Checkpoint and Restart for Distributed Components in XCAT3

Sriram Krishnan Dennis Gannon

Department of Computer Science, Indiana University.
215 Lindley Hall, 150 S Woodlawn Avenue, Bloomington, IN 47405-7104
{srikrish, gannon}@cs.indiana.edu

Abstract

With the advent of Grid computing, more and more high-end computational resources become available for use to a scientist. While this opens up new avenues for scientific research, it makes reliability and fault tolerance of such a system a non-trivial task, especially for long running distributed applications. In order to solve this problem, we present a distributed user-defined checkpointing mechanism within the XCAT3 system. XCAT3 is a framework for Component Component Architecture (CCA) based components consistent with current Grid standards. We describe in detail the algorithms and APIs that are added to XCAT3 in order to support distributed checkpointing. Our approach ensures that the checkpoints are platform independent, minimal in size, and always available during component failures. In addition, our algorithms maintain correctness in the presence of failures and scale well with the number of components, and checkpoint size.

Key Words: Grids, Components, Web Services, Distributed Checkpointing.

1. Introduction

A computational Grid is a set of hardware and software resources that provide seamless, dependable, and pervasive access to high-end computational capabilities. Resources on the Grid may be geographically distributed, and part of different administrative domains. While the growth in the availability of compute and data resources opens up whole new avenues for scientific research, maintaining reliability in a such a system is a non-trivial task. Reliability of resources becomes especially critical when long running distributed applications have to be orchestrated.

Until recently, there has been no consensus on the programming model that is appropriate for Grid computing. The Open Grid Services Architecture (OGSA) [13] is the first effort to standardize Grid functionality and produce a progamming model consistent with trends in the commercial Web service sector. The Open Grid Services Infrastructure (OGSI) refers to the basic infrastructure on which OGSA is built. At its core is the Grid Service Specification [10], which defines standard interfaces and behaviors of a Grid service in terms of Web services technologies.

On the other hand, the DOE sponsored Common Component Architecture (CCA) [9] project has been adopting a component based approach towards building large scale scientific applications. A software component is defined as a unit of composition with contractually specified interfaces and explicit context dependencies. A software component can be deployed independently and is subject to composition by third parties [17]. In our previous work [14], we presented XCAT3, a framework for CCA-based components consistent with current Grid standards. In this paper, we present the design and implementation of a distributed checkpointing and restart mechanism for components in the XCAT3 framework, in order to address fault tolerance for long running distributed applications on the Grid. Some of the issues we address in our system are:

Portability: Since no assumptions can be made about the architecture of the resources on the Grid, the checkpoints need to be architecture independent. Hence, we store the checkpoints in a platform independent XML-based format.
Checkpoint size: Only the minimal information required to restart an application needs to be stored. We take a user-defined checkpointing approach where the user is responsible for specifying the minimal state required to restart a component.
Interoperability: In order to be interoperable with standard Grid and Web service clients, no changes should be made to any protocols used. Hence, our algorithms and techniques work at a higher level of abstraction and make no changes to any protocols or semantics.
Scalability: The checkpointing algorithm should scale well with the number of components, as well as the checkpoint size. We show with an example that this holds true for our system.

Correctness: The algorithm should ensure that the checkpoints are *globally consistent*. Furthermore, the checkpointing mechanism itself should be resistant to failures - it should not leave the system in an inconsistent state if any of the parties fail during checkpointing.
Checkpoint availability: Since the most common cause of failure is the failure of the host that the component is executing on, the checkpoints cannot be stored locally as they may not be available in the event of a crash. Hence, the checkpoints are stored in stable storage at a separate location.

The remainder of this paper is organized as follows. We present background in Section 2. In Section 3, we introduce the XCAT3 framework, and in Section 4, we describe the mechanisms that are added to XCAT3 to support checkpoint and restart of components. In Section 5, we present a scenario and analyze the performance of our system. We present related work in Section 6, and our conclusions and future work in Section 7.

2. Background

2.1. Checkpointing for Fault Tolerance

Checkpointing of applications enables a basic form of fault tolerance in the presence of transient failures. Long running applications can save their state to stable storage, and roll-back to it upon failure. When an application is distributed, a *consistent global state* is one that occurs in a failure-free, correct execution, without loss of any messages sent from one process to another [3]. A *consistent global checkpoint* is a set of individual checkpoints that constitute a consistent global state. For a distributed application, a consistent global checkpoint is required to restart execution upon failure. In this subsection, we describe the main techniques used for checkpointing, and the types of algorithms used in practice to generate a consistent global checkpoint.

2.1.1. System-level versus User-defined Checkpoints. System-level checkpointing is a technique which provides automatic, transparent checkpointing of applications at the operating system or middleware level. The application is seen as a black-box, and the checkpointing mechanism has no knowledge about any of its characteristics. Typically, this involves capturing the complete process image of the application. User-defined checkpointing is a technique that relies on programmer support for capturing the application state. While a detailed comparison between the two approaches can be found in [15], we present some of the key differences:
Transparency: As defined above, system-level checkpointing is transparent to the user, while user-defined checkpointing is not. This means that user-defined checkpointing involves more programmer effort.
Portability: Transparent system-level checkpointing has proven to be hardly portable across heterogeneous architectures. However, user-defined checkpoints are quite portable since the user can store the checkpoints in a platform-independent format.
Checkpoint size: Since system-level checkpointing is oblivious to the details of the application, it is usually not possible to determine the critical state of the application. Hence, lots of unnecessary temporary data tends to get checkpointed leading to large checkpoint sizes. However, user-defined checkpointing relies on user support to store only the minimal checkpoint required for restart.
Flexibility: Rather than blindly checkpointing an application at regular intervals like system-level checkpointing does, user-defined checkpointing stores application state at programmer-defined logical states. This provides a higher degree of flexibility for the user, since these checkpoints can also be used for other purposes such as post-processing analysis, and visualization.

2.1.2. Consistent Global Checkpoints. Checkpointing for distributed applications can be broadly divided into two categories: uncoordinated, and coordinated. A detailed exposition of these checkpointing techniques can be found in [4]. We briefly describe them below.
Uncoordinated Checkpointing: In this approach, each of the processes that are part of the system determine their local checkpoints individually. During restart, these checkpoints have to be searched in order to construct a consistent global checkpoint. The advantage of this approach is that the individual processes can perform their checkpointing when it is most convenient. The disadvantages are that (1) there is a possibility of a *domino effect* which can cause the system to rollback to the beginning, (2) each of the processes has to maintain multiple checkpoints resulting in a large storage overhead, and that (3) a process may take a checkpoint that need not ever contribute to a consistent global checkpoint.
Coordinated Checkpointing: In this approach, the checkpointing is orchestrated such that the set of individual checkpoints always results in a consistent global checkpoint. This minimizes the storage overhead, since only a single global checkpoint needs to be maintained on stable storage. Additionally, this approach is also free from the domino effect. Algorithms that use this approach are either **blocking** or **non-blocking**.

Blocking algorithms typically are multi-phase. In the first phase, they ensure that all communication between processes is frozen. The communication channels are now empty, and the set of individual checkpoints constitute a consistent global checkpoint. The individual checkpoints are then taken, and subsequently all communication between processes can be resumed. Non-blocking algorithms typically use a communication-induced approach where each process is forced to take a check-

point based on protocol-related information piggybacked on the application messages it receives from other processes [3].

3. XCAT3: Distributed Components on the Grid

We briefly introduce the Common Component Architecture (CCA) and the Open Grid Services Infrastructure (OGSI) before presenting the architecture of the XCAT3 framework in the following subsections. A more detailed explanation of the XCAT3 architecture can be found in [14].

3.1. Common Component Architecture

The Common Component Architecture is defined by a set of framework services, and the definitions of the components that use them. Each component communicates with other components by a system of *ports*. There are two types of CCA ports: **provides ports** which are the services offered by the component, and **uses ports** which are the stubs that a component *uses* to invoke the services *provided* by another component.

A uses port of one component can be connected to a provides port of another component as long as they implement the same interface. Connections between uses and provides ports are made at runtime. A component needs to execute a `getPort` statement to grab the most recent reference to provider, and a `releasePort` when it has finished using it. The get/release semantics of component connections enable the framework to infer if any port calls are being made at any point in time, and also enable the connections to be changed dynamically.

Every CCA component implements the *Component* interface, which contains the `setServices` method responsible for setting the *Services* object for the component. The Services object can then be used to add and remove ports, and get and release them as needed. The Services object also contains a *ComponentID* which has methods that uniquely identify the component, and provide metadata about it. The most important framework service that CCA defines is the *Builder* service for creation and composition of these components.

3.2. Grid Services

The Open Grid Services Infrastructure extends the Web services model by defining a special set of service properties and behaviors for stateful Grid services. Some of the key features of OGSI that separate Grid services from simple Web services are:

Multiple level naming: OGSI separates a logical service name from a service reference. A Grid Service Handle (GSH) provides an immutable location-independent name for a service, while a Grid Service Reference (GSR) provides a precise description of how to reach a service instance on a network, e.g a WSDL reference. A GSH can be bound to different GSRs over time.

Dynamic Service Introspection: Grid services can expose metadata about their state to the outside world through the use of Service Data Elements (SDE). SDEs are XML fragments that are described by Service Data Descriptors (SDD). SDEs can be queried by name or type, and can be used to notify state changes to clients.

Standardized ports: Every Grid service implements a *GridService* port, which provides operations to query for SDEs, and manage lifetime of the Grid service. OGSI also specifies standard ports for creation, discovery, and handle resolution.

Recently, OGSI has been superseded by the Web Service Resource Framework (WSRF) [1], which represents the refactoring of the functionality of OGSI into a framework of independently useful Web service standards, and the alignment of the functionality of OGSI with current and emerging Web service standards.

3.3. XCAT3 Architecture

Currently, the XCAT3 framework is implemented in Java, and we plan to implement a C++ version that is interoperable with the former. In XCAT3, we implement the CCA specification in the context of the Grid services specifications. Some of the key features of XCAT3 components are as follows.

Ports as Grid services: As per the CCA specification, one component can have more than one provides port of the same type. Simple Grid and Web services allow multiple ports of the same `portType`; however, multiple bindings of the same port are semantically equivalent. Hence, the same operation on different ports of the same type affects the service in exactly the same way. However, unlike Web service ports, ports in CCA are designed to be stateful. Hence, every provides port in XCAT3 is implemented as a separate Grid service. The consequence of this is that every provides port inherits multiple level naming from the OGSI specification, and this enables the ports to be location independent. Additionally, any Grid service that is compliant with the OGSI specification can serve as a provides port. Uses ports are just client-side stubs for the remote provides ports.

ComponentID as a Grid service: The ComponentID, as specified by the CCA specification, is also implemented as a Grid service. It exposes handles and references of all the provides ports that a component contains as SDEs, and acts as a manager for the component. Users can query a compo-

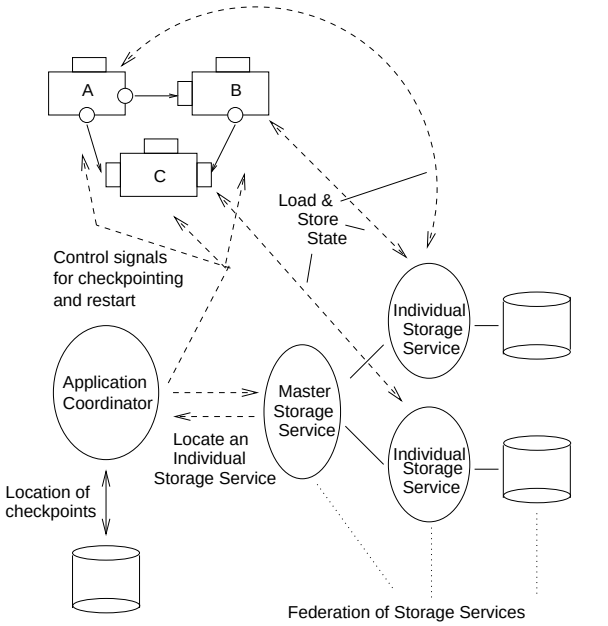

Figure 1. Checkpointing & restart big picture

nent for the types of services provided via the ComponentID, and connect to them directly.

Some of the useful services in the XCAT3 framework are:

Builder service: As mentioned before, the Builder service defines methods for component creation and composition. We allow remote instantiation of components via ssh or Globus GRAM [7] provided by the Java CoG [18] kit. For composition purposes, the Builder service provides `connect` and `disconnect` methods for connecting and disconnecting a uses port to a provides port respectively. Once the ports have been connected, all communication between them is via Web service invocations provided by the XSOAP [5] toolkit.

Handle Resolver: Since we employ multiple level naming for our ports and ComponentIDs, we use a handle resolution mechanism that translates a GSH to a GSR. This is provided by the Handle Resolver service. The Handle Resolver, as other Grid services in the XCAT3 framework, is implemented using the GSX [12] toolkit, which provides a lightweight implementation of the OGSI specification.

4. Checkpoint and Restart in XCAT3

In this section, we discuss in detail the various services, algorithms, and implementation of the checkpoint and restart mechanism in XCAT3.

4.1. Architecture

In order to add the capability to perform checkpoint and restart of components, we added the following services to the XCAT3 framework, as shown in Figure 1:

Application Coordinator: The Application Coordinator provides operations to checkpoint and restart an application composed of a set of XCAT3 components. The Application Coordinator contains the list of GSHs for each of the components constituting the application. Additionally, it also contains all information required to restart the application, viz. deployment information for the components, locations where they are currently executing, the protocols used to instantiate them, etc. An instance of an Application Coordinator can be uniquely identified by means of an *applicationID*. The Application Coordinator can be *passivated* by storing all this information into persistent storage, and can be *activated* from persistent storage using the applicationID only when it is required, e.g. when a checkpoint or restart has to be initiated.

Storage services: As we point out earlier, the checkpoints need to be stored in stable storage, and should be available upon failure. Additionally, the stable storage should scale well with the number of components. Although a scalable and reliable persistent storage service is not the primary research focus of this paper, we provide a proof-of-concept implementation of a federation of Storage services to address these requirements.

The federation of Storage services is comprised of a *Master Storage service* and a set of *Individual Storage services*. The Master Storage service contains references for every Individual Storage service. Whenever a client needs to store data into the Storage services, it contacts the Master Storage service which assigns to it a reference to an Individual Storage service from the available pool. The client uses this reference to send its data to the Individual Storage service, which stores it into stable storage and returns a unique *storageID* which can be used to access the data at a later time.

Component & ComponentID: The component writer is expected to generate the minimal state required to restart a component. However, the component writer can not be expected to generate framework specific information, e.g. connections between uses and provides ports. The component writer can also not be expected to write code to access the Storage services to load and store the states. We follow an approach where the component writer writes code to generate local state specific to the component instance, while the rest of the work is done by the framework.

In order to help the component writer generate and load component state, the component writer is provided with a *MobileComponent* interface that extends the regular CCA Component interface. Inside this inter-

face, methods to generate and load component states are added with an assumption that the framework would invoke these as and when required. Since the outside world interacts with a component using its ComponentID, we add operations for loading and storing component state into a *MobileComponentID* interface, which extends the CCA ComponentID interface. We also add other control operations to the MobileComponentID which help in the process of checkpointing & restart. The MobileComponentID implementation provided by the framework retrieves the local component state by making a callback on the *MobileComponent* interface. Additionally, it generates the states of the Services object, viz. the uses and provides ports added, and their connection information, and also of the Service Data Elements for the various Grid services that are part of the component. This complete component state is encapsulated in a platform-independent XML format, and is transferred to the Storage services as required.

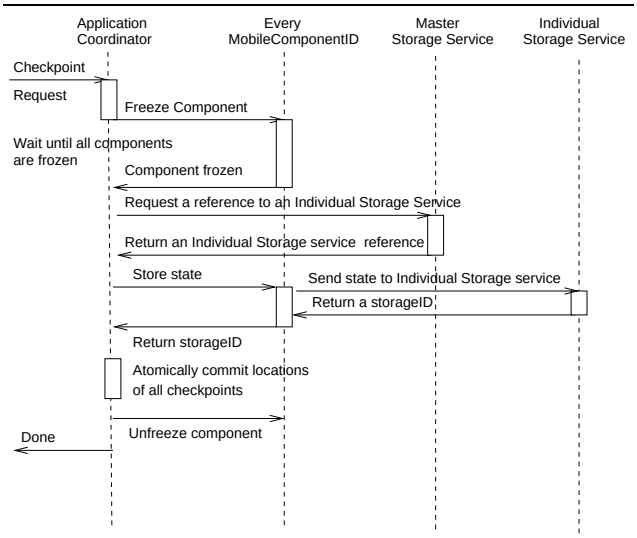

Figure 2. Distributed checkpointing algorithm

4.2. Checkpointing Algorithm

In XCAT3, a checkpoint can be initiated by a user by activating an instance of the Application Coordinator using an applicationID, and then invoking a `checkpointComponents` operation on it. As we mention in Section 2, a non-coordinated approach can lead to a surplus of checkpoints and is vulnerable to the domino effect, while non-blocking coordinated algorithms are typically communication-induced, and may involve changes to the underlying messaging layer. Since we prefer to use commodity implementations of the messaging layer for purposes of interoperability, we choose to implement a coordinated blocking algorithm in order to create a consistent global checkpoint.

The checkpointing algorithm is shown in Figure 2. Details are as follows:

1. The Application Coordinator retrieves the references for all components that are part of the distributed application.
2. For every component that is part of the application, the Application Coordinator sends a `freezeComponent` request to the appropriate MobileComponentID.
3. On receipt of the `freezeComponent` request, the MobileComponentID implementation waits until all remote invocations are complete. It does so by waiting until all uses ports in use are released via a `releasePort` invocation. Subsequently, all `getPort` calls block until further notice. After all ports are released, the MobileComponentID implementation sends a `componentFrozen` message to the Application Coordinator.
4. On receiving `componentFrozen` messages from every component that is part of an application, the Application Coordinator can infer that all communication between the components is stalled, and that individual checkpoints can now be taken. Subsequently, for every component, it contacts the Master Storage service to receive a reference for an Individual Storage service, and sends these references to their MobileComponentID's as part of the `storeComponentState` message.
5. On receipt of the `storeComponentState` message, each of the MobileComponentID implementations generate and store the complete state of the components into the Individual Storage services referenced by the messages. They return to the Application Coordinator the storageID's received from the Individual Storage services.
6. On receiving the storageID's from every MobileComponentID implementation, the Application Coordinator stores a list of {*instanceHandle*, *individualStorageServiceHandle*, *storageID*} tuples into stable storage, which can be used to locate the checkpoints if need be. It also removes prior checkpoints and tuples referring to them, if they exist. Since we don't want a situation where we end up with locators for a set of checkpoints referring to a combination of old and new ones, this step is performed *atomically* using transaction support provided by a MySQL database.
7. The Application Coordinator then sends `unfreezeComponent` messages to every MobileComponentID implementation signifying the end of the checkpointing process. All blocked `getPort` calls can now proceed as expected.

We do not attempt to formally prove the correctness of our distributed checkpointing algorithm since it is just a flavor of coordinated blocking distributed checkpointing algorithms. Coordinated blocking distributed checkpointing algorithms are well known to be correct in the distributed systems community. However, we present two key arguments

towards its correctness:

Consistency: By blocking all communication between the components that are part of the application, we reduce the problem of distributed checkpointing to that of individual checkpointing of the set of components since the message channels between the components are now empty. Hence, the set of individual checkpoints now constitute a consistent global checkpoint which can be used for restarting the application.

Atomicity: We have to ensure that we never end up with a global checkpoint that contains a combination of new, as well as old checkpoints. We ensure that this does not happen in our system by atomically updating the information about the global checkpoint using transaction support provided by a MySQL database. Thus, either the global checkpoint contains all new checkpoints, or all old ones if the transaction fails, but never a combination of both.

4.3. Restart Algorithm

An application can be restarted also by invoking the `restartFromCheckpoint` method on the Application Coordinator, using an applicationID. During restart, it is not sufficient to restart just the failed components from the most recent consistent global checkpoints. This is because messages might have been exchanged between the components after the most recent checkpoint was taken. Hence, the restart algorithm restarts every component from the most recent global checkpoint.

Furthermore, when all components are restarted from the global checkpoint, it has to be ensured that all components have their states set from the checkpoint before any execution threads are resumed. If there are some components whose states have not been set from the global checkpoint before some of the execution threads are resumed, the resumed threads might cause inconsistencies if they interact with other components whose states have not yet been set. Details of the algorithm are as follows:

1. The Application Coordinator retrieves handles for all components that are part of the system. It destroys instances of all components, if they are still alive. This has to be done because it is not practical to easily roll back the control and data of processes that are in the middle of their execution. It is much easier to destroy executing threads, and restart them from the global checkpoint.

2. Using the XML deployment descriptors for the components, the Application Coordinator re-instantiates every component with the help of the Builder service. Since re-instantiated components signify the same component instances, the GSH's for the MobileComponentID's of the component instances are reused.

3. For every component, the Application Coordinator sends a `loadComponentState` message to the appropriate MobileComponentID along with the location of the Individual Storage service and storageID needed to retrieve the component state.

4. On receipt of the `loadComponentState` message, the MobileComponentID implementations load the state of the component from the Individual Storage service. The Services objects are initialized, and all ports are initialized using their original GSH's. New references for the ports are registered with the Handle Resolver service. The MobileComponentID implementations send a confirmation to the Application Coordinator once this is complete.

5. After the Application Coordinator receives confirmation from all components that their states have been loaded, it sends a `resumeExecution` message to every MobileComponentID implementation.

6. On receipt of the `resumeExecution` message, the MobileComponentID implementation forwards it to the component which can now resume all threads of execution. Whenever these threads use a `getPort` call for the first time to gain access to a uses port, a fresh reference for the remote provides port is retrieved from the Handle Resolver service and all communication can proceed seamlessly.

5. Sample Application

The Master-Worker model is a commonly used model for long running computations. Indeed, several examples of the model have been implemented using XCAT, such as the Chemical Engineering [8] application that we worked on in collaboration with NCSA, and the Collision Risk Assessment System (CRASS) [6] worked on at Indiana University. As a sample application to test our performance, we use a Master-Worker dummy application, which exactly follows the execution model of the above examples.

The Master-Worker XCAT3 application consists of a single Master component, and N Worker components. The Master component contains a *MasterPort*, which is a provides port that is responsible for receiving a work packet from a user, and processed results from Workers as an array of bytes. The Worker component contains a *WorkerPort*, which is a provides port that is responsible for receiving a work packet from the Master as an array of bytes. Additionally, the Master component contains uses ports to connect to each of the Worker's WorkerPort, and every Worker component contains a uses port to connect to the Master's MasterPort. The interactions between the Master and the Workers is shown in Figure 3.

To store states of the Master and Workers, we make a distinction between a *super-state* and the actual physical state. We define a super-state as a logical block of execution in

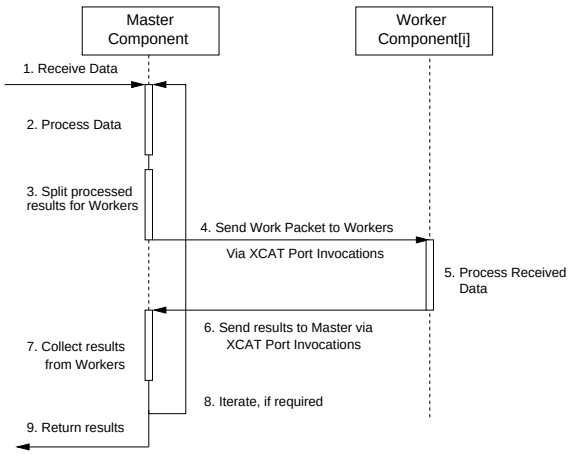

Figure 3. Master & Worker Interaction

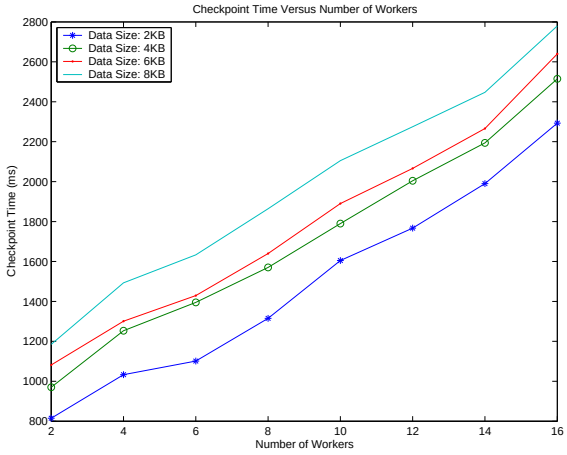

Figure 4. Checkpoint Time Versus Workers

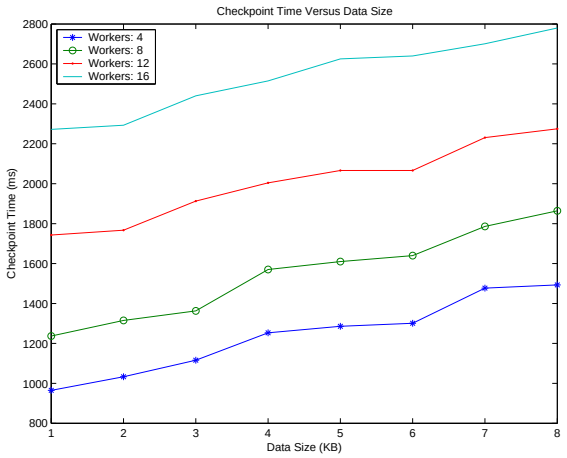

Figure 5. Checkpoint Time Versus Data Size

the program. Within the super-state, the physical state may still vary depending on the state of the internal data structures at any point of execution. From Figure 3, we infer that the Master has the following possible super-states: (1) INITIALIZED, where it has received its work packet from a client, (2) PROCESSING_DATA, where it is processing data received, (3) SENDING_DATA, where it has finished processing and has sent a work packet to 0 or more Workers, and (4) RECEIVING_DATA, where it has received results back from 0 or more Workers. All others super-states can be inferred with minimal overhead from the above. Similarly, the Worker has the following possible super-states: (1) INITIALIZED, where it has been been initialized with all connections, and (2) PROCESSING_DATA, where it is processing the data received from the Master. Using the above analysis of the states, the `generateComponentState` and `setComponentState` are coded such that the actual physical state is generated and set appropriately with respect to the super-states defined. Additionally, the `resumeExecution` is coded such that it can restart execution from the same. The only other thing to ensure is that either the individual states do not change during execution of the `generateComponentState` method, or if they do, they do so atomically, i.e. modified data and their indices get updated at the same time. This is easily done by synchronizing all state changes inside mutually exclusive blocks.

In order to measure the performance for the above application, we scripted it such that we could parameterize the number of Workers, and the size of data that is being processed. The experiments were run on an 8-node Linux cluster. Each of the cluster nodes is a dual processor system with 2.8GHz Intel Xeon processors and 2GB of memory running Redhat Linux 8.0. The version of Java used is 1.4.2. All the components and storage services are run on the same cluster. We plot the time to checkpoint a set of components against the number of Workers keeping the data size per Worker constant in Figure 4, and against the data size per Worker keeping the number of Workers constant in 5. In both cases, there is a single Master Storage service, and 8 Individual Storage services.

We expect our algorithm to be linear with respect to both the number of Workers, and data size. This is validated by our figures. Additionally, the time taken to checkpoint the application is acceptable considering the fact that our implementation is in Java, the checkpoints are stored in XML format, and that we use SOAP for our remote invocations. We envision the use of the checkpointing and restart capability for long running applications, in which case the impact of checkpointing on processing time would be negligible.

6. Related Work

Checkpointing and restart for applications has been implemented by a number of systems. Condor [2] is a popular system that provides system-level checkpointing and restart for applications on the Grid. However, Condor does not provide checkpointing for applications that are multi-process, and which involve any sort of inter-process communication. There are several systems that provide distributed checkpointing for parallel MPI and PVM based applications, such as [11], and [16]. However, portability of checkpoints is not a big requirement for MPI applications since they are typically run on a single cluster. Hence, most of the above systems use a system-level approach to distributed checkpointing.

Recently, the Grid Checkpointing and Recovery (Grid-CPR) group at the Global Grid Forum has been working on user-level APIs and associated services that will permit checkpointing and restart of applications on heterogeneous Grid resources. However so far, they have only concentrated on checkpointing and restart of single process jobs. In the context of components, both Enterprise Java Beans and CORBA Components provide mechanisms to load and store state into persistent storage. However, in both cases, they are primarily concerned with persisting the state of the data encapsulated by the component, and not the state of the execution. Furthermore, they only concentrate on storage of individual component states, and do not explicitly handle global states.

7. Conclusions

In this paper, we described the support for user-defined checkpointing and restart for distributed applications within the XCAT3 framework. We presented the architecture of XCAT3, the APIs and algorithms used for checkpoint and restart, and with the help of a sample application demonstrated that our implementation scales well with the number of components and checkpoint size.

Currently, we are working on migration of individual components for performance reasons and policy violations. As part of our future work, we plan to integrate our system with the WSRF specification, when stable implementations for it become available. Additionally, we plan to add monitoring for our components in order to identify failures as soon as they occur. At present, we do not deal with link failures between components and we plan to address this in the future.

References

[1] Globus Alliance, IBM, and HP. Web Service Resource Framework, June 2004. http://www.globus.org/wsrf.

[2] Jim Basney, Miron Livny, and Todd Tannenbaum. High Throughput Computing with Condor. In *HPCU news, Volume 1(2)*, June 1997.

[3] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Transactions on Computer Systems*, volume 3, Feb 1985.

[4] E.N. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A Survey of Rollback-recovery Protocols in Message Passing Systems. Technical report, School of Computer Science, Carnegie Mellon University, 1996. CMU-CS-96-181.

[5] A. Slominski et al. Design of an XML based Interoperable RMI System : SoapRMI C++/Java 1.1. In *International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Pages 1661-1667*, June 25-28 2001.

[6] D. Gannon et al. Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. In *Special Issue on Grid Computing, Journal of Cluster Computing*, July 2002.

[7] K. Czajkowski et al. A resource management architecture for metacomputing systems. In *IPPS/SPDP 98, Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[8] M. Govindaraju et al. XCAT 2.0: Design and Implementation of Component based Web Services. Technical report, C.S. Dept., Indiana University, June 2002. TR562.

[9] R. Armstrong et al. Toward a Common Component Architecture for High-Performance Scientific Computing. In *8th IEEE International Symposium on High Performance Distributed Computation*, August 1999.

[10] S. Tuecke et al. Grid Service Specification, April 2003. http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-29_2003-04-05.pdf.

[11] Sriram Sankaran et al. Checkpoint/Restart System Services Interface (SSI) Modules for LAM/MPI. Technical report, C.S. Dept., Indiana University, 2003. TR578.

[12] Indiana University Extreme Computing Lab. Grid Service Extensions (GSX), Dec 2003. http://www.extreme.indiana.edu/xgws/GSX.

[13] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer 35(6)*, 2002.

[14] S. Krishnan and D. Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, April 2004.

[15] L.M. Silva and J.G. Silva. System-level versus User-defined Checkpointing. In *Seventeenth Symposium on Reliable Distributed Systems*, Oct. 1998.

[16] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *10th International Parallel Processing Symposium*, 1996.

[17] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[18] G. von Laszewski et al. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 25, Commodity Grid Kits - Middleware for Building Grid Computing Environments. Wiley, 2003.