To solve communication problems we must offer a reliable communication service for the contacts made within the system. There must also be a failure detector monitoring the REMs in the system, so in case a Slave fails we can reschedule its current executions, in case the Master fails a Slave must take its place. MyGrid's home machine failure and Master REM failures are solved using checkpointing mechanisms. Also, to guarantee that the scalability problem is solved we must guarantee that each slave within the RRES has approximately the same load of executions.

The RRES solution is similar to the Master-Worker (MW) framework describe in [Goux et. al., 1999] and [Goux et. al.,2000]. Similarly task executions are submitted to the Master who takes care of dividing the work to the Workers. The difference is that the Workers are the ones that actually perform task executions. This creates a scalability problem. The Master must forward the whole task to the Worker; tasks with large data to be executed can cause the Master to become a bottleneck [Goux et. al., 2000]. Also, a task-dependency issue arises; applications that have task-dependencies implies on idle Workers while the dependencies are not computed [Goux et. al., 2000].

Some of OurGrid´s characteristics and the RRES features are enough to handle these issues. BoT applications have no dependencies by nature, addressing the task-dependency issue. The scalability issue is addressed by the fact that no real computation is done by RRES Slaves, the GuMs are the ones who perform computation. Also, with the NFS, no data transfer is required within the RRES. Slaves already have access to all the data that needs to be transferred in order to perform task executions, each Slave will handle these transfers according to tasks that are scheduled to them.

3.1. Load Balancing

A Load Balancer guarantees that every Slave will have approximately the same load. Since each REM is composed of one Slave and one Master, we can assume that each REM will have approximately the same load according to replica executions, because only the Slave process is always active. The load balancer depends on the ordering of replica executor ids. We have chosen to identify our REMs using the last byte of their IP addresses, ie. a machine with IP address *150.165.85.61* will be identified by id=*61,* and such identification is unique inside a LAN, guaranteeing that each REM will have a unique id.

Suppose the system is composed of 4 REs, one being the Master and their ids are 32, 72, 51 and 79. The Master id is the REM who started first, ie. 32; the first Slave to join the service is identified by id 72; it is also the next Master, if the first one fails. When the time comes for a Slave to become Master it will wake-up the Master process running at its machine, it will also update the configuration file that points to the Master of the system.

Replicas are submitted from MyGrid to the Master. Each replica contains information so that one can identify its owner, job, task and replica number. The Master maintains a task counter, *n*, that is incremented when replicas of a new task, ie. replicas with different owner-job-task fields, are received. These new replicas will then have an associated task number, based on the counter, and one of the Slaves will be scheduled to execute the replica. The replica with task number *n* will be assumed by the RE with id *x* obeying the following relation $x = n \% (number\ of\ REs\ in\ the\ system)$, where $a \% b$ is the remainder of the division of *a* by *b*; x is not the id of the Slave that will execute the replica, but its position in the ordered list. Supposing the ids are 32, 72, 51, 79 and x = 2; the chosen Slave is the one with id 72.

The reason that the function is based on a task counter and not a replica counter, is that only one replica of a task may enter the FINAL phase. This is a MyGrid solution to avoid more than one finished replica of a task, avoiding duplicated/corrupted results. Managing this condition is easier if every replica of a task goes to the same Slave. If that was not the case, the system would be more overloaded with messages indicating which replicas should be allowed to enter the FINAL phase.

3.2. Reliable Communication

3.2.1. Contacts from the MyGrid broker to the Master and from the Master to the Slaves

In the system three messages are sent from the MyGrid broker to the Master, they are: 1. Use Service; 2. Execute Replica; 3. Cancel Task. The first message is to be processed only by the Master, but the other two have to be forwarded by the Master to the Slaves. So a guarantee has to be made that both Slaves and Master received the message. These messages have to be processed in the order that they are issued; we cannot have a Cancel Task being processed before an Execute Replica, if the replica is of the same task. Also we cannot have any of the two last messages before the first one. To solve this problem, messages sent from the MyGrid broker to the Master obey a queue policy; the first message of the queue has to be confirmed before the following is sent. When the message is confirmed it is removed from the queue, and the next message can be sent. Message receivers ignore duplicate messages, and just confirm them again, since the problem may have occurred with the acknowledgement message.

Previously it was said that the MyGrid uses cancels job operation. This was changed to cancel task to guarantee that the queue based communication would work. If it were the contrary the master would receive a cancel job and forward it to all Slaves. MyGrid knows information on executing tasks, it is simpler for MyGrid to find out which of its tasks have to be cancelled and send only the according messages.

The Use Service message is sent only to the Master; when it is processed the Master will acknowledge the reception of message to the MyGrid broker. The other two messages have to be forwarded to the Slaves, and also confirmed by them. Figures 3 and 4 depict these messages.

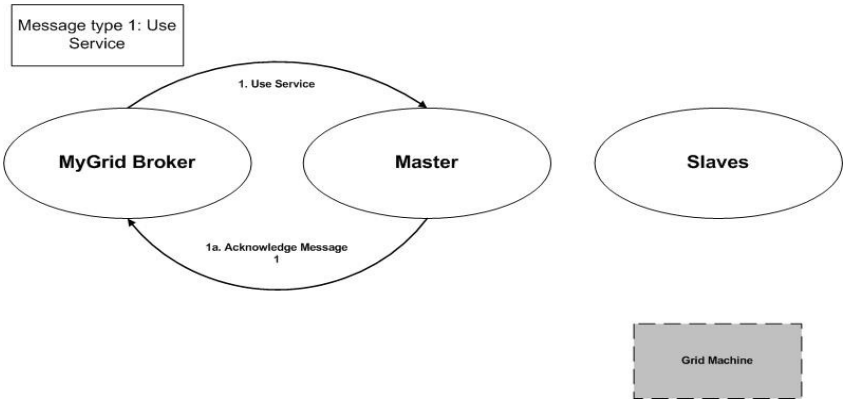

Figure 3. Use Service Message Exchange

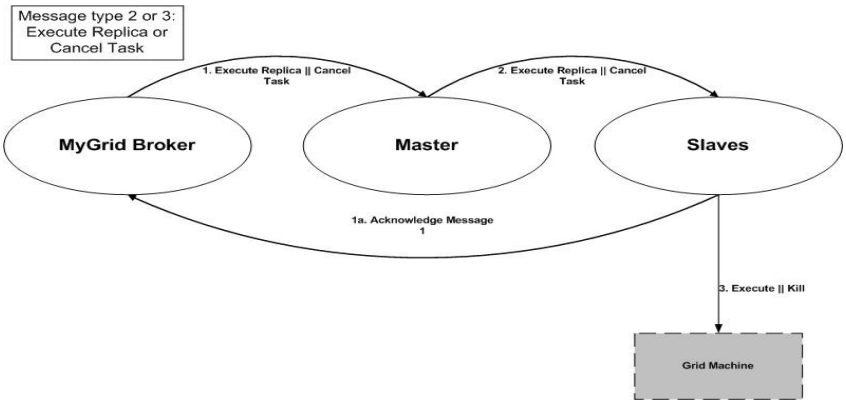

Figure 4. Replica Execution Message Exchange

3.2.2. Contacts from Slave to Master and Master to MyGrid

Two messages are sent from Slaves to the Master, they are: 1. Join Service; 2. Replica Execution Result. Unlike the previous messages these ones do no have to follow any order; they correspond to execution results which are naturally unordered (there is no way of telling which execution will end first). MyGrid already treats them as unordered.

Slaves also have to contact the Master informing their arrival in the system. Like MyGrid ´s message, this one has to be processed before any other. This is because the Slave will only be a part of the system after its join message is processed, and will have no other messages to send before this one. Since these messages do not follow any given order, they have to be identified in some way. So that it is known which message the acknowledgement is for. Messages will be identified by their replicas, since replicas are unique in the system there is no risk of acknowledging the wrong message. Figures 5 and 6 depicts these interactions.

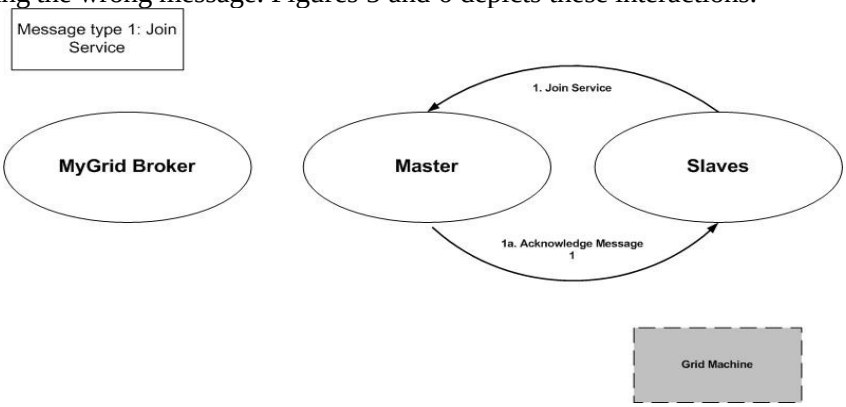

Figure 5. Join Service Message Exchange

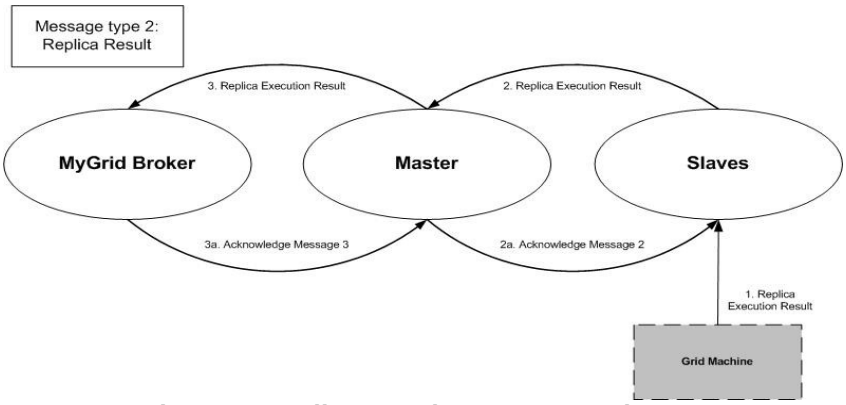

Figure 6. Replica Result Message Exchange

3.2.3. Failure Detection Service

Being composed by different machines, the RRES will need to know when instances of the service have failed. Knowing these it will be able to recover from such failures. Since the RRES is designed to run on OurGrid sites, it is assumed that it will be running on a local area network. Perfect Failure Detectors require synchronous systems over which they can be built [Larrea M. et. al., 2004]. With the limitation that the RRES will run from within a LAN, we are able to use the Perfect Failure Detector (PFD) described in [Brito A. and Brasileiro F., 2004]. This PFD runs directly from a Linux Kernel, creating a hybrid system - an asynchronous system with a small part of it that is synchronous. It is on this synchronous part that the PFD will run.

We call a Failure Detection Service (FDS) the set of failure detections modules running in distributed machines; REMs will be started only on these machines. Machines inside the FDS are identified through ids given when the PFD is loaded on such machine. As said before the system identifies REMs using the last byte of their IP addresses, these are unique in a LAN and help to guarantee that the PFD does not make any wrong suspicions.

3.3. Checkpointing

3.3.1. MyGrid checkpointing

MyGrid´s checkpoint will store a unique broker id that is a randomly generated long value, a job counter and the status of all pending jobs. The unique id is used to identify brokers in the RRES; it guarantees that if the same broker is started on a different machine. The RRES will be able to identify it. Using MyGrid´s id and the job counter, we are able to make any replica unique in the system; these are identified by the String "*mygridId.jobId.taskId.replicaId*". The checkpointing mechanism for MyGrid job status work as follows:

1. **Job submission**. When a job is submitted, the job counter is incremented and its initial state is checkpointed.
2. **Execution**. After submission, all replicas are in a READY state. When any of them has its state changed to RUNNING, i.e. when the scheduler matches a replica with a GuM, the state of the replica is checkpointed again.
3. **Final phase**. When a result for the running replica is returned, its state will change to either FINISHED, FAILED, CANCELLED or ABORTED. This new state is now checkpointed. When all replicas of the job are finished, the checkpoint information on that job is removed.

3.3.2. Master checkpointing

Checkpointing mechanisms are also included in the Master, this mechanism is composed of two

checkpoints: Replica Execution Status checkpoint and Broker results checkpoint. These checkpoints have to be stored on an NFS accessible by every REM in the system; this is necessary so that the checkpoint is available to any REM in case it has to become the Master.

The first checkpoint will contain replica execution status. This checkpoint also contains information on Slaves. This guarantees that in case of failures, replicas will not have to start over and it works as follows:

1. **New Slave wants to join the system**. The new Slave sends a remote object and the id being used by the FDS to the Master. The remote object will contain methods that the Master will use to contact it. The id and the remote object are checkpointed by the system.
2. **Execute Replica Request**. When a new execute replica request is received the state of the replica and the allocated GuM is checkpointed. A Slave will be scheduled to execute the request.
3. **Slave has sent back a result**. When the Slave sends a result back to the Master, the state of this result is checkpointed. The previous information regarding the replica execution is removed from the checkpoint. Information on this broker will be looked up and the result will be sent back to MyGrid; information on this replica execution is removed from this checkpoint. The information looked up is the remote object used to contact the broker. The id of the broker will be contained inside the replica as well as its result.
4. **Slave leaves the system**. When a Slave leaves the system for any reason, including failures, information on this Slave is removed from the checkpoint. Any pending executions are re-scheduled by the Master.

The other checkpoint will contain information on MyGrid brokers. This guarantees that in case of failures the Master will still know how to contact MyGrid brokers. It works as follows:

1. **New Broker wants to use the RRES**. When a new broker wants to use the system, it will send to the Master its id and a remote RMI object, with the methods that RRES will use to contact it. This information is checkpointed at this moment.
2. **Old broker wants to use the RRES**. If a broker has changed the machine that it was running on, it will contact the Master the same way as it did before. Its new remote object will be checkpointed with the same id. This information will overwrite the old one.
3. **A result has been received for a Broker.** When a Slave sends a result to the Master, this result is associated with the broker inside the checkpoint (according to the id of the broker) and is also checkpointed.
4. **MyGrid confirms a result**. MyGrid will now have to confirm when it has received a result. When this is done information on the result is removed from the checkpoint. There will now be no more information about the replica in the system.

3.4. Failure recovery

3.4.1. Slave Failures

Let us first explain the possible states of a replica execution: (1) The replica is known by the RRES but has not started execution on the chosen Slave; (2) The replica is executing on a Slave; (3) A replica is finished, but its result has not yet been sent to the broker; (4) The broker received the result of the replica and the replica can be removed from the system. The failure recovery will vary according to the state that the replica execution was in, these recovery actions are as follows:

1. **A Slave failed in states 1 or 2**. When the failure detector warns the system that the Slave has failed, the Master will lookup the replicas that this failed Slave was responsible

for, these will be re-schedule these to other Slaves. The Slave that is now in charge of the execution must kill the execution that the previous Slave started.

2. **A Slave failed in states 3 or 4**. Information on the result is known by the Master, since the Slave will send the result back to it. Nothing has to be done with a result in case the Slave that was executing its replica has failed; this result has to be sent back to MyGrid as usual.

3.4.2. Master Failures

Since the Master represents a bridge between contacts made by MyGrid brokers and Slaves, a failure in the Master it will cut the connection between brokers and Slaves. The system cannot wait for the Master to come back up, so that these connections are re-established. In this case a Slave will assume the Master's responsibilities. To allow this recovery, some problems need to be addressed. (1) Which Slave will become the new Master? (2) How will MyGrid brokers and other Slaves know the new Master? (3) How will the new Master continue activities like nothing had happened? (4) Who will be responsible for the replicas that the Master was executing? Solutions to these problems are:

1. **Problem 1**: Slaves know when they have to become Masters because they receive the list of ids that came before them when they join the RRES. When all of these ids have failed they know that they have to assume the Master role. The fact that ids are unique guarantees this property.
2. **Problem 2**: The new Master will update the configuration Master that pointed to the old Master, so that it now points to the new Master.
3. **Problem 3**: The new Master will read the checkpoints, and based on the information contained in the checkpoint it can wait for messages and continue scheduling activities.
4. **Problem 4**: In this case, the solution will be the same as if a Slave had failed. The new Master will act the same way but the failed id will be the id of the old Master.

4. Design and Implementation

4.1. Master Interface

The Master, being contacted by both MyGrid brokers and Slaves, must define methods that provide ten basic functionalities. These functionalities can be divided into three types: (1) methods used by MyGrid broker; (2) methods used by Slaves; (3) one wake-up method called when the sleeping master process has to assume the Master role of the RRES.

Methods of type (1) and (2) represent the messages exchanged from the MyGrid broker to the Master *(executeReplica, cancelReplicasOfTask, confirmResult and useTheService);* and from the Slaves to the Master *(replicaAborted, replicaFailed, replicaFinished, replicaCanceled and joinTheService).* The first two methods of type (1) are scheduled to a Slave, the third used to acknowledge a result, and the fourth used so that the Master can identify that a MyGrid will use the service.

The first four methods of the type (2) are called by the Slaves so that the Master can send back the corresponding result to MyGrid. The other method is used to signal that a new Slave wants to join the service. The wake-up method, named *reallyBecomeMaster*, is used by the slave process when it detects that it is time for this machine to become the Master.

4.2. Slave Interface

The Slave interface must define only execution related methods, and two acknowledgement methods, all of them are called by the Master only. Executions methods are: *executeReplica* and *cancelReplicasOfTask*. Different from the Master's versions of these methods, they actually

perform the action requested. The acknowledgement methods are *confirmResult* and *confirmMembership*. The first being used so to acknowledge that the Master has received the result and the later used to confirm that this Slave is a member of the service.

4.3. Remote Scheduler Interface

MyGrid´s Scheduler module must also have a new remote interface used to receive contacts from the Master. This interface is called *RemoteSchedulerInterface*. It defines the following methods: *replicaAborted, replicaFailed, replicaFinished, replicaCanceled, youMayUseTheService, taskCanceled and executionStarted*. The first four methods are used to receive results and are forwarded to be treated by the scheduler as it currently does.

The other three messages are used for confirmations, they unblock the communication queue so that other messages can be processed.

4.4. Reliable Communication Service

When any module of the OurGrid system sends a message to a different module, this message is identified as an event. Events are stored inside a queue and only one event will be removed from the queue to be processed inside a module at a time [Cirne et. al., 2006]. This is used so that only one thread will be executing the code at a given time.

We have not broken this property in the new RRES architecture; the architecture is also composed with events and event queues. Currently no guarantee can be made if a event is ever processed, a failure may occur while the event is still in the queue, now a possibility since the modules are in different machines. In case this happens we cannot have the module that made the contact believe that this event was processed. This guarantee has to be sent back by the module that was contacted, to guarantee that a message is processed we will have the sender of the message resending it until a confirmation is received.

To support reliable communication a simple Messaging System was developed. It is composed by *CommunicationEvents, CommunicationRequests* and *EventProcessors*. A sender module will manage communications being made using the *CommunicationEvents* and the *EventProcessor*. *CommunicationEvents* encapsulate *CommunicationRequests*. The event is used only by the processors. The receiver module will receive a *CommunicationRequest*, these will have a method called *acknowledge()* that takes care of communicating back with the sender module acknowledging the request.

Two kinds of *EventProcessor* have been developed. Both of them keep trying to communicate with the receiver until told otherwise. The difference between them is in the order which events are processed. The first obeys a queue in which an event is processed only after the one before it is removed, following a FIFO policy; this kind of event processor is called *QueueBasedCommunicationEventProcessor*. The first event will be re-processed in fixed time intervals until it is removed, then the second event will follow the same steps and so on.

In the second kind, events are independent; they are processed in no fixed order, each event being re-processed in fixed time intervals independent from any other. To support removal of events these have to be identified in some way, for this reason this kind of event processor receives an *Object* with the event, used to identify the event. This event processor is called *IDBasedCommunicationEventProcessor*. Normally, when *acknowledge()* is called the receiver takes care of removing the event from the processor.

4.5. Failure Detection Service

Being a Java Application, the RRES cannot talk directly to the PFD without the use of native code (pure Java code cannot talk directly to the Operating System). A simpler approach has been chosen to make this communication possible. A mediator is used; this mediator is a device file that is controlled by the PFD, such file is called *pfdfile*. When the file is read it will show a list containing the correct ids, in other words, ids that are alive within the service. The failure

detection can start execution by writing *RPFD (Run Perfect Failure Detector)* to the file; it can be stopped by writing *SPFD (Stop Perfect Failure Detector)*.

A Java Library was developed, it is able to communicate with the Failure Detector using the file; it is called PFDOracle (Perfect Failure Detection Oracle). With it a user can inform a set of machines to monitor and will be notified of failures when such machines fail. It is also able to start and stop the Failure Detector by writing to the file. The PFDOracle has a basic service interface called *PFDOracleServices* that defines the basic methods for working with the PFD. It also has another interface called *PFDOracleListener*. Implementers of this interface are notified when an id has failed through the method *idHasFailed*.

4.6. Checkpointing

Checkpoints are based on Java Serialization. Serialization allows a user to store Java Objects to a file and recover such objects, all a user has to do is make the Class that represents the object implement the *Serializable* interface. With checkpoints the Master and MyGrid will have the capability of performing *crash-recovery*. Java checkpointing mechanisms like the ones described in [Silva et. al., 2002] and [Lawall and Muller, 1999], store the state of an object in fixed time intervals or whenever a certain method is called. We have developed journal based checkpoints to have more control over the failure-recovery. When recovering failures the journal will have information on what was happening with the system before failure and not only a set of persistent objects.

Some parts of the RRES will need to be constantly performing checkpoint, supposing that MyGrid sends a task execution to the Master. If this information is not checkpointed before acknowledging MyGrid, the execution information can be lost forever if the Master fails. For this reason we do not use timely checkpoints. The checkpoint system developed stores information in real time, synchronized with the file-system, i.e. as soon as some information is received it will be stored in the checkpoint. This kind of checkpointing was also developed to decrease checkpointing time; it will probably increase recovery time, since the whole journal will have to be read and interpreted.

This incremental checkpointed will probably become very large with time; to limit the size of the journal a second journal is kept within the checkpointer's memory. The memory journal is update on the same time as the file one, but the memory journal will trim old entries according to the status of new ones. So, supposing that a Job has finished, any entry related to that Job can be removed from the journal; to do this constantly on disk can be quite an overhead. When the file journal reaches a maximum size limit it will be replaced with the one in memory. At the time of replacement the checkpointer will freeze the checkpoints by preventing current threads to use it. When the replacement is done and the checkpoint is completed, they are unfrozen. This freeze operation is implemented by blocking any Java Threads accessing the checkpoint.

There are three main entities involved with checkpointing. The two interfaces *JournalWriter* and *JournalReader*, and the class *JournalEntry*. Implementations of the first two are used to write and read the checkpoint. In the RRES we use a *FreezeBasedJournalWriter* that adds the freezing functionalities. The *JournalEntry* contains the information that will actually compose the checkpoint.

5. Conclusions

In conclusion we can list the main characteristics of the RRES:

1. **Adaptability –** New Slaves can join the system and no new configuration is needed so that executions can be scheduled to these Slaves. Also the system can easily adapt to REM failures.
2. **Dynamism –** New execution requests sent by MyGrid are scheduled to Slaves on

demand. Results are also sent back to MyGrid as soon as they are available. Only in the case that MyGrid fails are these characteristics broken. Nevertheless, they are valid again, as soon as MyGrid recovers.

3. **Reliability –** Communication is reliable inside the system, also, faults are tolerated. Using both reliable communication and fault recovery, failures do no affect the system and are unnoticed by both MyGrid and REMs.
4. **Scalability –** REMs do not perform large computations, their job is only to manage executions. Since this management is distributed amongst REMs, a system with demand proportional with the number of REMs will not have any scalability issue.

The system brings fault-tolerance to MyGrid. It also makes its architecture more service-oriented. We expect that the functionalities performed by MyGrid will improve in execution-time, now that the replica execution overhead has been removed from it. The next step will be to experiment with the RRES and verify these assumptions, and also to verify the cost of fault-tolerance in a MyGrid.

A prototype of the system has been developed, experiments will be made using this prototype so that the REES may become a part of OurGrid's production releases.

References

Brito A. and Brasileiro F. (2004) “Programando um Subsistema Síncrono para Suporte a Mecanismos Eficientes de Tolerância a Falhas”. Workshop de Tolerância a Falhas / Simpósio Brasileiro de Redes de Computadores

Cirne W., Paranhos D., Costa L., Santos-Neto E., Brasileiro F., Sauvé J., Silva F. A. B., Barros C. O. and Silveira C. (2003) “Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach”, Proceedings of the ICCP'2003 - International Conference on Parallel Processing

Cirne W., Brasileiro F., Andrade. N. A., Costa L., Andrade A., Novaes R. and Mowbray M. (2006) “Labs of the World, Unite!!!”, Journal of Grid Computing

Foster I., Kesselman C., Tuecke S. (2001) “The Anatomy of the Grid: Enabling Scalable Virtual Organizations.” International J. Supercomputer Applications

Foster I., Iamnitchi A. (2003) “On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing” Lecture Notes in computer science, Springer

Goux J., Linderoth J., and Yoder M. (1999) “Metacomputing and the Master-Worker Paradigm”

Goux J., Kulkani S., Linderoth J., and Yoder M. (2000) “An enabling framework for master-worker applications on the computational grid.” Submitted to HPDC 2000 Conference Proceedings

Larrea M., Fernández A. and Arévalo S. (2001) “On the Impossibility of Implementing Perpetual Failure Detectors in Partially Synchronous Systems.” Brief Announcements 15th Int´l Symp. Distributed Computing

Lawall J. L. and Muller G. (1999) “Efficient Incremental Checkpointing of Java Programs.” Proceedings of the International Conference on Dependable Systems and Networks

OurGrid Team. (2006) “OurGrid Website and Documentation”. http://www.ourgrid.org

Silva H. and Chiao C. M. (2004) “Obtenção de Tolerância a Falhas na Ferramenta de Computação MyGrid”. Escola Regional de Redes de Computadores

Silva F. A., Jansch-Pôrto I. and Lisboa M. L. (2002) “Recuperação com base em checkpointing: Uma abordagem orientada a objetos”. Workshop de Tolerância a Falhas / SBRC