Distributed Replica-Exchange Simulations on Production Environments using SAGA and Migol

André Luckow[1], Shantenu Jha[2,3,4], Joohyun Kim[2], Andre Merzky[2] and Bettina Schnor[1]

[1]*Institute of Computer Science, Potsdam University, Germany*

[2]*Center for Computation & Technology, Louisiana State University, USA*

[3]*Department of Computer Science, Louisiana State University, USA*

[4]*e-Science Institute, Edinburgh, UK*

Abstract

There exists a class of scientific applications for which utilizing distributed resources is critical for reducing the time-to-solution. However, the ability to orchestrate many distributed jobs in a dynamic and inherently unreliable distributed environments is a major challenge. The more resources and components involved, the more complicated and error-prone the system becomes. We discuss a specific class of applications – Replica-Exchange simulations – where utilizing as many (often heterogenous) distributed resources as possible, is critical for the effective solution of the scientific problem. Such applications require effective mechanisms to handle the unreliability inherent in dynamic distributed systems. In this paper, we describe the design, development and deployment of a unique framework for constructing fault-tolerant distributed simulations. The framework consists of two primary components – SAGA and Migol, is scalable, general purpose and extensible. SAGA is a high-level programmatic abstraction layer that provides a standardised interface for the primary distributed functionality required for application development. We provide details of a newly developed functionality in SAGA – the Checkpoint and Recovery API. Migol is an adaptive Grid middleware, which addresses the fault tolerance of Grid applications and services by providing the capability to recover applications from checkpoint files transparently. In addition to describing the integration of SAGA-CPR with the Migol infrastructure, we outline our experiences with running a large scale, general-purpose, SAGA-CPR based Replica-Exchange application in a production distributed environment.

I. Introduction

There exist several types of applications, which are well suited to distributed environments. Probably the best known and most powerful example are those that involve an ensemble of decoupled tasks, which we refer to as *pleasingly-distributed* applications; in spite of its conceptual simplicity, many scientific problems based on parameter sweeps and/or Monte Carlo simulations, can be solved using infrastructure that supports this common application class. A slightly more complicated and challenging class of distributed applications are those that have a small level of coupling between individual sub-tasks. An interesting example of such applications are those based on *Replica-Exchange (RE)* [1], [2] simulations. Such applications can be used to understand important physical phenomena – ranging from protein folding dynamics to binding affinity calculations required for computational drug discovery.

Distributed RE simulations must be able to orchestrate different heterogeneous resources in a complex and dynamic environment. Writing such applications is a complex task for a myriad number of reasons, not least of which is that distributed computing environments are inherently prone to failures and thus unreliable [3], [4], [5]. Some applications can respond to such failures via redundant or speculative computing. However, redundant computing has its limitations, especially when there is a level of heterogeneity and coupling between tasks. Speculative computing is still possible, but its use to mitigate the consequence of distributed failures, leads to a whole host of load-balancing and scheduling problems.

Partly due to some of these challenges, even though distributed RE simulations are loosely-coupled, failures can become a major problem for such applications. In RE simulations, there is a need to occasionally attempt an exchange between pairs of replicas; the pairing of replicas is not a constant but is dynamically evolving. However, once a pair of replicas has been established, a delay or loss of one replica will stall the other replica. Thus over time, due to the fact that in principle, every replica will ultimately interact/exchange with another replica, a single failure, if left uncorrected can ultimately cause the simulation to encounter an exponentially increasing slow-down. In the worst case, a single failing task can render the entire computation worthless. Thus, it is essential to provide support for fault tolerance at some level. *Migol* [6] provides fault-tolerance at the middleware level by supporting the transparent starting, monitoring and recovery of tasks.

Having motivated the need for a fault-tolerant framework for application development, in particular for distributed RE simulations, this paper describes the design of the SAGA [7] Checkpoint & Recovery package (CPR), and its implementation using the Migol adaptor. SAGA-CPR [8] represents the first extension to the core SAGA API specification, and is thus a validation of the extensibility of the specification. Additionally Migol can be regarded as a reference implementation of the GridCPR architecture [9]. We demonstrate this by presenting the seamless integration of CPR concepts and abstractions into the SAGA framework; the resulting system is able to provide an easy to use, high-level programming abstractions for Grid enabled, fault tolerant applications. The pairing of Migol and SAGA-CPR thus provides a natural, portable and a very generic solution to the problem of a programmatic interface for designing fault-tolerant applications. To highlight this and that it is critical that all aspects of the distributed application life-cycle – design, development and deployment – are made easier, we will discuss our experiences in developing and deploying a RE application in a production environment.

Before outlining the structure of the paper, we highlight the main advantages of our approach: First and foremost it is a general purpose framework that can be used over a wide-range of distributed production environments, such as the TeraGrid, UK's NGS, DEISA, etc., as opposed to WISDOM [10], that is essentially confined to gLITE/EGEE. Secondly, our approach can scale to use resources of different sizes, as opposed to *Folding@home* [11] which is based upon BOINC, and is thus inherently limited in the size of physical problems that can be solved effectively. Thirdly, our framework is extensible: it can be used to implement many other application in addition to those based upon RE simulations [12]. The power to do so arises from simple design decisions: the use of standard interfaces on the one hand, and the use of appropriate programmatic and system abstractions that allow users to do what they can do best (i.e. provide the simulation and orchestration logic), whilst ensuring that middleware used provides required services (such as checkpoint management, application monitoring and recovery) seamlessly and effectively from the application developers perspective.

The remainder of the paper is structured as follows: In the next section we provide the basic ideas behind RE and specifically Replica-Exchange using Molecular Dynamics simulations. We then discuss the fault-tolerant Migol framework. In section 4 we present the SAGA-CPR package and show its relation to Migol. In section 5, we discuss how SAGA-CPR and Migol are used to implement a fault-tolerant, distributed framework for RE. In the subsequent sections we discuss our experience in implementing REMD and performance figures. We conclude by providing a detailed analysis of related work, which will highlight the truly unique features of our implementation.

II. Replica-Exchange Molecular Dynamics

In Molecular Dynamics (MD) approaches, a sufficient sampling of configurations is an important requirement for connecting atomistic results to macroscopic or thermodynamic quantities available from experiments. However, even with the most powerful computing resources at the moment, straight-forward MD simulations are unable to reach the relevant time-scales required to study conformational changes and searches. This is part due to the inherent limitations in the MD algorithm – a global synchronization is required at the end of each time step. This provides an important motivation for research into finding ways to accelerate sampling and enhance "effective" time-scales studied. Generalized ensemble approaches – of which Replica-Exchange Molecular Dynamics (REMD) [2] are a prominent example – represent an important and promising attempt to overcome the general limitations of insufficient time-scales, as well as specific limitations of inadequate conformational sampling arising from kinetic trappings. The fact that one single long-running simulation can be substituted for an ensemble of shorter-running simulations, make these ideal candidates for distributed environments.

Replica-Exchange (RE) simulations can be thought of as consisting of two distinct components: the underlying simulation engine/mechanism used for each replica, and the coupling-mechanism between the individual replicas. It is important to note that RE is in fact a class of algorithms and not a specific algorithm [13]; for example, there can be not only different simulation strategies – such as the Monte Carlo and the described MD approach – but also multiple levels-of-coupling.

The degree and frequency of coupling and exchange can be either regular [1], [2], or irregular [14], [15]. An example of the latter – parallel replica dynamics as implemented in Folding@home, involves coordination between replicas only when an "event" occurs. In contrast, for regular RE applications, attempts to exchange states between certain pairs occur at fixed intervals. A major challenge common to both types however, is the design, development and deployment of a general purpose RE framework for distributed environments.

III. Migol: A Fault-Tolerant Service Framework

Migol guarantees the correct and reliable execution of applications or tasks even in the presence of failures. The framework is based on the Globus Toolkit 4. Figure 1 shows the current Migol architecture and the interactions between the different services.

The fundamental metadata model of Migol is the *Grid Service Object (GSO)* schema, which defines a generic and extensible information model for describing Grid applications. A GSO stores all relevant information about an

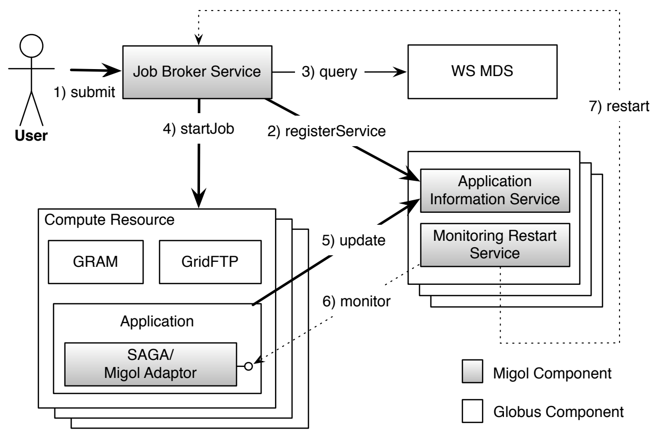

Fig. 1: **Migol Architecture: Migol provides services for supporting the fault tolerance of Grid applications. Applications that are managed by Migol are transparently monitored and recovered in case of failures.**

application: resource requirements, the location of binaries and checkpoint files, global unique identifier (GUID), etc.

Grid Service Objects containing the metadata of all running applications are stored in the *Application Information Service (AIS)*. To avoid a single point of failure, the AIS is replicated using a ring-based replication protocol, which ensures the data consistency (see [16] for details).

Applications are started via the *Job Broker Service (JBS)* (step 1, Figure 1). Before job submission, the JBS must register the GSO of the application at the AIS (step 2). Resource discovery is performed through WS MDS [17] (step 3), which aggregates data of different services, e. g. the Network Weather Service (NWS) [18]. Available resources are matched by the JBS according to the application requirements. For execution of the application on Grid resources, the JBS relies on a custom module, the Advance Reservation Service, which is also capable of supporting resource reservation on top of GRAM.

Migol provides several mechanisms for supporting the fault tolerance of distributed applications. To detect failures, the *Monitoring and Restart Service (MRS)* periodically monitors all services registered at the AIS (step 6). In case the MRS discovers an inactive application, it initiates a restart respectively a migration using the JBS (step 7).

For recovery, Migol relies on application-level checkpointing, i. e. applications have to be written to accommodate checkpointing and restart. The checkpoint metadata maintained by the AIS must be updated by the application each time a new checkpoint is witten (step 5).

Migol can be considered as GridCPR reference implementation for which the SAGA-CPR package provides a well-defined, application-level interface. In addition, SAGA provides various other useful abstractions, such as the File and RPC API, which ease the development of distributed applications. The following sections describes how the functionality of Migol can be seamlessly integrated with SAGA-CPR.

```
saga::cpr::service service (saga::url
    ("migol://flotta.haiti.cs.uni-potsdam.de
         :8443/wsrf/services/migol/AIS-JGroups"));
saga::cpr::self = service.get_self ();
```

Listing 1: **SAGA-CPR: Initialize Migol Session**

IV. SAGA Checkpoint Recovery API

The SAGA API specification [7] defines the core of the SAGA API as well as the mechanisms to extend the core API via additional functional packages. The SAGA-CPR API Package [19] is such an extension, which is currently an OGF working draft. The SAGA-CPR package provides a clean abstraction for starting, monitoring and recovering of checkpoint-restartable jobs. To support these use cases, applications can register checkpoints and job metadata with the infrastructure using this API.

For the management of CPR Grid jobs, SAGA defines the `cpr::job` and `cpr::service` class. The handling of CPR jobs is similar to regular jobs: A job is defined by a job description. In contrast however, CPR jobs require two job descriptions – one for starting and one for *re*starting the application. In addition, CPR jobs can be queried for checkpoint metadata, and explicitly checkpointed or recovered.

Listing 1 shows how applications can connect to the CPR infrastructure. During instantiation of the `saga::cpr::service` object the adaptor is able to register itself with the CPR backend. This step can be used for example, to register a monitoring endpoint. Using the `saga::cpr::self` object an application can obtain metadata about the current job from the application.

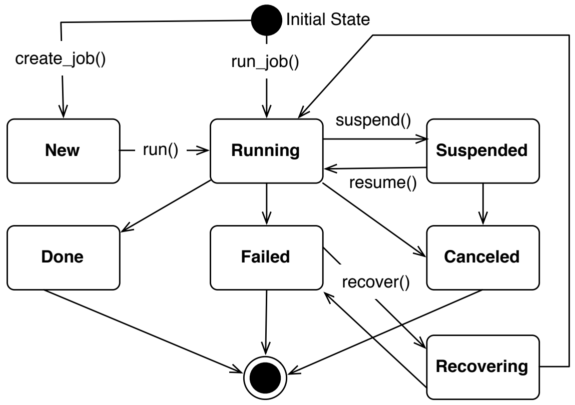

Fig. 2: **SAGA-CPR State Model: The CPR state model introduces the recovering state. This state indicates that an automatic recovery attempt is occurring.**

Further, applications can use a checkpointing API to update checkpoint metadata at the backend. The SAGA-CPR API allows the hierarchical organization of checkpoint files. Listing 2 illustrates the registration of a checkpoint file with the CPR framework.

```
saga::cpr::checkpoint remd_chkpt("remd_chkpt");
remd_checkpoint.add_file (saga::url
  ("gsiftp://qb.loni.org/work/remd/chkpt.dat"));
```

Listing 2: **SAGA-CPR: Checkpoint Registration**

CPR jobs are subject to an extended state model. An application can query or subscribe to a job's state via the `cpr::job` object. Figure 2 summarizes the CPR state model. The CPR model extends the SAGA job state model [7] by the new state `recovering`. This state is used to indicate that the infrastructure is currently trying to restart a job. If this recovery attempt fails, the state of the application is permanently set to `failed`. The application must then deploy an application-level recovery schema in order to continue the execution.

The Migol adaptor provides a compliant SAGA-CPR stack for the C++ reference implementation [20]. A major building block is the application-level monitoring mechanism used to detect failures. To support the monitoring of arbitrary SAGA applications, a monitoring Web service is started by the Migol adaptor with the initialization of a `cpr::service` object. This service is implemented using the gSoap HTTP server [21]. The Monitoring Restart Service will periodically send heartbeat messages to the application's monitoring service.

Another critical aspect is the management of the application's metadata. To ensure the recoverability of an application, metadata such as the job description and information about written checkpoint files must be available even if the application failed. The Migol adaptor relies on the AIS as the metadata backend. Every job is associated with a global unique identifier (GUID). All metadata belonging to an application can be stored or queried with reference to the GUID. The following information is propagated to the AIS by the Migol adaptor:

- The job description for starting and restarting of an application is mapped to the resource, service, and file profile of the Grid Service Object schema used by Migol. The registration of these information is done during the `create_job()` operation.
- The SAGA job state is mapped to the more comprehensive Migol model, which introduces additional states such as *migrating* or *pending*. All state transitions are directly propagated to the AIS. State queries, e. g. using the `get_state()` operation, are always conducted against the AIS. The Migol state is accessible via the `state_detail` metric of the job object.
- After startup of the application, the monitoring endpoint, i. e. the URL of the Web service, is updated.
- During runtime, metadata about written checkpoint are updated via the `cpr::checkpoint` object.

To ensure the availability of these information on failures, the AIS is actively replicated across multiple Grid nodes.

V. Implementing Replica-Exchange Using SAGA

The *Simple API for Grid Applications (SAGA)* [7] provides an easy-to-use standardised API for developing a broad range of distributed applications, including, but not limited to loosely-coupled data and/or pleasingly-distributed applications. In particular, SAGA offers an API for the management of checkpoint-recoverable jobs and file transfers that can be used over heterogenous distributed environment. Thus, SAGA is ideally suited for encoding the orchestration logic of RE simulations.

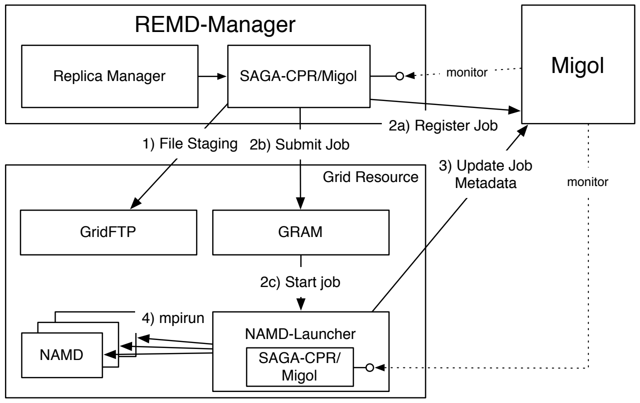

Fig. 3: **Components of REMD-Manager: The main part of the framework is the replica manager. The manager orchestrates a set of replica processes using the SAGA/CPR API. The Migol infrastructure ensures that the REMD-Manager and all replica processes are monitored and recovered if necessary.**

As illustrated in Figure 3, the proposed framework comprises of three components: The task manager, also referred to as *REMD-Manager*, is deployed on the user's desktop and provides the user interface to the overall REMD run. The second component is the Migol infrastructure that submits, monitors, and if required, recovers replica simulations. The last element is the task agent, the *NAMD-Launcher*, that resides on the High Performance machines where MD simulations are carried out. The NAMD-Launcher is triggered by the Grid job and is responsible for spawning and monitoring the MD run. NAMD [22], a highly scalable, parallel MD code, is used to carry out the MD simulation corresponding to each replica run. It is important to mention that any other MD code could be used just as simply and effectively.

The *REMD-Manager* is at the core of the framework; it orchestrates all replicas, i. e. the parameterization of replica tasks, file staging, job spawning and the conduction of the replica-exchange itself. This component heavily relies on the SAGA File and CPR API as well as the Python bindings for the implementation of the RE logic[1].

[1]The complete REMD-Manager code can be found at https://svn.cct.lsu.edu/repos/saga-projects/applications/REMDgManager/

Depending on the number of configured processes n, the REMD-Manager starts initially $\frac{n}{2}$ pairs of replicas. Each replica process is assigned a different temperature. Before launching a job the REMD-Manager ensures that all required input files are transfered to the respective resource. For this purpose, the SAGA File API and the GridFTP adaptor (step 1 in Figure 3) are used. The replica job is then submitted to the Grid resource using the CPR API and Migol/GRAM (step 2a-2c). Migol ensures that the the job description of each replica is stored within the Migol backend to ensure a later recovery. Globus GRAM is used to start the application.

To integrate NAMD with the SAGA/Migol infrastructure a SAGA based task agent – *NAMD-Launcher*, is used. This agent is responsible for updating the metadata of the application, i. e. the state, monitoring endpoint and new checkpoint URLs, at the Migol backend. The agent then launches the actual NAMD job using MPI. During the entire runtime the replica process is monitored by Migol using the monitoring endpoint of the NAMD launcher. This NAMD-Launcher enables the flexible orchestration of multiple NAMD jobs through the REMD-Manager without modification of the NAMD source itself.

When paired replicas reach a pre-determined states (eg., after a fixed number of steps), the decision as to whether to exchange paired-replicas is determined by the Metropolis scheme. If successful, parameters such as the temperature, are swapped. Both jobs are then relaunched using the mechanisms described above. Often the Metropolis scheme returns a negative result, and an exchange is not carried out; thus it is difficult to respond to an possible exchange speculatively.

In summary, SAGA allows the simple decoupling of the REMD application and orchestration logic from the underlying distributed infrastructure. All this whilst remaining general purpose and extensible, for example, using the Migol adaptor, the application can also benefit from additional features, such as the automatic monitoring and the transparent recovery of failed tasks.

VI. Experiences with REMD

To evaluate the performance of the REMD-Manager several experiments have been conducted on the LONI Grid [23]. The REMD-Manager is used to deploy tasks to the LONI clusters: QueenBee (QB), Poseidon and Eric. QB, which is both a LONI and a TeraGrid resource, is the largest LONI machine and has a peak performance of over 50 TFlops. Figure 4 gives an overview of the testbed.

The main objective of the first set of experiments is the quantification of the runtime overhead, which Migol-enabled applications, such as the REMD-Manager encounter. Scientific results obtained from using this infrastructure will be reported elsewhere.

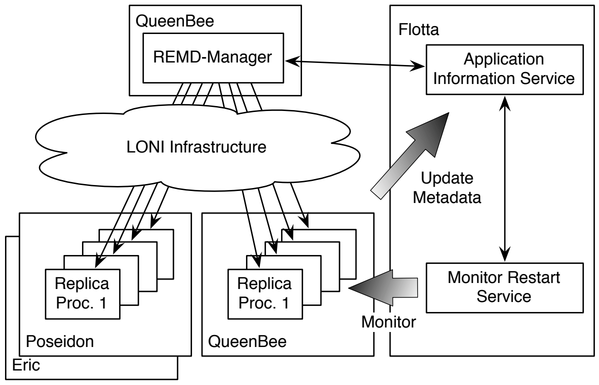

Fig. 4: **Fault-Tolerant MD Simulations: The REMD-Manager orchestrates a set of distributed replica processes using the SAGA API. All processes synchronize important metadata with the Migol infrastructure. Migol then actively monitors all processes and ensures that, even in the presence of failures, all task are eventually completed.**

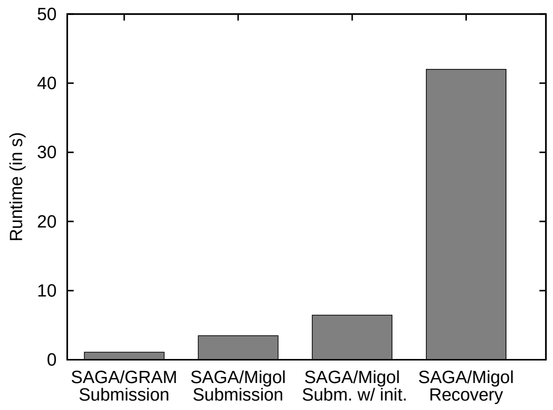

Fig. 5: **SAGA-CPR Migol Adaptor Overhead**

Figure 5 shows the response-times of SAGA-CPR submissions in comparison to their non fault-tolerant counterparts. Since each replica exchange step involves the relaunching of two replica jobs, the efficient spawning of remote tasks is a critical operation for the REMD-Manager. Initially, the submission time of a single NAMD task using SAGA-CPR/Migol is assessed. The experiment showed that a CPR/Migol job submission is on average 2 seconds slower than a GRAM submission. This overhead is mainly attributable to the additional metadata registration operation at Migol's AIS. For jobs that run on the order of hours, a couple of seconds overhead is effectively negligible.

Further, the Migol adaptor showed some additional initialization overhead. The overall runtime of the NAMD submission task including the initialization was 6.5 sec (bar 3 in Fig. 5), about 4.5 sec slower than the GRAM submission task. This overhead can be attributed to the initialization operations for setting up the HTTP server as well as the conduction of several metadata updates on the AIS. Since this initialization only occurs once after the startup of the REMD-Manager, this overhead is acceptable.

Number of NAMD steps	100
Number of MPI processes per NAMD run	16
Required staging files/size	6 files/10 MByte
Number of replica processes	2-8
Total number of replica-exchange steps	16

TABLE I: **REMD Application Characteristics For completeness we should probably mention the temperature range over which simulations were performed**

In addition, we investigated how the runtime of a single replica run is effected by Migol's active monitoring mechanism and the required checkpoint registration. For this purpose, a medium-size NAMD job was started with and without Migol support. Monitoring intervals between 20 s and 2 minutes were chosen to study the effect of monitoring frequency on runtimes. The update interval for checkpoint metadata was set to five minutes. Since the time measured for a checkpoint update operation was on average 1.3 seconds, we do not expect this to be a critical factor. The runtime of the NAMD job on QB without CPR/Migol amounted to 21.3 minutes. At worst a 1 minute overhead was observable with a monitoring interval of 20 s. With lower monitoring intervals, overheads were reduced further, e.g., the runtime overhead with a 2 minute monitoring interval was only 10 s, which is only slightly higher than the variance of typical NAMD runtimes.

We also evaluated the performance of the REMD-Manager. The REMD-Manager was configured to run a simulation with 2 to 8 replica processes and 16 replica-exchange steps. To stress test the Migol infrastructure very short NAMD tasks with only 100 steps have been used. Table I summarizes the REMD configuration used. The runtime of a REMD simulation depends to a great extend on the queuing time at the local resource management system. Thus, we attempted to minimize the queueing times during our experiment. However, as the results show, small queueing delays could not always be avoided. Figure 6 illustrate the results of this evaluation. Since the total number of replica exchange steps remained constant, the runtime decreases the more replica processes are used. With more than four replica processes a slight decrease of the efficiency can be observed. The more replica processes, the more dominant the sequential overhead at the REMD-Manager becomes. To emulate the most general case, where each exchange step requires the staging[2] of different files, in our setup, we staged six files with the total size of about 10 MB. This transfer, which required approximately 5-10 s on the LONI network. Due to the small problem set computed by each replica (only 100 NAMD steps, which require 35 seconds computation time on QB), this bottleneck becomes very evident. However, in more realistic scenarios with larger problem chunks this issue will be avoided.

[2]With a sophisticated data management strategy this size can further be reduced

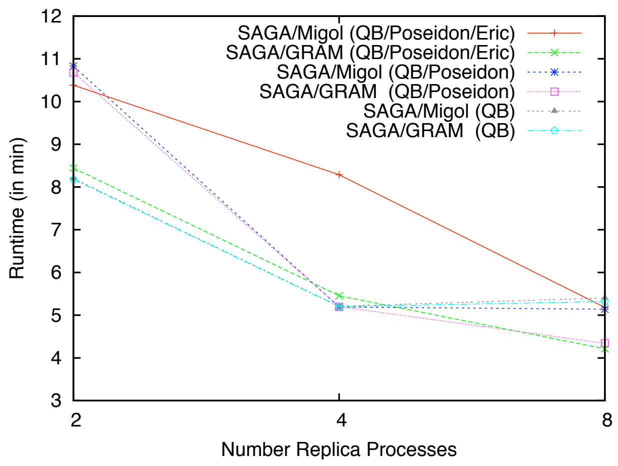

Fig. 6: **REMD Runtime: The time that it takes to complete 16 replica exchanges; each replica runs for 100 steps, before attempting an exchange with the paired-replica. Although the simultaneous deployment of replicas across multiple resources (labels with QB/Poseidon) has scheduling challenges compared to the usage of a single cluster (scenario QB), for scenarios studied here there is a slight reduction in time-to-completion.**

On average, with all factors considered, the SAGA/Migol adaptor added a total runtime overhead of about 15 seconds to the time-to-completion. It is important to note that this does not change significantly with either the number of replicas, number of replica-exchanges, nor the runtime of each replica. Thus the results indicate that the SAGA/Migol overhead is acceptable, corroborating earlier findings shown in Figure 5.

Figure 6 also shows that our approach can be employed to orchestrate multiple resources concurrently, as well as different resources (QB/Poseidon/Eric) individually. During the coupled distributed run, one half of the processes were allocated to the smaller machines Eric and Poseidon, while the bulk stayed on QB. As the number of replicas gets larger, the concurrent distributed runs have a lower time-to-completion than when QB was used in isolation was observed (all else being equal). However it is important to note, that smaller machines, such as Poseidon, showed long queuing times leading to high variance in the overall time-to-completion. This overhead is significant for short-running tasks and less so for longer running tasks.

Since the probability of a failure during a 10 minute run on a few resources is rather low, the reliability of the proposed framework was validated by introducing faults into the systems. We killed selected replica processes and measured the time required by Migol to restart the system. Due to the selected monitoring interval of one minute and failure threshold of 2 tries, the failure detection time averages to 2.5 minutes.

As shown in Figure 5, the recovery time required for the restart of the job is ~ 42 seconds. This is mainly caused by the complex interactions conducted by the Migol backend (cmp. section III): The monitoring service initializes the restart at the JBS. A major performance penalty is the delegation-on-demand mechanism required to obtain the credential of the user from the AIS – this procedure demands the creation of a public-private key pair, which is very costly. Further, the resource discovery and selection mechanisms used by Migol's JBS are designed with a focus on long-running applications, and currently show some substantial overhead, especially when used for short-running tasks.

While these results show that SAGA-CPR in conjunction with Migol incurs some overhead, we believe that this is acceptable compared to the benefits a fault-tolerant, self-healing infrastructure offers. In addition, it must be noted that further simple yet effective optimizations are possible. For example, by directly restarting jobs via the GRAM service a lot of the overhead caused by the dynamic discovery mechanisms of the JBS can be avoided. Further, we will evaluate possibilities to decouple the dispatching of replica runs from allocation of cluster resources to avoid long queuing delays. Systems, such as Falkon [24] or the Condor Glide-In [25] mechanism provide the possibility to acquire chunks of resources from the resource management systems, which can then be used to dispatch short tasks.

VII. Related Work

Previous CPR Efforts: Several frameworks for high-throughput computing and task farming exist, Condor-G [25], Nimrod-G [26], and Legion [27] to name a few. These provide basic fault tolerance support by automatic re-scheduling failed tasks. Advanced features such as the management of checkpoints however, are not supported. Further, these frameworks rely on a very simple failure detection mechanism – usually by simply polling the job state at the Globus gatekeeper. This allows the detection of some errors, but application-level failure detectors as used by the Migol/SAGA library can detect much more complex errors. For example, especially parallel applications can fail quite inconsistently: in the best case the application aborts, at worst the application hangs indefinitely. These kind of failures are not visible at Grid resource management system level.

At the level of related application programming interfaces for checkpointing, proprietary interfaces are dominant. This is because applications most often rely on application level checkpointing, and perform also their own checkpoint management (checkpointing policies, frequencies, dependencies, staging etc). The Open Grid Forum's[3] GridCPR group (Grid CheckPoint and Recovery) made an early attempt to describe a generic CPR architecture, and to define a generic CPR API, which would support applications to manage their complete checkpoint/recovery life cycle [9]. Based on that architecture, and on a set of CPR use cases [28], the SAGA group in OGF defined the CPR API package [19] (work in progress), whose implementation is described in this paper. The rendering of the CPR API in the SAGA API framework allows (a) to seamlessly combine CPR operations and other high level Grid programming abstractions provided by SAGA, and (b) to abstract from the actual implementation of the CPR mechanism. The CPR API which has been demonstrated with the Migol framework, can work as well with other systems, e. g. the XtreemOS system level checkpointing capabilities [29].

Other Distributed RE simulations: Several projects, such as Folding@home and WISDOM, utilise distributed infrastructures. While Folding@home [30][4] is based on BOINC [31], the WISDOM project utilizes the EGEE infrastructure. Although WISDOM has similar application characteristics as discussed here, the project is currently tied to the gLite [32] middleware. In contrast to WISDOM and Folding@home, our approach is not restricted to a specific distributed environment. SAGA based job-launching and file-handling is supported on most general-purpose Grids via the appropriate adaptors, as SAGA is a community specification and is soon to be standard [33]

VIII. Conclusion and Future Work

We have developed a fault-tolerant framework that implements a commonly occuring application usage pattern: loose-coupling of multiple tightly-coupled applications. The framework is general purpose and extensible to different usage patterns, deployment scenarios and specific simulation codes.

The fault-tolerant framework used to implement RE simulations in a production environment is created using the distributed programming interfaces provided by SAGA and its coupling to Migol. SAGA provides a middleware-independent, programing abstraction for distributed environments. RE simulations utilize the new SAGA-CPR API to interface with a checkpoint-recovery infrastructure, such as Migol. Using the newly developed SAGA adaptor for Migol, any SAGA application can re-use Migol's fault-tolerant services for monitoring and recovery. The application developer is not required to provide any special code, just the Migol adaptor must be configured. The Migol framework has strong self-healing capabilities: critical services, such as the Application Information Service (AIS) are able to automatically detect failures and reconfigure themselves, and thus addresses common failure modes in distributed environments without user interaction. In case

[3]`http://www.ogf.org`

[4]Folding@Home [30] is parallel replica dynamics but that is a special case of replica-exchange; when a certain event happens, there is a need for coordination amongst ALL replicas. We should maybe point this out, but I think it is fair at this level of detail, to consider folding@Home to be in the same application class to effectively parallelize simulations

of failures, e. g., a node-crash, applications are automatically restarted from the last saved checkpoint.

In contrast to other RE implementations on distributed simulations, it is critical to note and emphasise the general usability and extensibility – across different infrastructures, across a range of scientific applications and usage patterns (e.g. the multiple variants of the RE) – of our approach.

Acknowledgements

This work would not have been possible without the efforts and support of the wider SAGA team, especially Hartmut Kaiser, Ole Weidner and Joao Abecasis. We also thank Daniel Katz for hosting AL at the CCT, when this work was conceived. Important funding for SAGA specification and development has been provided by the UK EPSRC grant number GR/D0766171/1 (via OMII). SJ acknowledges the e-Science Institute, Edinburgh for supporting the research theme, "Distributed Programming Abstractions". This work has also been made possible thanks to the internal resources of the Center for Computation & Technology (CCT) at Louisiana State University and computer resources provided by LONI.

References

[1] U. Hansmann, "Parallel Tempering Algorithm for Conformational Studies of Biological Molecules," in *Chemical Physics Letters*, vol. 281, 1997, pp. 140–150.

[2] Y. Sugita and Y. Okamoto, "Replica-Exchange Molecular Dynamics Method for Protein Folding," in *Chemical Physics Letters*, vol. 314, 1999, pp. 141–151.

[3] B. Schroeder and G. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, 2006. [Online]. Available: http://www.pdl.cmu.edu/PDL-FTP/stray/dsn06.pdf

[4] H. Li, D. Groep, L. Wolters, and J. Templon, "Job Failure Analysis and Its Implications in a Large-Scale Production Grid," *e-science*, vol. 0, p. 27, 2006.

[5] O. Khalili, J. He, C. Olschanowsky, A. Snavely, and H. Casanova, "Measuring the Performance and Reliability of Production Computational Grids," in *GRID*. IEEE, 2006, pp. 293–300.

[6] A. Luckow and B. Schnor, "Migol: A Fault-Tolerant Service Framework for MPI Applications in the Grid," *Future Generation Computer Systems – The International Journal of Grid Computing: Theory, Methods and Application*, vol. 24, no. 2, pp. 142–152, 2008.

[7] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith, "A Simple API for Grid Applications (SAGA)," OGF Document Series 90, http://www.ogf.org/documents/GFD.90.pdf.

[8] A. Merzky, "SAGA CPR Draft," 2008.

[9] N. Stone, D. Simmel, T. Kielmann, and A. Merzky, "GFD.93 – An Architecture for Grid Checkpoint and Recovery Services," Open Grid Forum, OGF Informational Document, 2007.

[10] "WISDOM – Initiative for Grid-Enabled Drug Discovery Against Neglected and Emergent Diseases," http://wisdom.eu-egee.fr/.

[11] "Folding@Home," http://folding.stanford.edu/.

[12] S. Jha, H. Kaiser, Y. E. Khamra, and O. Weidner, "Design and Implementation of Network Performance Aware Applications Using SAGA and Cactus," in *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, 2007, pp. 143–150.

[13] S. Jha et al., *Programming Abstractions for Large-scale Distributed Application s*, to be submitted to ACM Computing Surveys; draft at http://www.cct.lsu.edu/~sjha/publications/dpa_surveypaper.pdf.

[14] M. R. Shirts and V. S. Pande, *Phys. Rev. Lett.*, vol. 86, no. 22, pp. 4983–4987, May 2001.

[15] Y. M. Rhee and V. S. Pande, "Multiplexed-Replica Exchange Molecular Dynamics Method for Protein Folding Simulat ion," *Biophys. J.*, vol. 84, no. 2, pp. 775–786, 2003.

[16] A. Luckow and B. Schnor, "Service Replication in Grids: Ensuring Consistency in a Dynamic, Failure-Prone Environment," in *Proceedings of Fifth High-Performance Grid Computing Workshop in conjunction with IEEE International Parallel & Distributed Processing Symposium*, Miami, USA, 2008.

[17] J. Schopf, L. Pearlman, N. Miller, C. Kesselman, I. Foster, M. D'Arcy, and A. Chervenak, "Monitoring the Grid with the Globus Toolkit MDS4," in *Journal of Physics: Conference Series – Proceedings of SciDAC*, 2006.

[18] R. Wolski, N. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *Journal of Future Generation Computing Systems*, vol. 15, no. 5-6, pp. 757–768, 1999.

[19] A. Merzky, "SAGA Extension: Checkpoint and Recovery API (CPR)," http://forge.ogf.org/short/saga-core-wg/drafts, OGF Informational Document, SAGA Core Working Group, 2007.

[20] H. Kaiser, A. Merzky, S. Hirmer, and G. Allen, "The SAGA C++ Reference Implementation," in *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06) - Library-Centric Software Design (LCSD'06)*, Portland, OR, USA, October, 22-26 2006.

[21] R. V. Engelen and K. Gallivan, "The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks," in *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2002, p. 128.

[22] J. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, pp. 1781–1802, 2005.

[23] "LONI: Louisiana Optical Network Initiative," http://www.loni.org.

[24] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: A Fast and Light-Weight TasK ExecutiON Framework," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

[25] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, *Cluster Computing*, vol. 5, no. 3, pp. 237–246, July 2002.

[26] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid," 2000. [Online]. Available: citeseer.ist.psu.edu/buyya00nimrodg.html

[27] S. J. Chapin, D. Katramatos, J. F. Karpovich, and A. S. Grimshaw, "The Legion Resource Management System," in *IPPS/SPDP '99/JSSPP '99: Proceedings of the Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1999, pp. 162–178.

[28] R. Badia, R. Hood, T. Kielmann, A. Merzky, C. Morin, S. Pickles, M. Sgaravatto, P. Stodghill, N. Stone, and H. Yeom, "GFD.92 – Use-Cases and Requirements for Grid Checkpoint and Recovery," Open Grid Forum, OGF Informational Document, 2006.

[29] J. Mehnert-Spahn, M. Schöttner, T. Ropars, D. Margery, C. Morin, J. Corbalán, and T. Cortes, "XtreemOS Grid Checkpointing Architecture," in *CCGRID '08: IEEE International Symposium on Cluster Computing and the Grid (poster)*, May 2008.

[30] M. Shirts and S. Pande, "Mathematical Analysis of Coupled Parallel Simulations," *Physical Review Letters*, vol. 86, no. 22, pp. 4983–4987, May 2001.

[31] D. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10.

[32] http://glite.web.cern.ch/glite/.

[33] http://saga.cct.lsu.edu.