Hamster: An AOP solution for Fault Tolerance in grid middleware

Francisco Soares-Neto and Rafael Farias
Escola Politécnica de Pernambuco
University of Pernambuco
Recife, Pernambuco
Email: {xfrancisco.soares, rafael.lucas}@gmail.com

João Paulo Borges and Fernando Castor
Informatics Center
Federal University of Pernambuco
Recife, Pernambuco
Email: {jpab, castor}@cin.ufpe.br

Abstract—Grid computing is useful in several organizations, from research endeavors in parallelizing algorithms, to serving as a backbone to cloud computing platforms. However, grid computing infrastructures are still liable to faults which, depending on the application being executed, might have serious consequences. In this paper we present Hamster, a software architecture that attempts to maximize resource usage by monitoring grid middleware components and making them capable of recovering from failures. We present the aspect-oriented implementation of Hamster. It extends the OurGrid grid middleware, making it capable to recover from a number of failure scenarios that would otherwise halt the execution of an entire grid. In addition, our implementation of Hamster is pluggable and reusable; we integrated it with four different versions of OurGrid without modifying the latter. We evaluate memory and CPU consumption of the solution using both load-time-weaving (LTW) and source-code-weaving (SCW). The SCW solution imposes an acceptable memory and CPU consumption overhead, differently from LTW that shows a massive use of both.

I. INTRODUCTION

In developing economies, universities and companies often do not have financial resources to acquire super computers or powerful servers to meet their computing needs. Notwithstanding, these organizations usually have a great amount of average workstations that sit idle for large periods of time. By leveraging grid computing [1], these organizations can solve tasks that would require a large amount of processing at a low cost. A computational grid consists of a group of geographically distributed computing resources interconnected by a wide-area network and a middleware platform capable of managing heterogeneous and distributed resources. Grid users may submit applications which will be executed in one of these distributed resources. Unfortunately, computational grids are prone to faults, which can lead to losing hours or days of work.

There are many proposals [2] [3] [4] [5] to decrease the impact of faults in grid computing environments. These approaches focus on grid resource providers, nodes that run the grid applications. In general, they use some kind of checkpointing mechanism that saves and recovers the state of each node as required. These solutions are not, however, designed for handling failures of the nodes responsible for the grid middleware, the nodes that maintain the grid infrastructure. Faults of these nodes might require a full restart of the grid, which might be a very expensive operation. First, from an administrative perspective, because manual recovery potentially spans hundreds of nodes deployed across different administrative domains. It is also expensive in terms of computing resources since it can waste hours or even days of work processing. Finally, it can be expensive in terms of user time, as users might have to resubmit their jobs and wait for them to finish.

This paper presents Hamster, a software architecture for the construction of fault-tolerant grid middleware. Its main goal is to maximize the use of available resources, monitoring and recovering faulty middleware components and avoiding resource idleness or, worse, a complete grid stop. Hamster comprises a number of services that complement existing application-level checkpointing mechanisms. It has been realized as an extension to the OurGrid middleware platform [6] but it is compatible with other grid middleware infrastructures. To achieve the objective of reusing Hamster with different versions of OurGrid with the minimal effort we tried load-time-weaving, but during evaluation (Section VI) it has shown a massive CPU and memory consumption. On the other hand source-code-weaving presented an acceptable overhead.

In this paper, we present Hamster’s requirements and main components, its implementation using aspect-oriented programming, source code weaving and load-time-weaving, and an innovative study of performance, memory and CPU consumption, of both implementation. We also compare the fault models of OurGrid and our extended version and explain why our implementation of the Hamster architecture is reusable. The latter has been incorporated into different versions of OurGrid without requiring any modification to the latter.

II. GRID COMPUTING FUNDAMENTALS

A computational grid consists of a group of geographically distributed computing resources interconnected by a wide-area network and a middleware platform capable of managing heterogeneous and distributed resources. Grid users may submit applications which will be executed in one of these distributed resources. An application may be a simple process or parallel, computationally complex jobs.

Grids can be divided in two types: dedicated grids, in which some machines are always resource providers; and
opportunistic grids, in which any machine can execute tasks, as long as they’re running the required components for resource providing.

The main components of a grid middleware platform, as shown in figure 1, are the following. The access agent or application broker component allows users to submit their applications to the grid. It enables specific restrictions for task execution and must be executed in all machines which will submit tasks. The scheduling service receives requests, verifies users, and uses a monitoring service to verify which providers may execute the application - to which it forwards the tasks. A number of resource providing nodes, when idle, might receive tasks from a scheduling service for execution. In opportunistic grids, faults on resource providers are frequent, since the resource providing service is interrupted whenever a user requires the machine resources. When a resource provider finishes executing a task, it returns its result to the application broker. The security service protects resource providers by limiting system permissions to the grid applications.

Several grid middleware platforms are available for public use, such as OurGrid [6], InteGrade [8] and Globus [9].

III. REQUIREMENTS FOR A FAULT-TOLERANT GRID COMPUTING ARCHITECTURE

The design of an architecture such as the one we propose poses many challenges. In this section, we list a number of requirements for the construction of a fault tolerance mechanism for grid middleware. These are the main drivers of the Hamster architecture and must be met by implementations of the latter. Hence, hereafter, for simplicity, we call Hamster both the (conceptual) software architecture and its potential implementations. We have gathered these requirements based on our experience in working with two different grid middleware platforms, InteGrade and OurGrid, and also from the literature [10].

First, Hamster must make the grid highly available. It might make some compromises, as long as it avoids wasting resources while maintaining scalability and performance. This is achieved considering every monitored node as an independent node, and that a failure of any of the components should not affect the normal functioning of any other node. That is necessary to avoid tasks from not being executed, machines from remaining idle, and time being wasted by a user having to manually resubmit a task. Hamster’s philosophy is that the grid must always be busy, as long as there are tasks to be executed. It should also present high availability even when multiple grid components fail. Also, it has to monitor itself and be capable of recovering its own components when they fail. We thereby consider both the Hamster nodes and the grid components to fail according to a crash-recover model by which a number of nodes can fail concurrently. Node failures can also be detected using adaptive timeout-based failure detectors [11].

Hamster should also be scalable, thus being capable of dealing with very large grids, comprising potentially thousands of nodes running in parallel. Moreover, machines may be geographically distributed, across countries and continents. Hence, message exchange must be efficient and large amounts of file transfers in a short timespan must be avoided.

It is important that Hamster implementations are lightweight, not requiring too much memory, CPU power, or network bandwidth. Hamster execution should not affect the performance of a machine if a user requires it to perform tasks locally. Moreover, the framework also has to respond quickly to failures, because if it takes too long, resources can be wasted.

Another important requirement is that it should be easy to configure, execute, and integrate with existing grid middleware platforms and its coming versions. This requirement is difficult to meet but is crucial for the reliability of the service. Hamster has to be compatible with different architectures for grid computing middleware with minimum changes. It should also be adaptable enough to consider differences in the recovery procedures required by different grid platforms

IV. THE HAMSTER ARCHITECTURE

Hamster is a grid middleware monitoring and recovery architecture that aims to avoid resource idleness that stems from failures of grid middleware components. Since the failure of such components can stop the whole grid, this event should be avoided at all costs. Hamster attempts to achieve this by monitoring grid components and, when they fail, recovering from the failure in the way that is less damaging to the grid as a whole.

The monitoring that Hamster performs focuses on the components that manage the computational grid. When such a component fails, Hamster acts by first trying to micro-reboot the component and, if this is not possible, restarting it while trying to recover the component’s state just before the failure. When recovery is not possible in the former environment of the failed component, it is necessary to transfer the faulty component to another machine where it can run without errors. When this component migration is necessary, Hamster automatically reconfigures the grid, so that other grid components that need to know about this change are consistently updated about it.
Components of the Hamster architecture work alongside the components of a computational grid. Each one of the latter is monitored by processes which execute in parallel on the same machine where the component is in execution. Hamster components are identified by the IP address of the machine where they are executed. Moreover, each Hamster component monitor is part of a user-defined group. Groups comprise identifiers of machines located within the same physical location. This information is used to decide about the machine where a component should be recovered. That is meant to avoid the recovery of a faulty component in a machine that is inaccessible to the application submitter. This kind of measure is important because grids can span different administrative domains, potentially different continents. For example, an application broker recovered in a different lab, to which its original user has no access, would stop this user from getting the results of the submitted job.

Hamster relies on the idea of information replication; therefore, all the information necessary to the operation of the grid components, as well as information on their current status, is replicated in a parallel grid, composed of the Hamster components (we call Hamster node each node where Hamster component is in execution). That information is stored on different Hamster nodes and, in case the component fails, it is possible to recover the current state at the moment of the failure. Recovery may take place either on the same machine or, if the latter is unavailable, on a different one.

V. OURHAMSTER: HAMSTER MEETS OURGRID

OurGrid [6] is a well-known grid computing middleware platform written in Java. It comprises four components: the PEER component is the scheduling service, managing the global grid resources, and distributing work among the resource providers; the BROKER component is the application broker, responsible for submitting user jobs and displaying their results back to the users who requested them; the WORKER component is the resource provider, responsible for executing the user application that was submitted by a BROKER and directed by a PEER, i.e., each resource providing machine executes a WORKER component; and the OPENFIRE component is the responsible for grid user management and communication, serving as a communication infrastructure. Each grid node might run up to one instance of each component.

Tests with failure injection in OurGrid have shown us the need for a service such as the one provided by Hamster. Both failures of some grid components and wrong suspicions in failure detection resulted in both partial and complete failures of OurGrid-based grids (Section VI). To alleviate these problems, we have developed a modular implementation of the Hamster architecture and combined it with OurGrid to produce the OurHamster reliable grid middleware. Initially, we developed OurHamster by modifying the implementation of OurGrid, introducing code in the latter to perform tasks such as sending heartbeats, monitoring changes in the grid components and saving the application states of these components. However, this approach would prevent us from reusing the Hamster implementation on new releases of OurGrid. Therefore, we have adopted a different approach where Hamster components only interact with OurGrid by means of aspect-oriented programming [12] constructs using the AspectJ language [13].

The use of aspects resulted in an implementation of Hamster that is reusable and pluggable. It relies on two conditions to be kept up-to-date with OurGrid’s evolution: (i) OurGrid’s architecture, which has been stable for the last two years; and (ii) method naming conventions. Moreover, for the latter case, adapting Hamster to changes would still be easier than having to go through all the OurGrid codebase to perform changes.

OurHamster uses four aspects to connect OurGrid with the Hamster service. Each aspect is related to the functionality of one of the components of OurGrid. The aspects are responsible for either retrieving data or detecting component changes that would affect the recovery processes of the system, and communicating that information through Hamster’s Communication module. Later, Hamster stores OurGrid information and disseminates it to other Hamster nodes over the grid. The storage of such information, as explained previously, is part of Hamster’s recovery processes.

The Broker (fig. 2) aspect is responsible for sending information about the initialization of a new Broker to the local Hamster node. The Worker (fig. 3) aspect does the same for a worker component. The Peer (fig. 4) aspect is associated not only with a Peer’s initialization, but also with any changes in its configuration list, such as a worker added or removed from the worker list. It sends that information to the local Hamster component, where this data will be disseminated over the system. Finally, the Common (fig. 5) aspect is associated with the common aspects of all components in the architecture.

```java
public aspect Broker {
  pointcut startBroker() : call(* waitForBroker*(..));

  before() : startBroker() {
    GridMonitorSpy gms = new GridMonitorSpy
    (TipoMonitorado.BROKER, 9000);
    gms.start();
  }

  after() : startBroker() {
    BrokerLocalOperation.saveBrokerInformation();
  }
}
```

Fig. 2. Broker aspect

```java
public aspect Worker {
  pointcut startWorker() : call(* waitForWorker*(..));

  before() : startWorker() {
    GridMonitorSpy gms = new GridMonitorSpy
    (TipoMonitorado.WORKER, 9000);
    gms.start();
  }

  after() : startWorker() {
    WorkerLocalOperation.saveWorkerInformation();
  }
}
```

Fig. 3. Worker aspect
public aspect Peer {
    pointcut startPeer() : call(* waitForPeer*(..));

    pointcut removeWorker(WorkerSpec workerSpec) : call(public void removeWorker(WorkerSpec)) && args(workerSpec);

    pointcut definePeerUserInfo(Collection<UserInfo> data) : execution(public void setTableModelData(Collection<UserInfo>) && args(data));

    pointcut defineWorkerUserInfo(Collection<UserInfo> data) : execution(public void setTableModelData(Collection<UserInfo>) && args(data)) && within(WorkerTablePanel);
}

Fig. 4. Peer aspect

/** Aspects common to all components. */

public aspect Common {
    pointcut startComponent() : execution(public void actionPerformed(..))
        && within(StartBrokerAction) || within(StartPeerAction) || within(StartWorkerAction));

    pointcut getComponentInformation() : execution(public ModuleContext createContext() && eflow(StartComponent()));
}

Fig. 5. Common aspect

of OurGrid. It sends general component information (server name, login and password) to Hamster nodes after a component is initialized.

Overall, the interface between OurGrid and Hamster is narrow, requiring only eight pointcuts, most of them located at the Peer aspect. This narrow interface suggests that our aspect-oriented implementation is not fragile and that only very specific or very systemwide modifications would have impact on Hamster. Furthermore, since OurHamster’s aspect-oriented version avoids changes inside the code, it is easier to migrate from one OurGrid version to another. We have found it suitable for work across multiple versions. We have introduced the proposed extensions in OurGrid in two manners: (i) by weaving the aspects into the JAR file of OurGrid, using load-time weaving; and (ii) by using source code weaving. The latter approach requires the source code of OurGrid, which is freely available, but does not modify it in any way. We discuss this matter further in Section VI.

VI. Evaluation

Evaluation has been performed through comparisons of a few development versions amongst themselves and against the original OurGrid implementation. As development first was made based on a defined version of OurGrid and later on refactored to AOP, we could only compare the original development version against a single version of OurGrid, namely it’s version 4.2.0. AOP version, on the other hand, could be executed against several OurGrid builds, from version 4.1.5 to its earliest version, 4.2.3, since they were based on OurGrid’s current 3-component architecture, on which OurHamster was based. These versions were tested using load-time weaving to merge the aspects with OurGrid code already compiled and packed into library files - with JAR extension, since OurGrid executes as a Java application.

The evaluation was executed on three virtual machines. Each machine was configured with 768MB of RAM, and Ubuntu 10.04 as their operating system. The host for all machines ran on a 2.6Ghz quad-core processor, with 4Gb of RAM and Ubuntu 10.04 as it’s OS. The virtualization platform chosen was VirtualBox, for it’s public availability and ease of configuration for the virtual machines. Another machine, with a 2.4 Ghz core 2 duo processor and 4Gb of RAM was used for the execution of the communication platform, the XMPP server OpenFire, but wasn’t taken into account in our measurements.

For each version, tests were executed five times. In every turn executed, each component was initialized separately on a different virtual machine, with their resource consumption behavior being monitored through the command-line command “top -d 10”. At every 10 seconds, a line description of resource consumption was recorded, which were later analyzed. Monitoring started right from the component startup, even before it’s initialization in the grid platform; it passed by the submission of a job by the Broker component, the distribution of the job by the Peer component, and it’s execution by the Worker component; monitoring ended after all the components were finalized - through a “Stop” command on the middleware.

A. OurGrid vs. OurHamster

In our tests, we have found an increase in average CPU consumption, as shown in Figure 6, depending on the observed component. For the Broker component, there was no special difference in the average CPU usage. In some cases, CPU consumption was even lower for OurHamster (OH4.2.0). This result may be a consequence of the small number of executions observed. It can also stem from some difference between the code stored in OurGrid’s (OG4.2.0) repository at the time it was checked out and the one used for the tested production build. The aspect-oriented version of OurHamster (AOH) also had slightly smaller CPU consumption, on the average (albeit still greater than the purely object-oriented version). Notwithstanding, the difference was small enough that it can be attributed to punctual fluctuations in resource consumption.

At last, the CPU consumption of the Worker was greater for OurHamster (both versions) than for OurGrid. However, it was proportionally lower than the growth in CPU usage of the
Fig. 6. CPU consumption comparison between OurGrid, OurHamster and aspectized OurHamster

Fig. 7. Memory consumption comparison between OurGrid, OurHamster and aspectized OurHamster

Peer implementations. Moreover, the overall CPU usage was low, with the AOH version of OurHamster consuming slightly more than 0.6% of the CPU. These results provide evidence that Hamster is, in fact, lightweight and satisfies some of the requirements we have discussed in Section III.

Memory consumption has not increased in the same proportion as CPU consumption, as shown in Figure 7. The only component of OurHamster to exhibit an increase in memory consumption greater than 10%, when compared to OurGrid, was the aspect-oriented version of the Broker component. Even then, the increase was not much greater than that. Moreover, in our tests, the aspect-oriented version of the Worker component actually had a lower memory consumption than its counterparts. We consider all of them to balance out in memory consumption, since the decrease may have been caused by the reduced number of executions, and could have been phased out with more sampling. Still, that indicates a relation between memory consumption and the support of Hamster to each component, with the Worker being the least supported, as expected. Analogously to CPU consumption, it seems reasonable to say that the overhead that Hamster imposes in terms of memory consumption is also low.

B. Aspectized OurHamster with Load Time Weaving

As a proof of concept, execution of the Aspectized OurHamster was also done with other versions of OurGrid, ranging from 4.1.5 to 4.2.3. The integration of our defined aspects with those versions proved itself perfect. To test that integration, we used an AspectJ feature called "Load-Time Weaving" (LTW), where AspectJ’s aspects were only weaved when a class is loaded and defined to the JVM. Use of the LTW feature of the AspectJ compiler allowed us much greater flexibility to change between versions, which proved good for testing purposes. However, it implied a grave cost to CPU and Memory usage (see figures 8 and 9).

To enable LTW, it is necessary to use a "Weaving Class Loader" at run time. That explains the additional cost for both CPU and Memory. Most of our executions showed huge increase in CPU usage, which would not be feasible in a production environment. The Worker component was the only exception to the rule, since Hamster affects Worker execution only marginally. Memory usage at least doubled, and, although memory has become cheaper over the years, we still would not advise that solution for a real world application. Overall, we consider the LTW, in our case, ideal for new feature testing purposes, but not appropriate for final releases. In the end, for production, we would use compile-time source code weaving. For OurGrid, that is not a limitation, since its source code is publicly available.

VII. RELATED WORK

There are several related studies on the implementation of fault-tolerance. Various solutions have been proposed for the creation of rollback-based checkpoint mechanisms in opportunistic grids [3]. One of these strategies proposes the use of checkpoint replication, by which, on the event of failure of a checkpoint-storing machine, others can still retrieve it and recover the system state. Another strategy consists of dividing checkpoint data into multiple fragments with parity information. A single copy of the checkpoint would be saved, but split into several grid nodes, which avoids the need for large file transfers, but increases the overhead on checkpoint creation and restoration. These techniques would allow the
recovery of some of the lost processing, but do not cover middleware failures. If the communication substrate of the grid fails, a checkpointing mechanism does not apply. Differently, GridTS [14] replaces scheduling services with a single shared memory object, a Tuple Space, and resource providers chose tasks to execute. This involves a combination of transactions, checkpoints and replication of the tuple space to avoid a single-point of failure.

AOP has been used before on OurGrid with success, refactoring OurGrid’s own architecture to provide better maintainability to its codebase [15]. This involved ad-hoc finding of useful aspects from within OurGrid’s implementation to better structure the project. Others have worked with the use of aspects to enhance OurGrid’s communication architecture to improve testability, through the use of a failure-injector [16].

AspectC++ has been used for implementing fault-tolerance on a system, with good results in terms of overhead [17]. By optimizing the weaver, the generated fault-tolerant code behaved even better than a manual C implementation, potentially reducing design and maintenance costs. Our work diverges by using AspectJ to add fault-tolerance on OurGrid, in which Hamster was comparatively a small part, as well as analyzing the costs of load-time weaving.

VIII. CONCLUDING REMARKS

This paper presented Hamster, a mechanism to make grid middleware fault-tolerant. We hope, with this mechanism, to improve the reliability of grid computing systems and maximize their resource usage. We have shown that an implementation of the Hamster architecture, OurHamster, satisfies many of the requirements we have laid out: it has low memory and CPU footprint, when used with source-code-weaving, it is easy to use, with relatively few configuration parameters, easy to integrate with an existing middleware platform, and it promotes availability, keeping the grid functioning in a number of situations where it normally would not. As a side note, we are currently unaware of any other work which has analyzed, even superficially, the performance of the load-time weaving mechanism of AspectJ. Any application that can benefit from grid computing, such as weather forecast, financial market prediction, and DNA analysis, can leverage Hamster. Hamster is under development and its current prototype is available at http://code.google.com/p/aspect-hamster/

IX. ACKNOWLEDGEMENTS

We would like to thank the anonymous referees, who helped to improve this paper. João is supported by CNPq/Brazil (503426/2010-5). Fernando is partially supported by CNPq (308383/2008-7 and 475157/2010-9) and FACEPE (APQ-0395-1.03/10). This work is partially supported by INES (CNPq 573964/2008-4 and FACEPE APQ-1037-1.03/08).

REFERENCES


