AN ARCHITECTURE FOR CHECKPOINTING AND MIGRATION OF DISTRIBUTED COMPONENTS ON THE GRID

Sriram Krishnan

Department of Computer Science

Indiana University

Submitted to the faculty of the Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science

Indiana University

November 2004

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Doctoral Committee

Dennis Gannon, Ph.D.
(Principal Advisor)

Randall Bramley, Ph.D.

Andrew Lumsdaine, Ph.D.

Sept 15, 2004

Beth Plale, Ph.D.

To my parents

Acknowledgements

There are several people I wish to acknowledge for helping me during the difficult, yet rewarding, path towards a Ph.D. Without them, it would have been impossible for me to fulfil the goals that I set out to achieve in graduate school.

First and foremost, I am indebted to my advisor, Prof. Dennis Gannon, for his guidance and encouragement throughout my research career at Indiana University. I am grateful to him for letting me pursue my research interests with sufficient freedom, while being there to guide me all the same. Working with him has been one of the most rewarding experiences of my professional life.

I am also very thankful to my other committee members, Prof. Randall Bramley, Prof. Andrew Lumsdaine, and Prof. Beth Plale for their comments and constructive criticisms during the course of my research work and during the preparation of this dissertation. I am particularly thankful to Prof. Randall Bramley for his candid insights not only on my research, but also on other aspects of life.

I have been fortunate to work with a wonderful set of people, past and present, at the Extreme Lab. My discussions with them have spurred me on, and helped me grow professionally. In particular, I would like to thank Ken Chiu for numerous fruitful discussions that I have had with him on various aspects of my research. I am also thankful to Alek Slominski for providing constructive feedback on my work, and also for providing technical support for the XSOAP Toolkit that I used in my implementation. I would also like to thank Yogesh Simmhan for supporting the GSX Toolkit that I also used in my implementation. I am thankful to Madhu Govindaraju for helping me with various aspects of the Ph.D. process- research and otherwise.

I am grateful to the administrative staff of the Computer Science department, who have been extremely helpful throughout my stay here. I am especially thankful to Sherry Kay, our Graduate Secretary, who has kept me on top of all the necessary paperwork. I am also thankful to the Systems staff at the Computer Science department, who are one of the best that I have ever worked with.

I am extremely thankful to my friends, whom I have hung out with over the best part of five years in Bloomington. I have counted on them to keep me sane throughout graduate school, and appreciate their patience when I have been stressed out on several occasions with deadlines at work. I especially appreciate the quality time that I spent with them playing various sports. Volleyball, Cricket, Basketball, and other racquet sports have helped me re-charge my batteries on more than one occasion, and I thank my various sports teams and partners for the same.

Last but certainly not the least, I owe a great deal to my family for providing me with emotional support during graduate school. My parents have been very supportive; I am thankful to them for not imposing any unrealistic expectations on me, but always being there for me nonetheless. My elder brother, Santosh, has been someone I have always looked up to, and I am thankful to him for being a phone call away if I needed advice on anything under the sun.

Abstract

Sriram Krishnan

AN ARCHITECTURE FOR CHECKPOINTING AND MIGRATION OF DISTRIBUTED COMPONENTS ON THE GRID

A computational Grid is a set of hardware and software resources that provide seamless, dependable, and pervasive access to high-end computational capabilities. The Grid differs from other computational resources such as traditional supercomputers and clusters by the following key features: (1) coordination of resources that are not subject to centralized control, (2) use of standard, open, general purpose protocols and interfaces, and (3) delivery of non-trivial qualities of service despite unpredictable resource availabilities.

The Open Grid Services Architecture (OGSA) is the first effort to standardize Grid functionality, based on concepts from the Web services community. However, the Web services based OGSA presents a server-centric approach which is not very conducive to the orchestration of complex distributed applications where the interactions are not always of the client-server type. We present a distributed component based approach for composing complex applications on the Grid that is conformant with the Common Component Architecture (CCA), while maintaining compatibility with Grid standards.

Because Grid resources are not subject to centralized control and are geographically distributed, their availabilities may be very dynamic in nature. Migration of individual components can be an effective strategy for dealing with dynamic resource availabilities. However, migration of components that are part of a distributed application is complicated due to the possible interactions between them during execution. We present an approach for migration of distributed components, in the presence of communication between them. Additionally, reliability of Grid resources is also very difficult to guarantee. Checkpointing applications and rolling back to a saved state is an effective form of fault tolerance for dealing with failures of such resources. However, due to the distributed nature of the applications, the checkpoints generated need to be globally consistent. We present our approach for checkpointing and restart of distributed components for fault tolerance purposes.

Contents

List of Figures

1

Introduction

1.1 Motivation

A computational Grid [30] is a set of hardware and software resources that provide seamless, dependable, and pervasive access to high-end computational capabilities. By enabling the use of teraflop computers and petabyte storage systems interconnected by gigabit networks, the Grid enables scientists to explore new avenues of research hitherto irrealizable via conventional computing resources. The Grid differs from other computational resources such as traditional supercomputers and clusters by the following key features [29]: (1) coordination of resources that are not subject to centralized control, (2) use of standard, open, general purpose protocols and interfaces, and (3) delivery of non-trivial qualities of service.

The Open Grid Services Architecture (OGSA) [31] is the first effort to standardize Grid functionality, based on concepts from the Web services community. Web services are independent of programming languages & models, and system software, and have been adopted in the industry as a standard for building enterprise applications. Adoption of a Web services based framework for the Grid is useful for dynamic discovery and composition of services required for the creation of distributed, dynamic *virtual organizations* for coordination of a decentralized set of resources. Additionally, due to the widespread adoption of Web service technologies, standard protocols and tools are available for use to the Grid community. Several Web services standards have been defined, e.g. SOAP [20] which provides an XML-based messaging protocol between service providers and requestors, Web Services Description Language (WSDL) [22] which provides a way to describe and access Web services, and Business Process Execution Language (BPEL) [23] which provides a way to orchestrate long running Web-services based applications by *composition in time* of simpler Web services.

The Open Grid Services Infrastructure (OGSI) refers to the basic infrastructure on which OGSA is built. At its core is the Grid Service Specification [26], which defines the Grid as an extensible set of *Grid services* that may be aggregated in various ways to meet the needs of the virtual organizations. It defines standard mechanisms for creating, naming, and discovering service instances, and provides location transparency and dynamic service introspection. OGSI has recently evolved into the Web Service Resource Framework (WSRF) [25], which provides an even tighter integration of Web service technologies with those of the Grid.

On the other hand, various researchers in industry, as well as in academia, have been promoting the use of software components for orchestration of complex applications. A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. A software component can be deployed independently and can be subject to composition by third parties [53]. A component architecture is a system defining the rules of linking components together. The software engineering benefits of component based software are well known: they enable encapsulation, modular construction of applications and software reuse. Component systems have proven to be immensely useful to scientists who wish to build complex distributed applications by composing existing software components, thereby shielding them from the underlying complexity of the distributed set of resources.

Several component models have proven to be successful in various domains. Microsoft's COM [45] and DCOM [46] frameworks have been fundamental to inter-operability in Windows based applications. Their current Web services oriented .NET framework is also component based and is gaining widespread acceptance. In the CORBA world, the Object Management Group has released a specification for the CORBA Component Model (CCM) [7], whereas Java Beans and Enterprise Java Beans (EJB) [47] have been popular component standards for Java based applications. The Common Component Architecture

(CCA) is an initiative by DOE laboratories and universities to develop a common architecture for building large scale scientific applications from well-tested software components that run on both parallel and distributed systems. Several implementations of CCA exist, viz. XCAT [34], Ccaffeine [5], and SCIRun [38].

Component architectures such as CCA and CCM define standard mechanisms for components to define services they *provide*, and the ones that they *use*. Components can be composed with other peer components in such systems, by direct connections between matching provides and uses sides. We refer to this as *composition in space*, since it allows composing components distributed across space at the same time. Standard Grid and Web services do not provide this style of composition, since they are traditionally accepted to be self-contained entities. Hence, plain Web services are better suited for client-server applications where a Web service client requests a service from a service provider, rather than *truly* distributed ones where communication patterns between entities are more peer-to-peer. Publish-subscribe systems such as WS-Notification [4] that provide asynchronous communication between Web services enable direct communication between Web services; however, these are not very suitable for third-party composition. We propose an approach where distributed applications can be composed in space using CCA-based constructs, while being consistent with Grid and Web service specifications. This preserves interoperability with Grid standards, while enabling orchestration of more complex applications than the ones possible by standard Grid and Web services.

In addition to this, as we state earlier, an important requirement for computational Grids is the provision of non-trivial qualities of service to applications. Since the Grid resources belong to different administrative domains and are geographically distributed, their availabilities may be very dynamic. Additionally, reliability of such resources is also very difficult to guarantee.

The above problems are magnified if the applications running on the Grid are long running, as well as distributed. In such a situation, it is highly desirable for an application to be able to adapt to the dynamic Grid environment. Since resource availabilities may change over the course of execution of long running applications, an application should be able to store its state onto stable storage, and migrate to more suitable resources if need be. Suitability of resources can be determined by resource quality as well as policies defined by the resources and end users.

Resources can not only vary in their availabilities, but also fail during the course of execution of the application. In such as situation, it is highly beneficial to checkpoint the application state at regular intervals so that it can be restarted upon resource failure. Additionally, if the application is distributed with various components communicating with each other, the checkpointing process is more complicated since it has to be *globally consistent*. In other words, it should be a state that occurs in a failure-free, correct execution, without loss of any messages sent from one component to another [11].

We propose a user-defined scheme for persistence of components which is leveraged by the framework to provide abilities to (1) migrate individual components if need be, and (2) checkpoint the state of the entire application and restart upon failures.

1.2 Contributions

This dissertation addresses three key problems in Grid computing - programming mechanisms for orchestration of complex long running distributed applications, adaptability of these applications to the inherently dynamic availabilities of Grid resources, and their ability to recover from resource failures.

In this dissertation, we propose that long running distributed applications on the Grid should be orchestrated by composition of individual simpler components, both in space and time. Components composed in space may be executing concurrently on separate Grid resources. Component migration can be provided by the framework as a mechanism to deal with variable resource availabilities, and distributed checkpointing and restart can be used to provide basic fault tolerance and recovery from failures of Grid resources.

To justify the above statement with a proof-of-concept implementation, we present XCAT3 [40], a framework for CCA-compatible components consistent with current Grid standards, and address the above claims using the same. XCAT3 is preceded by its earlier generations CCAT [9], and XCAT2 [34]. CCAT is our first implementation of a framework for distributed CCA-based components, and uses Nexus [32] as a means for component communication. XCAT2 is the second generation that uses SOAP [20] for component communications, and is consistent with Web service standards.

The key contributions of this dissertation are summarized in the following subsections.

1.2.1 A CCA Framework for the Grid

We have seen that Web and Grid service technologies enable orchestration of long running applications using Workflow tools, which we refer to as composition in time. On the other hand, standard component technologies viz. CCA, CCM, etc. enable creation of distributed applications using direct component connections, which we call composition in space. It follows that long running distributed applications on the Grid could be created effectively by a combination of the above approaches. XCAT3 is framework where CCA-based components are implemented in such a way that they are consistent with current Grid standards. Composition in time is provided by Workflow tools and Jython scripts, while composition in space is provided by CCA-defined constructs.

Mapping CCA components into OGSI-based Grid services is not trivial due to the semantic differences between the two specifications. We present a novel way to map CCA components as a set of Grid services, such that they can exploit all the desirable features presented by Grid service standards such as multiple-level naming, dynamic service introspection, and interoperability via standard protocols and interfaces.

1.2.2 Component Migration

Scheduling long-running applications is a difficult task on the Grid, since resource availabilities can be unpredictable over long periods of time. Rather than trying to schedule a long-running application in advance, a more proactive approach may prove to be more effective in such cases. Components can be scheduled on the best available machines, and migrated to better resources as and when they become available. Migration can also be triggered by violations of policies specified by the component writers and/or Grid resource owners.

There are several other systems that provide migration for applications on the Grid, e.g Condor [8]. However, most of them deal with single-process (non-distributed) applications. In XCAT3, a distributed application is composed of a set of components that may be executing on Grid resources that are geographically distributed. Migrating a component in this case is more complicated because it may be communicating with other components that are part of the application. Hence, we need a mechanism to migrate components in the presence of communication between them.

We use an approach where the user implements APIs to generate and re-load component state during migration, while the framework implements algorithms that transfer this state to/from stable storage, and migrate the component transparently in the presence of connections. We choose a user-defined mechanism for component persistence primarily for reasons of portability and efficiency.

1.2.3 Distributed Checkpointing & Restart

Maintaining reliability of heterogeneous geographically distributed Grid resources is a non-trivial task, especially since they may belong to different administrative domains. Reliability of resources is especially critical for long-running distributed applications, since the probability of failure increases with the number of resources and the length of the computation. In such a scenario, an ability to checkpoint applications and restart them from a particular checkpoint upon failure of Grid resources is desirable. Additionally, checkpointing can also be used for dynamic resource scheduling, e.g. if a high priority application needs to preempt a long running lower priority one, the lower priority application can be checkpointed and restarted later when the higher priority application has finished executing.

Distributed checkpointing and restart has been the topic of a lot of work over the past twenty years. The theoretical background for the same is very solid, and has been shown to work for various systems. However, the systems for which these have to be used enforce certain modifications and enable some simplifications. Hence, we survey the existing theoretical work and refashion it to work within the context of CCA and Grid services. Our approach is that the framework first ensures that the distributed set of components can produce a *consistent global checkpoint*. Subsequently, it triggers the checkpointing by invoking APIs implemented by the individual components to store their states. During restart, the states of the individual components are re-loaded from the latest consistent global checkpoint.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 reviews the concepts and systems that are relevant to our work. In particular, it describes software components and relevant Grid specifications in detail. It also presents the existing algorithms and systems that provide checkpointing for applications within our domain of interest. It also describes systems that use migration of applications in order to deal with dynamic Grid environments.

Chapter 3 describes the XCAT3 framework in great detail. Specifically, it presents the architecture of the framework and its conformance with the CCA and OGSI specifications, and how it can be used to orchestrate complex distributed applications on the Grid via composition of components in space and time. It concludes with a description of some applications that have been implemented within the framework.

Chapter 4 introduces the concept of persistence for components within the XCAT3 framework, and uses the same to present component migration as a mechanism for dealing with dynamic resource availabilities and policy violations. It also describes the algorithms and programming APIs required for component migration.

Chapter 5 presents the distributed checkpointing capability provided by the XCAT3 framework as a building block for fault tolerant long running applications. It discusses the algorithms implemented for distributed checkpointing and restart of XCAT3 components, and the programming APIs required for the same.

Chapter 6 introduces an application implemented to avail of the distributed checkpoint/restart, and migration capabilities of the XCAT3 framework. It also analyzes the performance of distributed checkpointing and component migration for the same application.

Chapter 7 presents the conclusions for this dissertation, and outlines ideas for future research.

2

Background and Related Work

In this chapter, we present the concepts and systems that are related to our work. First we present the software systems that are being used in scientific computing and the Grid, and also discuss some other similar systems that are being used in the business community. Next we present some literature on providing persistence for applications. We describe the implications of distributed applications to the process of checkpoint and restart, and discuss the pros and cons of different software techniques that can be used for capturing application state. We also present examples of some systems that provide this capability for distributed and non-distributed applications.

2.1 Software Components

Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system [53]. Component software enables practical reuse of software units, and promotes modularity of requirements, architectures, designs, and implementations. It encourages the move from the current huge monolithic systems to modular structures that offer the benefits of enhanced adaptability, scalability, and maintainability. Component software thus promotes rapid application development by combining well-tested and newly developed software parts.

In this dissertation, we will generally assume the following characteristics for software components, as suggested in [13]:

Independently deployable: A component can depend on other components for correct operation, but it must be possible to install and upgrade it individually.

Reuseable by third-parties: As implied by the definition, a component must be reusable in different applications, in order to enable rapid prototyping of applications.

Compose-able by third-parties: End users should be able to connect components deployed by others. This process is called component *composition* or *assembly*. This enables the end user to solve completely new problems from existing solutions.

Large Granularity: A component typically encapsulates more functionality than an object. Indeed most components will be implemented internally as a set of objects.

Inherently Distributed: A remote operation is an inherent characteristic of components. This means that the local operation is a special case of the normal operation.

Software components need several key services and rules in order to function as desired, which is provided by a *component architecture*. Component architectures define interactions between the components and their environment, the roles of the components, standard interfaces for tools, user-interfaces for end users and assemblers of components, and mechanisms for serialization of data, establishing connections, and managing lifetimes. A *component framework* is a software entity that supports components conforming to certain standard architectures, and allows instances of these components to be 'plugged' in. The framework establishes environmental conditions for the component instances and regulates the interactions between them. Thus a component framework can be thought of as a dedicated and focussed architecture, usually around a few key mechanisms, and a fixed set of policies for these mechanisms at the component level.

Software components have been used successfully for corporate enterprise, internet and desktop based applications in the industry. Recently, it is being adopted in scientific computing, as well as the Grid. The Open Grid Services Architecture (OGSA) and Infrastructure (OGSI) define a Web-services based component model for distributed systems integration on the Grid. In the following subsections, we present several component architectures that are relevant to our work. We begin with traditional component architectures used in business and scientific computing, and then present the ones used on the Grid.

2.1.1 Traditional Component Architectures

Before component architectures were adopted by the Grid community, they were successfully used in other domains. Indeed, the component architectures prevalent in the Grid community draw from other standard component architectures used in enterprise and scientific computing.

2.1.1.1 CORBA Component Model (CCM)

The Object Management Group (OMG) has defined CORBA as a distributed object architecture that is platform and language independent. The CORBA Component Model (CCM) defines a component model that is built on CORBA technology. It defines the process of designing, developing, packaging, deploying, and executing distributed heterogeneous components.

CCM defines the concept of *Ports* for components. Ports define interfaces that are provided or used by a component. Four different types of ports are defined by the CCM.

- *Facets*, which are interfaces provided by a component and used synchronously by clients.
- *Receptacles*, which are interfaces synchronously used by components.
- *Event sinks*, which are interfaces provided by a component and asynchronously used by clients.

- *Event sources*, which are interfaces asynchronously used by components.

These ports are used at deployment time and runtime to connect components together to create complex distributed applications.

Every CCM component has a *Home* object that is responsible for lifetime management, and is deployed within a container. CCM also defines an XML-based language for packaging and deployment of components.

2.1.1.2 Enterprise Java Beans

The Enterprise Java Beans (EJB) architecture is a server-side component architecture for the development and deployment of object-oriented distributed enterprise-level applications. The EJB architecture is three-tiered with the presentation logic in the first layer, the business logic in the second tier, and other resources such as the database back-end on the third tier. The EJBs are components in the second tier implementing the business logic. Communication with EJBs is through Java Remote Method Invocation (RMI).

EJBs can be implemented by defining the following two interfaces and two classes:

- *Remote Interface*, which defines the Bean's business methods.
- *Home Interface*, which defines the lifecycle methods for creation and removal of beans.

- *Bean class*, which implements the bean's business methods. The Beans can be stateful (Entity beans) or stateless (Session beans).
- *Primary Key*, that provides a pointer into a database. This is used only by Entity beans whose state have to be stored in the database.

The clients never interact with the bean class directly. Instead, they always use the methods defined in the Remote and Home interfaces. Every bean exists inside a *container* which is responsible for creation of new instances, storing state in databases, and other management aspects. The container invokes callback methods on the bean instance when appropriate state management events occur.

Unlike CCM, EJB does not define ports for direct connections between components. This is because EJB was designed to be a server-side component architecture, and every EJB component was envisioned to be a self-contained entity without any dependencies on the outside world. An application developer can still provide this capability, but it would be application-specific.

2.1.1.3 Distributed Component Object Model

The Distributed Component Object Model (DCOM) architecture is Microsoft's distributed component architecture that is robust and comprehensive, and enables tight coupling between the application and the operating system. The Component Object Model (COM) defines how components and their clients interact on the same machine. DCOM extends COM by using a standard network protocol based on DCE RPC if the interprocess communication needs to be across different machines.

A DCOM object supports a set of interfaces which other DCOM objects may use. The interfaces are defined using the Microsoft Interface Definition Language (MIDL) which is based on DCE IDL. The definitions are compiled to create *proxies* and *stubs*. When a remote call is made, the proxy is responsible for marshalling the parameters and sending the call across the network. On the remote side, the stub is responsible for unmarshalling the parameters, making the local call on the correct DCOM object, and sending the results back to the proxy. Every interface in DCOM extends from the IUnknown interface, which forms the root of the interface hierarchy. The IUnknown interface can also be used to query the component about what other interfaces it supports.

DCOM also supports the notion of connections between objects. These connections can be made by a third party, and can be changed dynamically at run-time. DCOM objects are deployed in binary packages called components, which also contain code to manage lifetimes and registration.

DCOM has several benefits such as robustness, location independence, language neutrality, effective connection management, and an ability to scale. However, its proprietary nature limits its portability and general applicability.

2.1.1.4 Common Component Architecture

The Common Component Architecture (CCA) is a project sponsored by DOE with inputs from several national labs and universities. The goal of CCA is to specify a component architecture for high performance computing where the target architectures include workstations, distributed memory multiprocessors, clusters of symmetric multiprocessors, and remote resources.

CCA uses the Scientific Interface Definition Language (SIDL) [17] to describe component interfaces. Like CORBA IDL, SIDL is programming-language neutral. However, it provides support for describing complex scientific data structures, e.g multi-dimensional arrays.

Like CCM, CCA uses the concept of *ports* to define the communication model for all component interactions. Components export functionality to other components via *provides ports*, while other components avail of this functionality via *uses ports*. Communication links between components are implemented by connecting uses ports to compatible provides ports. Apart from the simple use of provides and uses ports, CCA also defines the concept of *collective ports* in order to handle interactions among parallel components. Unlike other component models, CCA allows addition of ports dynamically at runtime. This ability makes it a suitable choice in the creation of Problem Solving Environments (PSE), in which the end-user directly manipulates component connections to solve the particular problem at hand.

A CCA component interacts with the framework via a *Services* object. The Services object contains information about provides and uses ports registered by a component, and their connections. It also contains a *ComponentID* which can be used to uniquely identify a component. CCA also defines a set of framework services that are available to an end-user to create instances of components, and compose them together.

At present, CCA only defines the source level interactions between a component and the framework. Interoperability between two components existing in different frameworks is not currently addressed.

2.1.2 Component Architectures for the Grid

A key requirement for the Grid is interoperability between various services located in different administrative domains. Until recently, different Grid projects such as Legion [42], Globus [41], Condor [8], etc. provided their functionalities in their own customary manner. Needless to say, there was hardly any interoperability between the available software tools. The Global Grid Forum [1] was then created as the organization responsible for coming up with a standard mechanism to create and access services on the Grid. The consensus was to use Web service technologies as a basis for a component model for the Grid.

2.1.2.1 Web Service Basics

The World Wide Web Consortium (W3C) defines a Web service as a software application identified by a Uniform Resource Identifier (URI), whose interfaces and bindings are capable of being defined, described, and discovered by XML artifacts and which can support direct interactions with other software applications using XML-based messages via Internet-based protocols. A more general and descriptive definition can be found in [35], where a Web service is defined as a *platform and implementation independent* software component that can be

- *described* using a service description language,
- *published* to a registry of services,
- *discovered* through standard mechanisms,
- *invoked* through a declared API, usually over a network, and
- *composed* with other services.

The primary goal of Web services is interoperability. A requestor can access a Web service by using *standard* well-defined mechanisms, irrespective of the language and the environment that either of them uses. This feature makes the Web services approach appealing to modern enterprise and inter-organizational computing systems.

The description of a Web service includes the supported interface, network, transport and packaging protocols. The Web Service Description Language (WSDL) [22] is a widely accepted standard for this purpose. The Resource Description Framework (RDF) [44] specification can also be used, though it is less popular than WSDL. The most widely recognized mechanism for publishing and discovery Web services is the Universal Description, Discovery, and Integration (UDDI) [2] specification.

Interaction between Web services is typically via the exchange of XML messages. The de facto standard for messaging in Web services is SOAP [20], which provides an XML-based messaging protocol between service providers and requestors. Most SOAP implementations use HTTP as the transport protocol since it is an internet standard. However, others such as SMTP, FTP, or BEEP [3] could also be used.

Web service *workflow* tools enable composition of a set of Web services into ones that are more useful. Workflow is defined as an organization of processes into a well-defined flow of operations, and can be thought of as the composition of services over time to accomplish a specific goal. The current standard for Web services workflow is the Business Process Execution Language for Web Services (BPEL) [23].

2.1.2.2 Open Grid Services Infrastructure

The Open Grid Services Architecture (OGSA) represents an evolution towards a Grid system architecture based on Web services concepts and technologies. OGSA envisions the Grid to be an integration of *Grid services* across distributed, heterogeneous, dynamic *virtual organizations*. It defines a Grid service to be a potentially transient stateful service instance that can support reliable and secure invocation, lifetime management, notification, policy management, credential management, and virtualization. At the core of OGSA is the Open Grid Services Infrastructure (OGSI) which defines a component model for Grid services to support the functionalities specified above.

The OGSI specification defines the following:

- A set of WSDL 1.1 extensions, viz. support for inheritance of port types, representation of *service data* (metadata and state data) associated with a service, etc.
- A set of operations for querying and updating the service data, and creation of destruction of a Grid service as part of the *GridService* port type.
- A Grid Service Handle (GSH), which is a URI serving as an immutable location-independent name for a Grid service, a Grid Service Reference (GSR), which is a precise description of how to access a Grid service across the network at any point in time, and operations for mapping a GSH to a GSR.
- Mechanisms for requesting asynchronous notifications of changes to the service data elements associated with a Grid service.
- A base format for fault messages, which is conformant with the WSDL fault message model.

The Grid community has raised several concerns against the OGSI specification. One concern that most people have is that it is very monolithic and not very suitable for incremental adoption. Another is that it does not work well with existing Web services tooling because of the extensions made to WSDL 1.1, and the use of features of WSDL 2.0 (e.g port type inheritance) which have still not been officially adopted.

However, several different implementations of OGSI do exist. One of the most popular ones is the Globus Toolkit 3.x from Argonne National Labs. A lightweight implementation of OGSI which we use in our work is GSX [27].

2.1.2.3 Web Service Resource Framework

The Web Service Resource Framework (WSRF) represents the refactoring of the functionality of OGSI to produce a framework of independently useful Web service standards, and the alignment of the functionality of OGSI with current and emerging Web service standards, such as WS-Addressing [19].

WSRF suggests a WS-Resource [24] approach for modeling stateful Web services. A WS-Resource is defined as a composition of a Web service and a stateful resource that is (i) expressed as an association of an XML document with a defined type with a Web service portType, and (ii) addressed and accessed according to the implied resource pattern, which is the conventional use of WS-Addressing endpoint references (EPR). In the implied resource pattern, a stateful resource identifier is encapsulated in an EPR and used to identify a stateful resource to be used in the execution of a Web service message exchange.

WSRF allows WS-Resources to be declared, created, accessed, monitored for change, and destroyed via conventional Web service mechanisms, but does not require that the Web service component of a WS-Resource that provides access to the associated stateful resources be implemented as a stateful message processor [25].

WSRF factors the OGSI specification into five independent specifications that define the normative description of the WS-Resource approach. These are:

- **WS-ResourceProperties**, which defines a WS-Resource, and mechanisms for retrieving, changing, and deleting WS-Resource properties.
- **WS-ResourceLifetime**. which defines mechanisms for destruction, and extension of lifetimes for WS-Resources.
- **WS-RenewableReferences**, which defines mechanisms needed to retrieve upgraded versions of EPRs when needed.
- **WS-ServiceGroup**, which defines interfaces needed to access collections of Web services.
- **WS-BaseFaults**, which defines the base format for faults in a Web service message exchange.

Publish-subscribe systems such as WS-Notification [4] are built on top of WSRF, rather than part of it. This enables individual specifications to be used separately, thus encouraging incremental adoption of the specification.

At the time of writing this dissertation, there are no stable implementations of WSRF available for public use.

2.1.2.4 Discussion

The use of Web service technologies for the Grid goes a long way in solving problems of interoperability. However, Web services also offer a server-centric model of computing. Web services are envisioned as self-contained entities exposed to the outside world. Hence, none of the component models for the Grid provide ports for connections between Grid services, as provided by CCA and CCM. This implies that Grid services can also be composed in only one dimension, viz. time with the use of workflow tools. Recent Web services publish-subscribe specifications help alleviate this problem a little bit by providing asynchronous communication mechanisms between services. However, synchronous mechanisms required for direct communication between Grid services is still lacking. Also lacking is an ability for third-party composition of Grid services.

2.2 Checkpointing Applications

Over the history of computer science, the topic of checkpointing has seen lots of research and development in industry as well as academia. Some of the reasons that have motivated checkpointing of applications in various systems are as follows:

Fault Tolerance: With the ubiquitous availability of large supercomputers and Grid resources, increasing amounts of computational power has been made available to the end user. However, since most of the individual resources are built from commodity hardware, the reliability of the whole system decreases proportional to the total number of individual resources. In such a scenario, checkpointing and rollback recovery provides an effective technique for tolerating transient resource failures, and for avoiding total loss of results.

Dynamic Resource Adaptation: Scheduling long running applications is a challenging task since it is difficult to predict resource availabilities into the future. Additionally, Grid systems span multiple administrative domains which may have their own resource policies, viz. local jobs must get precedence over Grid jobs, Grid jobs may only run for a particular length of time, etc. A combination of such policies, dynamic resource availabilities, and similar policies for applications can necessitate migration of jobs from one resource to another during execution. In such a situation, the state of the application can be checkpointed and the application can be physically migrated to a resource that satisfies both resource and application policies.

Logging: It might be beneficial to periodically dump the state of an application for post-processing analysis or data-visualization purposes. Restarting an application from certain checkpoints can also be beneficial for debugging purposes.

2.2.1 Distributed Checkpointing

Checkpointing distributed applications is more complicated than checkpointing the ones which are not distributed. When an application is distributed, the checkpointing algorithm not only has to capture the state of all individual processes, but it also has to capture the state of all the communication channels effectively. In other words, if a process records a message as sent, then it has to be either recorded as received by a corresponding recipient process, or the message should be accounted for in the state of the communication channel between the two processes. Hence, a *consistent global state* is defined as one that occurs in a failure-free, correct execution, without loss of any messages sent from one process to another [11]. A *consistent global checkpoint* is a set of individual checkpoints that constitute a consistent global state. A consistent global checkpoint is required to restart execution of a distributed application correctly upon failure.

Distributed checkpointing can be broadly divided into two categories: uncoordinated, and coordinated. A detailed exposition of these checkpointing techniques can be found in [18]. We briefly summarize 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, e.g. when the amount of state to be stored is very small. The disadvantages are that (1) there is a possibility of a *domino effect* which can cause the system to rollback to the beginning of the computation, thus negating the very advantage of checkpointing, (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, all communication between the processes is blocked. Subsequently, individual checkpoints (which now trivially constitute a consistent global checkpoint since all communication channels are empty) are taken. All communication between the processes can now resume. The primary disadvantage of blocking algorithms is the large latency involved in storing the checkpoints. However, these have been successfully used in several systems such as LAM-MPI [49].

As the name suggests, non-blocking algorithms do not block communication between the processes during checkpointing. They typically use a communication-induced approach where each process is forced to take a checkpoint based on protocol-related information piggybacked on the application messages it receives from other processes [11]. The receiver of each application message uses the piggybacked information to determine if it has to take a checkpoint. The checkpoint has to be taken before the application may process the contents of the message, possibly incurring high overheads. One disadvantage of this approach is that it is typically implemented at the messaging layer, and is tightly coupled with it. Hence the modified messaging implementation might not interoperate with other vanilla implementations (without checkpoint support).

2.2.2 Software Techniques

In practice, checkpointing for applications is implemented either at **system-level** or is **user-defined**. 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. The approach is not transparent to the user, but is more flexible due the same reason. While a detailed comparison between the two approaches can be found in [50], we summarize some of the key differences:

Transparency: In user-defined checkpointing the programmer is responsible for specifying what data should be included in the checkpoint, and where the checkpoints could be taken within the application code. On the other hand, system-level checkpointing is transparent to the user and requires little or no programmer effort.

Portability: Transparent system-level checkpointing has proven to be hardly portable across heterogeneous architectures. However, user-driven checkpoints are quite portable since the user can store the checkpoints in a platform-independent format.

Checkpoint size: System-level checkpointing is oblivious to the details of the application. Typically large amounts of unnecessary data may be stored since it is not possible to determine the critical state of the application. This leads to large checkpoint sizes, and large performance overheads for the checkpoint operation. However, user-defined checkpointing relies on user support to store only the minimal checkpoint required for restart, consequently reducing the checkpoint size and the performance overhead.

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 job swapping across different platforms, post-processing analysis, and visualization.

In summary, although system-level checkpointing is transparent to the user and easy to use, it is less portable and flexible, and creates larger checkpoints as compared to user-defined checkpointing. In the past, most of the checkpoint schemes were supposed to be transparent to the application and implemented at the system level. More recently, user-level schemes have been explored in greater detail.

2.2.3 Example Systems

We now describe some software systems that provide checkpointing for applications in the context of our discussions above.

2.2.3.1 Condor

Condor [8] is a distributed batch processing system developed at the University of Wisconsin, that aims at delivering large amounts of processing capacity to consumers over long periods of time by exploiting existing computing resources on the network. The goal of the system is to provide a high throughput by scheduling jobs on idle workstations. However, it is paramount that the owner of a workstation does not pay a penalty for adding his or her workstation to the pool of Condor workstations. Hence, a job must have the ability to immediately vacate a workstation if policies specified by the owner are violated (e.g a job can be run only when the owner is not using the workstation). This calls for an ability to migrate to another idle workstation or queue until one becomes idle.

In Condor, each customer is represented by a customer agent, which manages a queue of application descriptions and sends resource requests to the *Matchmaker*. Each resource is represented by a resource agent, which implements the policies of the resource owner and sends resource offers to the Matchmaker. The resource requests and offers are represented as *ClassAds*, which are analogous to classified advertisements in newspapers. The Matchmaker is responsible for finding a match between a resource request and a resource offer.

If and when any policies are violated during the execution of a job, Condor has the ability to migrate a job to another location. It does so by checkpointing the state of the process, and then restarting the process on another machine with the same image. The checkpointing is transparent to the user, and is implemented at the user level with no modifications to the kernel code. The checkpointing process is invoked by a signal, and at restart time, things are set up such that it appears to the user code that the process has just returned from the signal handler. The code contained in the signal handler, the code required to install the handler, and the code to record information as the state of the process changes, are all contained in the Condor checkpointing library.

The biggest advantage of using Condor for checkpoint and migration is that it is transparent to the user. The user only has to re-link his or her code with the Condor checkpointing libraries. This works fine for users who have access to the software, but can be a hindrance to users of third party software. However, there are several disadvantages with Condor's approach to checkpoint and migration. The process image for a job may be huge, and a lot of unnecessary information could be stored if checkpointing is done at the process level. Process images are also highly dependent on the architectures, and hence can not be migrated across heterogeneous platforms. It also does not handle migration of a set of communicating processes (using signals, sockets, pipes, files, or any other means). Despite these shortcomings, a wide variety of real-world user code can be accommodated by this approach.

2.2.3.2 CUMULVS

CUMULVS [39] is a middleware infrastructure developed at the Oak Ridge National Lab (ORNL) for interacting with parallel scientific simulation programs, and supports online visualization and computational steering. It also provides a user-level mechanism that assists in creation of checkpoints, and restart from saved checkpoints.

The user application selects the minimal program state necessary to restart or migrate an application task. Application tasks can then be migrated across heterogeneous architectures to achieve load-balancing or to improve a task's locality with a required resource. CUMULVS is suitable for several scientific applications which need checkpointing only at a coarse-grained level.

CUMULVS benefits from the advantages and disadvantages of user-defined checkpointing described in Section 2.2.2 - greater portability and flexibility, smaller checkpoints, and a greater level of programmer effort.

CUMULVS does not do anything special in order to handle distributed applications.

Instead, it relies on the application writer to create a globally consistent checkpoint. This increases the expertise required on the part of the application writer.

2.2.3.3 LAM-MPI

Possibly the maximum amount of work in implementing distributed checkpointing has been in the context of parallel MPI or PVM based applications [52], [43], [49]. We illustrate this using one such implementation.

The LAM implementation of MPI developed at Indiana University provides co-ordinated checkpointing and recovery for MPI-based parallel applications. It uses a kernel-level process checkpointing system to transparently checkpoint a parallel program. The checkpointing and restart capability can be used for fault tolerance in case of failures of nodes, and also for re-scheduling an application for performance reasons.

A coordinated, blocking algorithm is used in LAM-MPI to capture a consistent global checkpoint for a set of MPI processes. Presently, the focus of LAM-MPI is to checkpoint and restart the complete set of processes upon failure. Migration of individual processes is not handled.

Like Condor is for single-process jobs, the transparent checkpoint capability of LAM-MPI for parallel processes hides the complexity of the system from the user. However, unlike Condor, the checkpointing is done at the kernel level. This assumes that the target machines provide checkpointing at the kernel level. However, the codes need not be re-compiled or re-linked to avail of this functionality. Similar to Condor, the process images may be huge as a lot of unnecessary state might be saved.

2.2.3.4 Mobile Agents

Mobile agents [12] are programs that can migrate from host to host in a network. The state of a running program is saved, transported to another host, and restored allowing the program to continue where it left off. Mobile agents differ from typical process-migration systems in the sense that they migrate of their own choice, whereas in a process-migration system the system decides when and where to move the running process. Mobile agents also differ from other types of mobile code such as *applets*, which are programs that are downloaded to a remote location, then executed from beginning to end.

Mobile agents are a good choice for many applications as they improve latency and bandwidth of client-server applications and reduce vulnerability to network disconnection. They are very applicable to mobile devices and mobile users.

However, they do have several technical hurdles. Current systems can save on network latency and bandwidth at the expense of higher load on the service machines, since the agents are often written in a relatively slow interpreted language for portability and security reasons. Additionally, the goal of mobile agent systems is to allow a program to move freely across heterogeneous architectures. Hence, the code has to be compiled into some platform independent representation (such as Java bytecode). Another important concern in a mobile agent system is security. Agent code should be protected from malicious machines, and vice versa. This is a currently a topic of open research.

2.2.3.5 Cactus

Cactus [37] was originally developed as a framework for the numerical solution of Einstein Equations. However, it has evolved into a general-purpose, open source, problem solving environment that provides a unified modular and parallel computational framework for scientists and engineers. Cactus has a central core (or *flesh*) that connects to application modules (or *thorns*) through an extensible interface. Thorns can implement custom-developed scientific or engineering capabilities, as well as other computational capabilities, such as data distribution and checkpointing.

Cactus uses the concept of Performance Contracts [55] to define an agreement between a user and the provider of one or more resources. If a contract is found to be violated, the Migration Logic Manager thorn instructs Cactus to checkpoint the application state to stable storage. The Migration Module thorn communicates with the external Migrator service, informing it of the current simulation location and the new target host and tells it to restart the application on the new resources. The Migrator service stages all files to the new location, and restarts the application from the saved checkpoint on the new resource where the performance contracts are satisfied [6].

The checkpointing is done at the user-level, and is hence architecturally independent and has all the pros and cons of user-level checkpointing. There is no explicit support for distributed applications. However, it is conceivable that a new thorn could be added in order to create a globally consistent checkpoint for distributed applications.

2.2.3.6 Component Persistence

Some component architectures defined in Section 2.1.1 provide mechanisms to load and store states into persistent storage. We discuss the ones provided by Enterprise Java Beans and CORBA Component Model.

Persistence of EJBs can be either container-managed or bean-managed, which are analogous to system-level and user-defined checkpointing techniques respectively, as defined in Section 2.2.2. Container-managed persistence is simplest to develop as it delegates the responsibility of persistence to the EJB container. When the bean is deployed, the fields that need to be stored are identified, and a mapping into the database is defined. The container generates the logic necessary to store the bean's state automatically. In addition, the container makes callbacks to the bean so that it can complete any clean up or preprocessing before a store or a load respectively. Bean-managed persistence is a little more complicated as the persistence logic is the responsibility of the programmer. In addition, it is not as database-independent as container-managed persistence. However, it provides more flexibility on how state is managed between the bean and the database, and can better accommodate complex and unusual set of data.

Persistence of CCM components is similar to that of EJBs. CCM allows either self-managed or container-managed persistence, which are analogous to EJB's bean-managed and container-managed persistence respectively. CCM uses the OMG Persistent State Definition Language (PSDL) to define persistent state, and a Persistent State Service (PSS) generates code that stores and retrieves state values when needed. The PSS may be implemented on top of either a relational or object database, or may store data in a flat file.

Most existing component persistence mechanisms only handle storage of data, and not of the execution stack. In addition, there is no formal notion of a consistent global state which is required for distributed applications.

2.2.3.7 Discussion

As is evident from the above description, persistence for applications has been provided by several software systems. However, the solutions are not general-purpose and are specific to the particular software systems and programming models in which they are implemented. Distributed checkpointing software does exist, e.g. for parallel MPI and PVM based applications. However, there is hardly any consensus or available software for distributed checkpointing using other paradigms, e.g the ones that are component or Web services based, especially on the Grid.

Recently, the Grid Checkpointing and Recovery (GridCPR) [28] 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, they have only concentrated on checkpointing and restart of single process jobs. Although this is a good first step, APIs and protocols for capturing consistent global checkpoints are imperative in order to handle complex Grid applications.

3

Distributed Components on the Grid

In this chapter, we present a detailed description of the XCAT3 framework, which forms a basis for all the work introduced in this dissertation. We first present the design goals for XCAT3, and then discuss its architecture in detail. We also describe how components can be written and used within the framework, and conclude with a sample application that has been implemented using our framework.

3.1 Design Goals

The primary functional goal of XCAT3 is to provide an ability to compose complex distributed applications on the Grid. We also wish to provide mechanisms inside XCAT3 to support checkpoint and restart for distributed components for fault tolerance, and migration of individual components for adaptability to dynamic Grid environments. This leads to several architectural requirements, which we present in this section.

3.1.1 Composition of Complex Applications

We introduced the concept of composition in space and time in Chapter 1. In this subsection, we elucidate this further with an example, and present some of its implications.

Consider a situation where an engineering design team wishes to build a distributed application that starts with a database query which provides initialization information to a data analysis application, which in turn feeds the resulting data to a filtering service that provides the required results. When this is implemented using standard Grid and Web service technologies, a central workflow engine (e.g. one that supports BPEL [23]) intermediates at each step of the application sequence and relays the messages from one service to another. This approach is called *composition in time*, since a larger task is composed by a set of smaller tasks executed sequentially over a period of time. This is shown in Figure 3.1.

The reason why the above approach is necessitated is twofold. First, philosophically, Web services are self-contained entities with no external dependencies. Hence, they are designed to provide certain operations independently, and are not meant to be connected directly to other Web services. Second, in practice, no official bindings are defined for *outgoing* Web service operations, i.e. the operations that the Web services need to invoke on

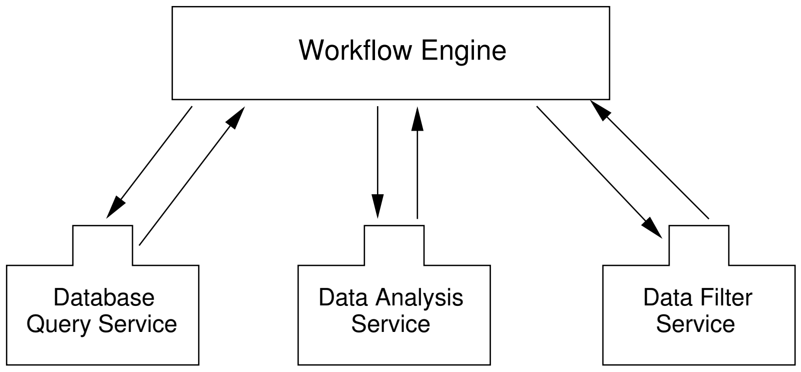

Figure 3.1: Standard composition of Web services in time via a workflow engine.

other Web services in order to accomplish their task. This is despite the fact that WSDL 1.1 provides a mechanism to describe these outgoing operations, viz. solicit-response (output-input), and notification (output-only). Hence, outgoing operations of Web services are never used in practice. Consequently, outgoing operations of Web services can not be connected directly to corresponding incoming operations, necessitating central mediation of messages.

This approach has several disadvantages. If the data traffic between the services is heavy, it is best not to require it to go through a central workflow engine. Furthermore, if the logic that describes the interaction between the data analysis component and the filtering service is complex and depends on application behavior, then putting it in the high level workflow may not work since workflow specifications provide limited computational capabilities.

What we really need for truly distributed applications is an ability to compose the services together in such a way that we obviate the need for a central workflow engine. As

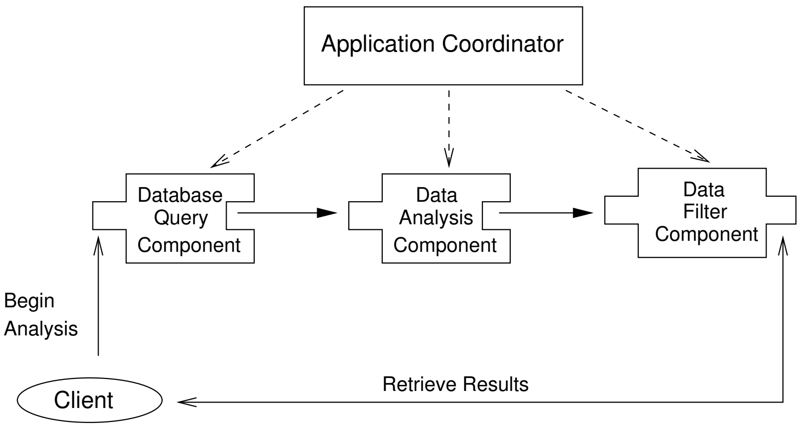

Figure 3.2: Composition of components in space via direct connections.

shown in Figure 3.2, we replace the workflow engine with an Application Coordinator (AC). The AC is only responsible for creating instances of the components if required, and connecting them together. We call this approach *composition in space*, since the components are distributed at different locations and are (possibly) executing concurrently.

Note that publish-subscribe systems may be used to provide asynchronous communication between Web services, thus providing limited composition in space. However, asynchronous communication is typically used for notification of state changes, and is not a very intuitive model for availing of functionalities provided by other Web services. Furthermore, publish-subscribe systems don't lend themselves very well to composition by third parties; the logic of the publisher and subscriber is deeply embedded within the Web services themselves.

The XCAT3 framework supports composition in space via its conformance to the CCA specification. As prescribed by CCA, components can provide services via *provides ports* and use services via *uses ports*. Composition in space can be accomplished by connecting the uses ports of components to compatible provides ports. Additionally, other publish-subscribe mechanisms can be added for asynchronous communication between components.

3.1.2 Conformance to Grid Standards

Since XCAT3 components execute on the Grid, it is desirable that they conform to Grid standards. At the beginning of development of XCAT3, OGSA and OGSI were the upcoming Grid standards. Conforming to OGSI presented the following benefits:

- Standardization of interfaces and protocols, thus enabling the components to be accessible by standard Grid clients.
- Multiple level naming, which is useful for checkpoint and restart, as well as for migration for components. Since a GSH is location-independent, a component can be easily located even if it is re-instantiated at another location.
- Leveraging useful Grid services (e.g. Registries) and specifications (e.g. BPEL) being developed within the Grid/Web community.

However, mapping CCA concepts to OGSI is not an entirely trivial task. CCA components and Grid services differ in their definitions of ports and port types. Within CCA, a component can have multiple ports of the same type. This is because ports are envisioned to be stateful, e.g. a microscope can have multiple electron guns of the same type, which can be represented as multiple instances of the same port type. However, in the Grid and Web services world, ports are stateless. Multiple bindings for the same port type may exist, but they are semantically equivalent, i.e. they can be used interchangeably. Hence, a mapping needs to be devised between CCA ports and their Grid service counterparts. Additionally, Grid services have no concept of a *ComponentID*, which is used by CCA to uniquely identify a component.

3.1.3 Other Grid Issues

Apart from conformance to CCA and OGSI, there are a few other architectural requirements that are necessitated by the distributed nature of the components.

- **Deployment & Instantiation:** Since the components will be executing on Grid resources, there has to be a mechanism to deploy these components on the same. Furthermore, once these components have been deployed, they need to be instantiated as required. This also calls for suitable authentication and authorization mechanisms.
- **Remote Invocations:** Since the components are distributed, there needs to be a Remote Method Invocation (RMI) mechanism to invoke methods across the Grid. Additionally, these invocations may need to be secure if sensitive data is being passed around.

3.2 Architecture of XCAT3

The architecture of XCAT3 reflects the design goals presented in the previous section. XCAT3 represents a framework that enables CCA-style distributed components to be accessed as Grid services. The prototype implementation of XCAT3 is in Java, although a C++ version is planned in the future.

3.2.1 CCA Compatibility

The CCA specification, which defines interfaces for components and framework services, is defined in the Scientific Interface Definition Language (SIDL). A framework that conforms to CCA provides implementations for the all the mandatory interfaces defined by the specification.

Babel [10] is the standard software toolkit that parses interface definitions in SIDL, and automatically generates appropriate glue code. This glue code mediates differences among calling languages and supports efficient inter-language calls within the same memory address space. We use Babel for code generation, and implement the interfaces in Java. However, at present, Babel does not have an ability to support remote procedure calls. Hence, we have to modify the generated code to replace the in-process calls with remote calls using the XSOAP [51] toolkit.

XSOAP (formerly called SoapRMI) is a lightweight implementation of Remote Method

Invocation (RMI) that uses SOAP [20] as the communication protocol. Since CCA only defines the source level interactions between the components and the framework, individual frameworks can choose the component communication mechanisms that work best for their cases. The choice of SOAP for XCAT3 was natural since it is the *lingua franca* for the Grid and Web service technologies.

Some of the key CCA interfaces implemented within the XCAT3 framework are:

- **Component:** Every component has to implement the `gov.cca.Component` interface. The only method that it contains is the `setServices`, which is used to initialize the *Services* object. The Services object is used by a component for interacting with the framework.
- **Services:** As mentioned above, every component contains a Services object which it uses to interact with the framework and other components that are part of it. The Services object is responsible for providing methods to register uses ports (`registerUsesPort`), add provides ports (`addProvidesPort`), fetch a previously registered port (`getPort`) so that remote methods can be invoked, release the said port (`releasePort`) to notify the framework that the port is no longer being used, etc.
- **ComponentID:** The Services object contains a ComponentID that can be used to

uniquely identify a component. The two operations that CCA defines are `getInstanceName` which returns the name of the component instance, and `getSerialization` which returns the unique framework-specific serialization of the ComponentID. However as we point out in Subsection 3.2.1.1, several additional methods have to be added for it to be useful in a distributed environment.

- **CCAException**: CCA defines a set of exceptions that may occur during execution, viz. PortNotDefined, PortAlreadyDefined, PortNotConnected, BadPortName, etc. XCAT3 creates a Java Exception class for every CCA defined exception. All exceptions extend from the base `gov.cca.CCAException`. All exceptions thrown during communication between components are caught and returned to the component that initiated the communication. The exceptions are mapped to *SOAP faults* on the wire and then to corresponding exceptions on receiving end of the initiating component.
- **Builder Service:** CCA defines a Builder service for creation of component instances, and composing them together. We describe the XCAT3 Builder service in detail in Subsection 3.2.1.1.

3.2.1.1 XCAT3 Builder Service

The Builder service is a standard CCA interface for standard component creation and connection. It defines methods for the same, but does not mandate how they are implemented.

Some of the key operations provided by the Builder service are:

- **createInstance:** Components can be instantiated using the createInstance operation. However, these components may have to be executed on remote Grid resources. Hence, XCAT3 provides an ability to launch remote instances using the standard Globus GRAM [15] protocol provided by the Java CoG kit [54]. On invocation of the createInstance operation, the Builder service requests a remote instantiation using GRAM, and blocks until it receives the serialized form of the ComponentID from the remote side. A ComponentID stub is created from the received information and relayed back to the user, who can use it to refer to the component instance. Apart from GRAM, the XCAT3 Builder service also supports component instantiation via local process executions, and ssh.
- **destroyInstance:** As the name suggests, this operation can be used to terminate execution of a component.
- **connect:** This operation can be used to connect a uses port of one component to a compatible provides port of another. In XCAT3, a connect call results in a *ConnectionID* object being stored within the Services object of the component using the connection. The ConnectionID object can be used to locate the component functioning as the provider, as well as the provides port that is part of the connection. In addition, a reference count is incremented for the provides port inside the Services object of the providing component. However, the CCA specification does not provide

standard operations for setting the ConnectionID and increasing the reference count. This is because CCA was originally designed for non-distributed applications and all connection information was supposed to be located centrally within the framework. With the components being distributed, it is necessary for the connection state to be stored in a distributed fashion for reasons of performance and scalability. Hence, we extend the ComponentID in XCAT3 to create the *XCATComponentID* interface which contains the required operations.

- **disconnect:** This operation enables a port connection to be broken. The ConnectionID object is removed from the using component, and the reference count is decremented for the providing component. This operation enables dynamically changing connection configurations to provide the flexibility desired in PSEs. Operations to support removal of the ConnectionID object from the using component, and decrementing the reference count for the providing component are added to the XCATComponentID interface.

For rapid prototyping purposes, XCAT3 provides an interface to the Builder service using Jython scripts. Jython is pure Java implementation of the Python scripting language, and provides an almost seamless interface to code written in Java. Hence, exposing the functionality of the Builder service which was originally written in Java via Jython was a trivial task. The Jython API provided to the user closely mirrors the API provided by the Builder service.

XCAT3 does not provide a capability to stage executables to remote Grid resources. Instead, it relies on components being installed and ready to use. However, it provides a mechanism to package a component within XML deployment descriptors, which contain information about how to instantiate the components. The descriptor contains the fully qualified class name for the component, environment variables viz. standard output and error, working directories, etc., the hosts on which the component may be deployed and the protocols supported by those hosts, the port types supported by the component, etc. The Jython scripting engine parses the descriptor to retrieve the deployment information, and uses the Builder service to instantiate the component.

3.2.2 OGSI Compatibility

OGSI compatibility for XCAT3 is accomplished using Grid Service Extensions (GSX) [27], which is a lightweight implementation of the OGSI specification built on top of the XSOAP toolkit. GSX adds the portTypes prescribed by the OGSI specification, the provision to add Service Data Elements (SDE) to Grid services, and the GSH/GSR-based multiple level naming to the XSOAP toolkit.

In order to make XCAT3 compatible with OGSI, all remote services are exposed as Grid services. This includes framework services such as the Builder service, as well as the components themselves. Additionally, the XCAT3 framework provides an implementation for the OGSI *Handle Resolver* service which is responsible for mapping a GSH for a Grid service to the most recent GSR that can be used to access it.

We note from Subsection 3.1.2 that CCA ports and Web service ports are not exactly equivalent due the state associated with the CCA ports. Consequently, a component can not be represented as a Grid service with every provides port being a Web service port. Hence, we model a component as a set of Grid services as follows.

Every provides port of a component is represented as a separate Grid service using GSX. Thus, every provides port has a location independent GSH, as well as a specific GSR that can be used to access it. The GSR is a WSDL document that the XSOAP toolkit generates, and can be used by any standard Grid client to interact with the provides port. Every uses port is a just a client-side stub, connected to either a provides port provided by another component, or to a standard OGSI-compliant Grid service. When the uses port is connected to a provides port or a Grid service, the ConnectionID object that is cached by the using component contains the GSH for the remote provider. This makes the connection information location independent as well, which means that the providing component or Grid service could migrate to another resource and still remain accessible as long as its GSR is updated with the Handle Resolver service.

Additionally, the ComponentID for every component is also exposed as a separate Grid service. It contains Service Data Elements containing the names, GSHs, and GSRs for all provides ports. Thus, any Grid client can obtain information about the ports supported by a component by querying its ComponentID interface. The `getSerialization` method

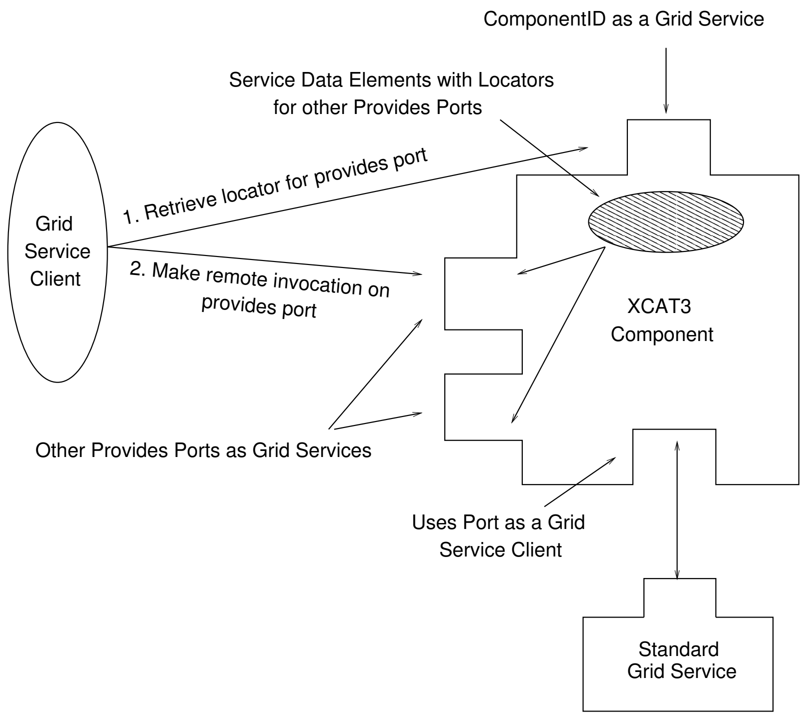
ComponentID as a Grid Service
Service Data Elements with Locators for other Provides Ports
1. Retrieve locator for provides port
Grid Service Client
2. Make remote invocation on provides port
XCAT3 Component
Other Provides Ports as Grid Services
Uses Port as a Grid Service Client
Standard Grid Service

Figure 3.3: An XCAT3 Component as a set of Grid services

of the ComponentID, which is supposed to return a unique framework specific serialization of the ComponentID, returns the GSH for the component. This GSH along with the Handle Resolver service can be used by a client-side stub to access the component as long as it is alive.

The OGSI specification uses a concept called *ServiceGroups* to group together a set of Grid services that are related. The ComponentID is analogous to an OGSI ServiceGroup in the sense that it groups together related services (provides ports) using Service Data Elements.

All the Grid services that are part of the component share the same *Component* object,

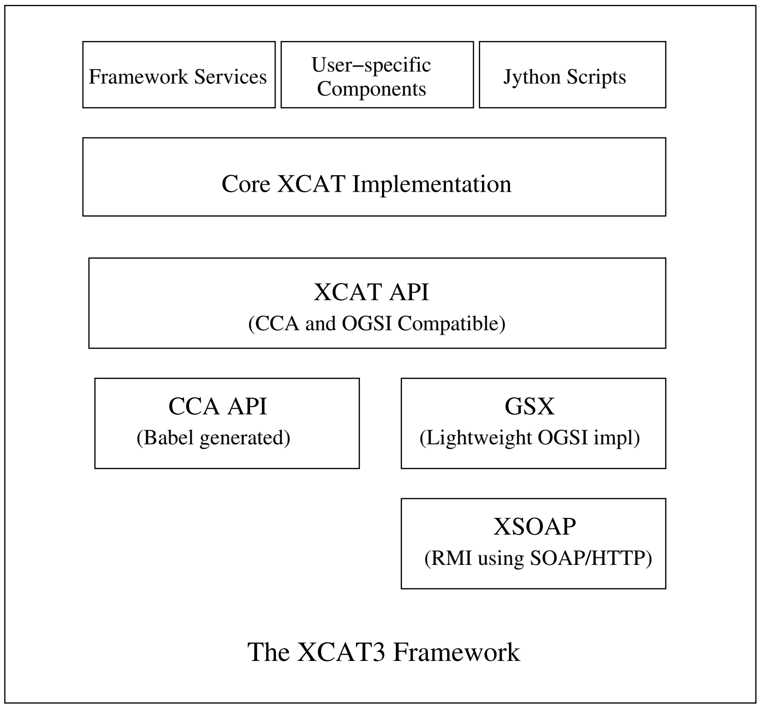

Figure 3.4: XCAT3 Software Stack

and can hence communicate with each other via mutual shared state. The modeling of an XCAT3 component as a set of Grid services is illustrated in Figure 3.3.

3.2.3 Summary

Thus, we have described the architecture of XCAT3 which provides a CCA framework consistent with Grid standards. The software stack is summarized in Figure 3.4.

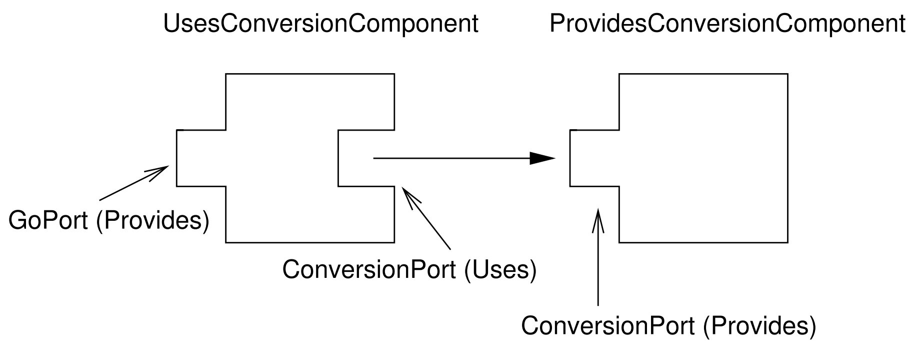

Figure 3.5: Simple uses-provides relation between components

3.3 Writing and Using XCAT3 Components

Implementing components in the XCAT3 framework consists of writing the port interfaces, implementing the ports, and writing the components themselves. Jython scripts, along with deployment descriptors for the components, can be used to subsequently compose them meaningfully. We illustrate this process in this section.

As an example, we present a system with two components - *ProvidesConversionComponent*, which provides a simple function to convert temperature from Celsius to Fahrenheit scales via a provides port *ConversionPort*, and *UsesConversionComponent*, which uses this functionality provided via a uses port. To begin execution, the UsesConversionComponent uses a *GoPort*, which is an especially designed CCA port for bootstrapping execution of set of a components. Figure 3.5 illustrates the relation between the components. Although this particular example is a trivial one, it depicts the steps involved in writing distributed components.

3.3.1 Port Interfaces

Ideally, the port interfaces should be defined in SIDL. However, since Babel currently does not support distributed objects and remote methods, port interfaces are simply defined in Java in XCAT3. Every port interface in XCAT3 extends from the *XCATPort* interface. The XCATPort interface extends the *Port* interface generated by Babel from the CCA SIDL specification, and also the *XSoapGridServiceInterface* provided by GSX for OGSI compatibility. The interface for *ConversionPort* in our simple example is shown below.

```
package samples.conversion;

import intf.ports.XCATPort;

/**
 * The interface definition for the Convert Port
 */
public interface ConversionPort extends XCATPort {

  /**
   * Method to convert a value in celsius to centigrade
   */
  public float centigradeToFahrenheit(float celcius);
}
```

3.3.2 Port Implementations

XCAT3 provides a basic implementation for a port via the *BasicPortImpl* class, which all port implementations extend from. This class implements the methods present in the

XCATPort interface that are inherited from the XSoapGridServiceInterface, so that the component writer need not be concerned with implementing OGSI-specific methods.

The implementation for the ConversionPort is shown below. Note that the component writer only implements the methods that are added via the definition of the port interface.

```
package samples.conversion;

import xcat.ports.BasicPortImpl;

/**
 * This is the implementation of the ConversionPort.
 */

public class ConversionPortImpl extends BasicPortImpl
  implements ConversionPort {

  /**
   * Default constructor.
   * This has to throw java.lang.Exception because the
   * superclass constructor throws it.
   */
  public ConversionPortImpl() throws Exception {
    super();
  }

  /**
   * Method to convert a value in celcius to centigrade,
   * which is the only method defined by the
   * ConversionPort
   */
  public float centigradeToFahrenheit(float celcius) {
    float fahr = ( ( 9f / 5f ) * celcius ) + 32;
    return fahr;
  }
}
```

The GoPort is also implemented in a similar fashion. It contains a `go` method, which calls the `startExecuting` method on the UsesConversionComponent to bootstrap execution.

3.3.3 Component Implementation

Every component has to implement the *Component* interface, and the `setServices` method inside it. Within the `setServices` method, a component is expected to create instances of provides ports and add them to the *Services* object using the `addProvidesPort` method, and also register uses ports using the `registerUsesPort` method. When a provides port is added, the XCAT3 *Services* implementation makes it available to the outside world as a Grid service transparently. We describe the implementation of the UsesConversionComponent below.

The basic outline for the UsesConversionComponent is shown in the following code snippet. Apart from a default constructor and the `setServices` method, the component implementation contains a single method, `startExecuting`. As mentioned earlier, this method is invoked by the `go` method of the GoPort in order to bootstrap execution.

```
package samples.conversion;

import gov.cca.Component;
import gov.cca.Services;
import gov.cca.TypeMap;

/**
 * The implementation of the UsesConversionComponent.
 */
public class UsesConversionComponent implements Component
  {

  // Every component contains an instance of the Services
  // object
  private Services usesCore;

  // Instance of a GoPort for this component to bootstrap
  // execution
  private GoPortImpl goPort;

  /**
   * Default empty constructor required for instantiation
   */
  public UsesConversionComponent() {
  }

  /**
   * The only method defined by the Component interface
   * @param cc the services object for this component
   */
  public void setServices(Services cc)
    throws gov.cca.CCAException {
    // setServices() implementation goes here
  }

  /**
   * This method represents whatever computation the
   * component needs to do. It is invoked by the
   * go method of the GoPort.
   */
  public void startExecuting() {
    // code to bootstrap component execution goes here
  }
}
```

The following code shows the part of the `setServices` method where the UsesConversionComponent registers a uses port in order to avail of the conversion service provided by the ProvidesConversionComponent. To register a uses port, the implementation provides (1) a name for the port that will be used later to grab a reference to it, (2) a unique namespace for identifying the type of the port, and (3) a *TypeMap* object, which is a CCA defined object for storing properties of a port. The classname for the port interface is stored as the *portClass* property within the TypeMap.

```
public void setServices(Services cc)
  throws gov.cca.CCAException {
  ...

  // signify the interface name for the uses port
  TypeMap uMap = usesCore.createTypeMap();
  uMap.putString("portClass",
                 ConversionPort.class.getName());

  // param[0] : name of port to register
  // param[1] : unique namespace for the port
  // param[2] : typeMap object for the port as defined above
  usesCore.registerUsesPort("convertUsesPort",
                            "http://foo.bar/conversion",
                            uMap);

  ...
}
```

The UsesConversionComponent adds a GoPort as a provides port inside the `setServices` method as follows. To add a provides port, the implementation provides (1) a reference to the provides port implementation, (2) a name for the provides port which can be used later to refer to it, (3) a unique namespace for identifying the type of the port, and (4) a TypeMap object for storing port properties. The classname of the port interface is stored as the *portClass* property within the TypeMap.

```
public void setServices(Services cc)
  throws gov.cca.CCAException {
  ...

  // create an instance of the GoPort
  goPort = new GoPortImpl(this);

  // signify the interface for the provides port
  TypeMap pMap = usesCore.createTypeMap();
  pMap.putString("portClass",
                 intf.ports.XCATGoPort.class.getName());

  // param[0] : the instance of the port to add
  // param[1] : name of port to register
  // param[2] : unique namespace for the port
  // param[3] : typeMap object for the port as defined above
  usesCore.addProvidesPort(goPort,
                           "providesGoPort",
                           "http://foo.bar/go",
                           pMap);

  ...
}
```

The `startExecuting` method of the UsesConversionComponent is shown below. The component grabs hold of the registered uses port via a `getPort` call. The use of the `getPort` call is twofold. Firstly, it signifies to the framework that the port is being used.

The framework can then make sure that the remote component that provides the service is always available when the port is being used. Secondly, it makes sure that no two threads in the same component are using the same uses port at the same time. If another thread has a reference to the uses port, the call blocks until the other thread releases it. On completion of the remote invocation, the component releases the port via a `releasePort` call. This signifies to the framework that the port is no longer being used, and enables another thread to use it, if need be.

```
  /**
   * This method represents whatever computation the
   * component needs to do. It is invoked by the
   * go method of the GoPort.
   */
  public void startExecuting() {

    // Get a reference to a usesConvert port so that
    // remote methods may be invoked
    ConversionPort usesPort = (ConversionPort)
      usesCore.getPort("convertUsesPort");

    // Make a call on the remote provides port
    System.out.println("UsesConversionComponent: " +
                       "Celcius : 0, Fahrenheit: " +
                       usesPort.centigradeToFahrenheit(0f));

    // Release the port when we are done using
    usesCore.releasePort("convertUsesPort");
  }
}
```

The implementation of the ProvidesConversionComponent is similar to the UsesConversionComponent shown above. The ProvidesConversionComponent adds a single provides port - the ConversionPort, which provides the remote service desired by the UsesConversionComponent. It does not need a GoPort since it is only a provider of a service which responds to requests from users.

3.3.4 Component Execution

As we mention earlier, XCAT3 provides an XML-based deployment mechanism for components. The XML descriptor for the UsesConversionComponent is shown below.

```
<componentStaticInformation>
 <componentInformation>
  <name>User of Temperature Conversion Service</name>
  <author>Sriram Krishnan</author>
  <portList>
   <usesPort>
    <portName>convertUsesPort</portName>
    <portType>http://foo.bar/conversion</portType>
   </usesPort>
   <providesPort>
    <portName>providesGoPort</portName>
    <portType>http://foo.bar/go</portType>
   </providesPort>
  </portList>
 </componentInformation>
 <executionEnv>
  <hostName>k2.extreme.indiana.edu</hostName>
  <hostName>rainier.extreme.indiana.edu</hostName>
  <creationProto>ssh</creationProto>
  <creationProto>gram</creationProto>
  <creationProto>local</creationProto>
  <nameValuePair>
   <name>className</name>
   <value>samples.conversion.UsesConversionComponent</value>
  </nameValuePair>
  <nameValuePair>
   <name>execDir</name>
   <value>/u/srikrish/Projects/xcat3/src/java/scripts</value>
  </nameValuePair>
  <nameValuePair>
   <name>execName</name>
   <value>ContainerLauncher.sh</value>
  </nameValuePair>
  <nameValuePair>
   <name>stdOut</name>
   <value>/u/srikrish/Projects/xcat3/user.out</value>
  </nameValuePair>
  <nameValuePair>
   <name>stdErr</name>
   <value>/u/srikrish/Projects/xcat3/user.err</value>
  </nameValuePair>
 </executionEnv>
</componentStaticInformation>
```

The descriptor contains information describing the component itself within the *componentInformation* block, viz. the *name*, *author*, and *portList*. The portList contains a list of uses ports inside the *usesPort* block and a list of provides ports inside the *providesPort* block.

The descriptor also contains the execution environment for the component inside the *executionEnv* block. The executionEnv provides the resources the components can be instantiated on (list of *hostName* elements), and the protocols that can be used to instantiate them (list of *creationProto* elements). The rest of the execution environment is a array of name-value pairs which contain enough information to instantiate a component on a particular resource. Some of the possible names are *className*, which represents the fully qualified classname for the component, *execDir*, which represents the directory where the component should be instantiated, and so on.

Jython scripts can be used to instantiate the components, connect the respective ports, and bootstrap execution by invoking the `go` method on the GoPort of the UsesConversion-Component. A script that does the same is shown below.

```
import cca

# code to read component XML descriptors
...

# create component wrappers
provides = cca.createComponentWrapper("provider",
                                      providerXML)
uses = cca.createComponentWrapper("user",
                                  userXML)

# assign a machine name
cca.setMachineName(uses,
                   "k2.extreme.indiana.edu")
cca.setMachineName(provides,
                   "rainier.extreme.indiana.edu")

# set a creation mechanism to Globus GRAM
cca.setCreationMechanism(uses,
                         "gram")
cca.setCreationMechanism(provides,
                         "gram")
```

```
# create live instances
cca.createInstance(uses)
cca.createInstance(provides)

# connect their ports
cca.connectPorts(uses,
                 "convertUsesPort",
                 provides,
                 "convertProvidesPort")

# bootstrap execution of components
cca.go(uses)
```

3.3.5 Application Factories

Although it may be necessary to build some components from scratch occasionally, the software engineering benefits of components are realized effectively when well-tested, pre-packaged components can be re-used by third parties. We use Application Factories for building reliable Grid applications by separating the process of deployment and hosting from application execution [33].

As we have seen earlier, a distributed application is composed of a set of components. The deployment information for a distributed application is captured within an Application Deployment Descriptor (ADD). The ADD contains a list of components that are part of the application, the deployment descriptors for the components (as described in subsection 3.3.4), the connection information between the uses and provides ports of the components, and the operations contained by the provides ports that need to be exposed to the clients.

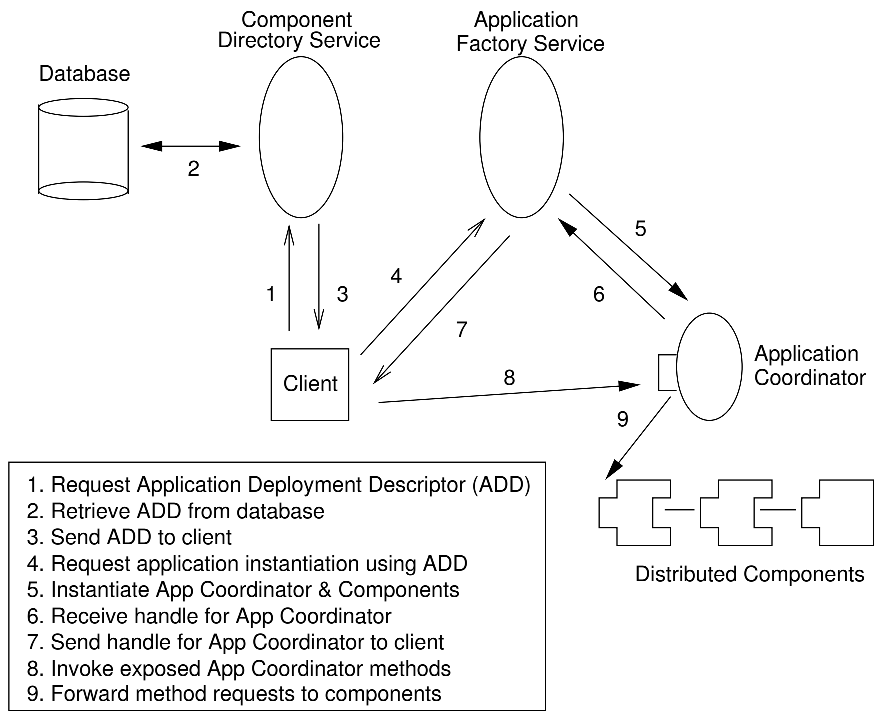
Component Directory Service
Application Factory Service
Database
Client
Application Coordinator
Distributed Components
1
2
3
4
5
6
7
8
9
1. Request Application Deployment Descriptor (ADD)
2. Retrieve ADD from database
3. Send ADD to client
4. Request application instantiation using ADD
5. Instantiate App Coordinator & Components
6. Receive handle for App Coordinator
7. Send handle for App Coordinator to client
8. Invoke exposed App Coordinator methods
9. Forward method requests to components

Figure 3.6: Typical user interaction with an Application Factory

Some operations that may need to be exposed are the ones that bootstrap execution, set parameters, etc.

Typical user interaction with an Application Factory is shown in Figure 3.6. The ADDs for distributed applications are published into a secure directory service. Users authorize themselves with this directory service, browse the applications that can be instantiated, and select the ones that they may be interested in. The users then request the Application Factory to instantiate a particular application by using its ADD. The Application Factory instantiates the particular application on behalf of the user, if he/she is authorized to do so. It may consult a Resource Broker to schedule the distributed application optimally on the resources that it can be deployed on. It also creates an Application Coordinator for the application, which is a Grid service supporting the operations exposed by the components via the ADD. The user can then interact with the distributed application via the Application Coordinator, using standard Grid protocols and clients.

3.4 Sample Application

Our framework has been used in the past for several applications, such as the Linear Systems Analysis (LSA) and the Collision Risk Assessment System (CRASS) projects [21] at Indiana University, a chemical engineering project for multi-scale simulations of copper electrodeposition [16] at NCSA, the Grid-enabled earth system models project [57] at NASA, etc. As an example, we present the NCSA chemical engineering application.

The aim of the chemical engineering application is to link a finite difference electrical resistance code to multiple Monte Carlo electro-deposition simulations, running on resources over the Grid. The number of Monte Carlo simulations may be dynamically varied at runtime. The codes also exchange data after every iteration. It is also desirable that the application writers (chemical engineers) make minimal modifications to their application codes. Our framework lends itself very well to the needs of this application because XCAT3 components can be launched on Grid resources, and can be composed dynamically at runtime.

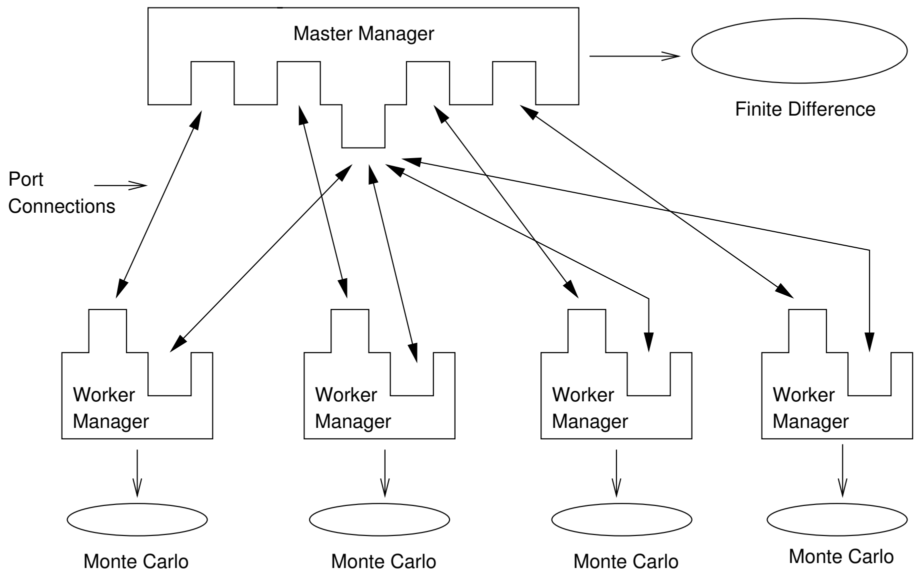

Figure 3.7: Component interaction in the chemical engineering application

Since we wish to make minimal changes to the application codes, we wrap them inside Application Managers. Application Managers execute the application codes as separate processes, and interact with them using file-based mechanisms. This provides an effective way to wrap legacy applications and make them *grid-aware*. Additionally, they expose CCA ports to the outside world for transfer of data and control to the applications they manage. The interactions between the managers for this application is shown in Figure 3.7.

As shown in Figure 3.7, a Master-Worker model is used to orchestrate this application. A Master Manager manages the finite difference code, while a Worker Manager manages every instance of the Monte Carlo code. The actual scientific codes are not modified, but they are expected to write out data from every iteration into data files, which the managers can read and transfer around as need be.

The Master Manager exposes a *ProvidesMasterPort* which accepts data from the Worker Managers (which use matching uses ports) after every iteration of the Monte Carlo code. On receiving data from all Worker Managers, the Master Manager combines them and writes them into a file for the finite difference code to read and proceed with the next iteration.

Every Worker Manager exposes a *ProvidesWorkerPort* which accepts data from the Master Manager (which uses matching uses ports) after every iteration of the finite difference code. On receiving data on their provides ports, the Worker Managers write them into files for the Monte Carlo codes to read, so that they can continue with their next iterations.

In addition to mediating control and data messages for the application codes, the Managers can store the status information for the application as Service Data Elements (SDE). Clients can query the Managers for these SDEs using standard OGSI mechanisms to receive information about application execution.

4

Component Persistence and Migration

In this chapter, we present a user-defined mechanism for component persistence within the XCAT3 [40] framework. We describe the design goals, the architecture, and the programming support for the same. We also discuss how this mechanism can be used to provide a component migration capability, despite the presence of connections and communication between distributed components.

4.1 Design Goals

In XCAT3, component persistence is provided for (1) adapting to dynamic availabilities of Grid resources, and for (2) tolerance to their failures. We discuss the functional and architectural requirements for component persistence in this section.

4.1.1 Functional Requirements

Our motivation for providing component persistence is two-fold:

- **Component Migration:** Component migration is defined as relocation of an individual component instance from one Grid resource to another during execution of a set of distributed components.
- **Distributed Checkpoint & Restart:** Distributed checkpointing is defined as the process of producing a *consistent global checkpoint* for a set of distributed components, which can be used to restart execution upon failure of any of the individual nodes. The restarted components may execute on the same resources if they are available during restart, or may be relocated to other locations.

Although migration of components seems similar to restart, there are some subtle differences.

- Component migration is performed typically for relocating a component instance to a better resource for improving the performance of the application, or upon violations of policies specified by the component writers or resource owners. On the other hand, distributed checkpointing and restart is performed for fault tolerance purposes.
- Component migration involves relocating a component instance to another Grid resource. Restarting a distributed application may or may not involve relocation to other Grid resources.

- Distributed checkpointing requires cooperation from *all* components that are part of the distributed application. All components need to store their states onto stable storage in order to recover from failures. However during component migration, only the state of the component that is being migrated needs to be stored. Cooperation is required from only the components that are connected to the component that is being migrated.
- Component migration typically does not result in any loss of computation. In other words, components need not be rolled back to a globally consistent state from the past for successful migration of another component. However, restarting a distributed set of components involves rolling back the state of all the components to the most recent consistent global checkpoint. All work done after the last checkpoint is lost.

We focus on component migration in this chapter, and discuss distributed checkpointing and restart in Chapter 5. For component migration, we require not only a mechanism to store and retrieve the state of an individual component that is being migrated, but also an algorithm that enables the same, despite uses-provides connections between the component being migrated and the other components that are part of the distributed application.

4.1.2 Architectural Requirements

The functional requirements described above impose the following architectural requirements on our framework.

- **Capturing Component State:** The state of a component in the XCAT3 framework consists of not only the state of the data encapsulated by the component, but also the state of the control flow. This is because components in XCAT3 can have their own threads of execution. This is unlike other traditional component architectures such as Enterprise Java Beans (EJB) or CORBA Component Model (CCM) which only provide mechanisms to capture the data encapsulated by the component. Hence, we need to provide mechanisms within XCAT3 to capture both control and data states for components.
- **Portability:** Grid resources are heterogeneous by definition since they span various administrative domains. Hence, no assumptions can be made about their architectures. This implies that the component state has to be stored in a platform and architecture independent manner, so that it remains portable across the Grid.
- **Checkpoint Size:** It is desirable that the checkpoint size is minimal, so that the distributed checkpointing & restart, and migration implementations are efficient.
- **Scalability:** The architecture should scale well with the number of components that are part of an application, and also with the number of applications themselves.

As we have seen in Chapter 2, user-defined checkpointing techniques are highly portable and efficient with respect with checkpoint sizes, as compared to system-level techniques. Hence, we employ a user-defined mechanism for component persistence within the XCAT3 framework.

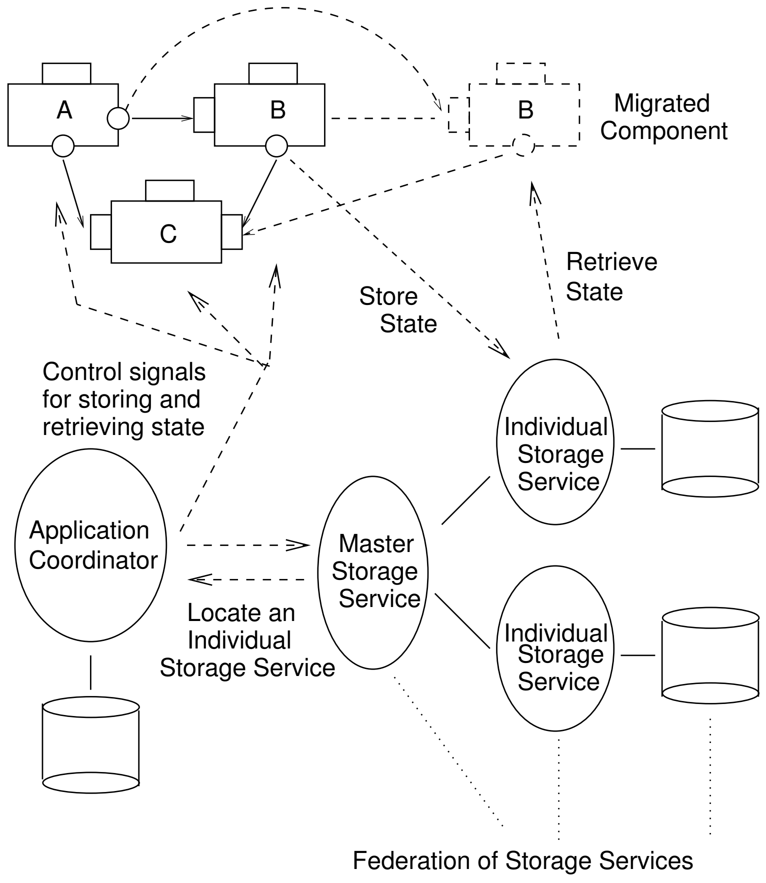
A
B
B
Migrated Component
C
Retrieve State
Store State
Control signals for storing and retrieving state
Individual Storage Service
Application Coordinator
Master Storage Service
Locate an Individual Storage Service
Individual Storage Service
Federation of Storage Services

Figure 4.1: The big picture for component persistence and migration

4.2 Architecture

The major members of the architecture for component persistence and migration are the *Application Coordinator* which is responsible for coordinating the persistence and migration of the components, a federation of *Storage services* which are responsible for storing the states of the components into stable storage, the components themselves which are responsible for generating and re-loading their minimal states, and the framework which is responsible for storing and retrieving these states to and from the Storage services.

The interactions between the various members of the system are shown in Figure 4.1. We describe each of them in detail in this section.

4.2.1 Application Coordinator

An Application Coordinator is created for every distributed application that is implemented as a set of XCAT3 components. The primary role of the Application Coordinator is to maintain locators for all the components that are part of the distributed application, and provide operations that can be used by an end-user to migrate an individual component, or perform distributed checkpointing and restart of the whole application.

The Application Coordinator encapsulates the following information about a set of components:

- **ApplicationID:** Every application consisting of a set of components is assigned an *applicationID* that can be used for identification purposes.
- **ComponentInfo(s):** A list of *ComponentInfo* objects, one for every component that is part of the application, makes up the rest of the state of the Application Coordinator.

Every ComponentInfo object contains enough information to locate a component, and restart it, if need be. It encapsulates the following information:

- **Instance Name:** This is the name assigned to a component by a user. This does not have to unique; it is just a human-readable name for convenience purposes. This is the name returned by the *getInstanceName* method of the ComponentID of the component instance.
- **Instance Handle:** This is the unique GSH of the ComponentID Grid service that represents the component. This handle, along with a Handle Resolver service, can be used to communicate with a component, as long as it is alive. If a component has to be restarted, the instance handle is reused.
- **Creation Protocol:** This is required for restarting a component instance upon failure. During restart, the Application Coordinator tries to use the same creation protocol used to instantiate the component for the first time, if possible.
- **Location:** This is the location that the component is executing on. During restart, the Application Coordinator will try to instantiate the component on the same location, if it is still available.
- **Deployment Descriptor:** This is the XML deployment information required to instantiate a component during restart.

The connection information between the components need not be stored explicitly. This is because all connection information is stored remotely by the components themselves (by the components that *use* the connection). Furthermore, CCA connections are dynamic and may change over the lifetime of the application. Replicating the connection information inside the Application Coordinator would require multiple updates whenever any connections are changed in order to maintain consistency. Hence, we just infer the connection information by querying the components, as needed.

The Application Coordinator need not stay alive throughout the execution of the components, since the components themselves do not need any of the functionality provided by it. Instead, it can be *passivated* by storing all the information into stable storage, and subsequently *activated* from stable storage as and when it is required using the applicationID as the primary key, e.g when a component has to be migrated, or a distributed checkpoint is desired.

4.2.2 Storage Services

Component states should not be stored locally on the Grid resources they are executing on, because they might be irretrievable upon failure. Instead, it is desirable to store component states into stable storage at other remote locations that are always available. Another requirement is that this remote stable storage should scale well with the number of components, and not be a bottleneck as the number of components increases. Although a scalable and reliable persistent storage service is not the primary research focus of this dissertation, 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. Currently, it does so in a round-robin manner in order to distribute network traffic among the Individual Storage services. The client uses this reference to send its data to the Individual Storage service, which stores it into stable storage. After doing so, the Individual Storage service sends the client a unique *storageID* which can be used to retrieve the data at a later time.

The Master Storage service does not maintain any metadata for locating the data referenced by the storageID. The client is expected to retain the reference to the Individual Storage service and the storageID in order to access the stored data in the future.

4.2.3 Component and 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 interface, methods to generate and load component states are added with an assumption that the framework would invoke these as and when required, when it is storing and loading component state into stable storage respectively.

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 migration, and checkpointing & restart. The MobileComponentID implementation provided by the framework retrieves the local component state by making a callback on the *MobileComponent* interface. Additionally, it also generates the state of the Services object, viz. the uses and provides ports added, and their connection information. The above, along with the Service Data Elements for the various Grid services that are part of the component constitute the complete state of a component. This complete component state is encapsulated in a platform-independent XML format, and the MobileComponentID implementation stores and loads it to and from the Individual Storage services, as and when it is required.

4.3 Programming Support

Since we follow a user-defined approach for component persistence, APIs are made available to the component writer to enable the same. In addition, some APIs are used by the framework to coordinate component persistence and migration. The important APIs for the same are described in this section.

4.3.1 MobileComponent

Every component that needs to be persistent and able to migrate has to implement the MobileComponent interface. It contains the following methods.

- **generateComponentState:** This method is supposed to return a string encapsulating the minimal state required to restart a component. It is invoked by the framework when it needs to retrieve the local component state. This method is not expected to return any connection information or Service Data Elements for the component, but only the current state of data and control encapsulated by the component.
- **setComponentState:** This method is invoked by the framework when a component is restarted, in order to load the state of the component from stable storage. The framework passes as arguments the initialized Services object for the component which contains all connection information, and the stringified local state returned by

the component via its `generateComponentState` method. The component is expected to use this information and only initialize its encapsulated data members.

- **resumeExecution:** This method is invoked on the component by the framework to resume all threads of execution. This method is separate from the `setComponentState` for logical separation of data and control states of the component. Additionally, during restart from a globally consistent checkpoint, we need to make sure that all components have loaded their states before any threads are resumed. This is because these threads can affect the state of other components via port calls, and may have unforeseen consequences if they do so before the states of the remote components are loaded consistently from the global checkpoint. We will describe this in further detail in Chapter 5 when we describe the distributed checkpointing and restart algorithms.

4.3.2 MobileComponentID

Every MobileComponent contains an implementation of the MobileComponentID interface, that is provided by the framework. The MobileComponentID interface functions as the entry point into the component during distributed checkpoint and restart, and component migration. We describe some of the operations provided by the MobileComponentID for component persistence and migration in this subsection.

The following methods are invoked on the MobileComponentID of a component whose state needs to be stored, or which is being migrated/restarted.

- **freezeComponent:** This method is a request by the Application Coordinator to freeze execution of the component, which is required before its state can be saved. In some cases, this requirement can be somewhat relaxed, and the component needs to only freeze its outgoing calls, i.e. calls made via its uses ports. This may enable the component to continue with its local execution without affecting the state of any external components, which may be sufficient for most purposes. Once the component does either of the above, it sends a notification to the Application Coordinator that the component is frozen.
- **unfreezeComponent:** This is a notification by the Application Coordinator that the component can proceed with its execution, usually after its state has been stored.
- **storeComponentState:** This is a request by the Application Coordinator to store the state of the component with an Individual Storage service. The MobileComponentID implementation then generates the complete state of the component, and stores the same with the Individual Storage service. It then sends a confirmation message to the Application Coordinator, along with the storageID returned by the Individual Storage service.
- **loadComponentState:** This method is invoked by the Application Coordinator on restart or migration of a component. The MobileComponentID implementation retrieves the complete component state from the Individual Storage service using the

storageID, and loads the component with it. As we have mentioned before, this state contains the local state of the component, the state of the Services object, and the Service Data Elements associated with the various Grid services. When the Services object is set, all the provides and uses ports are initialized. The GSH's for the provides ports are reused, and fresh references to them are registered with the Handle Resolver service.

- **resumeExecution:** This method is a notification by the Application Coordinator that the component can resume all threads of execution. The MobileComponentID implementation simply forwards this notification to the MobileComponent implementation.

The following methods are invoked on the MobileComponentID of a component that is connected via a uses port to a component being migrated.

- **requestMigration:** If a component has to migrate to another location, it can only do so if the other components that are using the services provided by its provides ports hold off on their requests during the migration process. The Application Coordinator requests a component to stop using a particular uses port momentarily using this method. The MobileComponentID implementation waits till the uses port is released, if it is being used, and then notifies the Application Coordinator that it has its approval for migrating the provider component. Further `getPort` calls for that particular uses port block until a confirmation is received that the migration is complete.

- **confirmMigration:** This is a notification that the provider component for a particular uses port is indeed migrating.
- **migrationComplete:** This method is used by the Application Coordinator to notify the component that the provider component for a particular uses port has completed its migration, and that the uses port is now available for further use. All `getPort` calls can now proceed as expected.

4.3.3 Application Coordinator

As we have seen before, the Application Coordinator is used to load and store component states, migrate components, and restart components from a checkpointed state. We look at some of the operations provided by the Application Coordinator in this subsection.

Since an application within the XCAT3 framework is comprised of a set of distributed components, checkpointing and storing/loading states for individual components does not make sense. Instead, only globally consistent states for components should be stored, in order for them to be able to restart correctly. We look at this in more detail in Chapter 5. However, in this subsection, we describe the support provided by the Application Coordinator to migrate individual components, and load and store its own state from/to stable storage.

The **migrateComponent** method is provided by the Application Coordinator to migrate a particular component to another Grid resource. The algorithm used by this method is described in Section 4.4. In short, it makes sure that all remote calls to a migrating component are blocked while a component is migrating. When the migration is complete, all calls may be resumed.

The following methods are provided by the Application Coordinator to store and load itself into stable storage:

- **storeInDatabase:** This method enables storing the state of the Application Coordinator into a database, for future use. The state of the Application Coordinator consists of an applicationID and a list of ComponentInfo objects, as described in Subsection 4.2.1.
- **loadFromDatabase:** This method can be used the load the state of the Application Coordinator from a database. The Application Coordinator initializes itself with the stored information, and can then be used for individual component migration, or checkpointing and restart of a distributed set of components.

4.4 Component Migration

In this section, we explain how the component persistence mechanisms described above can be used to provide migration for individual components on the Grid.

Migration is more complicated for a distributed application, as opposed to a non-distributed one. For a non-distributed application, migration would involve storing the state of the application, re-instantiating it on another resource, and loading the state back to the one that is stored. However, for a distributed application, migrating an individual member of the application is more complicated because it may be communicating with other members of the application. Hence, in order to migrate an individual member of a distributed application, all communication with that member must be stalled during the migration process. Furthermore, all other members communicating with the migrated member must be able to rediscover the latter after the migration is complete in order for the communication to resume. We describe how this is done for migration of components in the XCAT3 framework below.

4.4.1 Migration Algorithm

In short, in order to migrate an individual XCAT3 component to another resource, all communication with that component is halted, the component is migrated, the migrated component is rediscovered by the other components, and all communication to the component is resumed. The detailed algorithm, as shown in Figure 4.2, is as follows.

1. An end-user initiates migration of a component by invoking the `migrateComponent` method on the Application Coordinator.

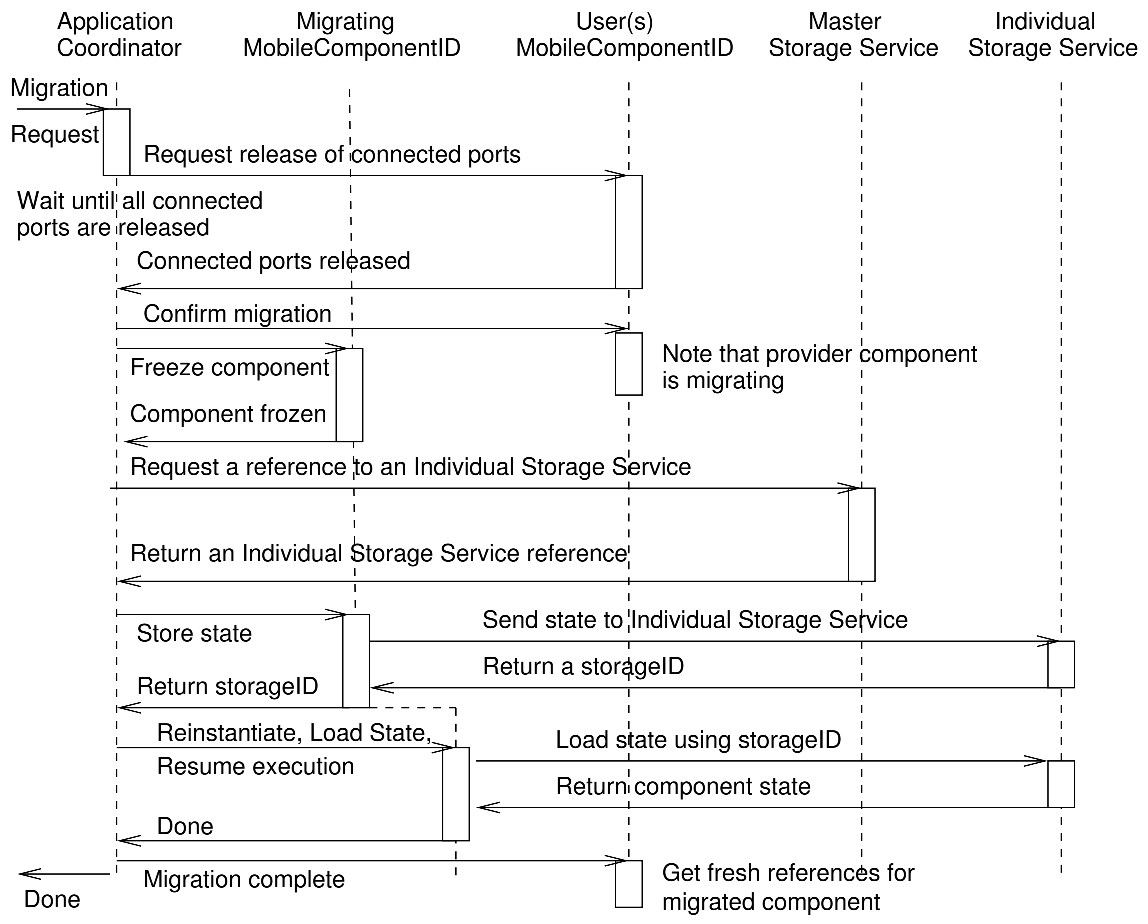
Application Coordinator
Migrating MobileComponentID
User(s) MobileComponentID
Master Storage Service
Individual Storage Service
Migration Request
Request release of connected ports
Wait until all connected ports are released
Connected ports released
Confirm migration
Freeze component
Component frozen
Note that provider component is migrating
Request a reference to an Individual Storage Service
Return an Individual Storage Service reference
Store state
Send state to Individual Storage Service
Return a storageID
Return storageID
Reinstantiate, Load State, Resume execution
Load state using storageID
Return component state
Done
Migration complete
Done
Get fresh references for migrated component

Figure 4.2: The component migration algorithm

2. Using the references for all the components that are part of the distributed application, the Application Coordinator infers the connection graph for the same. The various components that are part of the application constitute the nodes of the connection graph, while the uses-provides connections between them represent directed edges.

3. For every component that is connected to the component being migrated, the Application Coordinator sends a `requestMigration` message to the appropriate MobileComponentID. It also sends the name of the uses port that is connected to the component being migrated.

4. The components that are connected to the component being migrated receive the `requestMigration` message via their MobileComponentIDs. On receipt of this message, they wait until the uses ports mentioned are released by the component via the `releasePort` call, if they are being used. Once the uses ports are released, the components send a `migrationApproval` message to the Application Coordinator. All further `getPort` requests for the above mentioned uses ports block until further notice.

5. On receiving `migrationApproval` messages from all connection users, the Application Coordinator sends a `confirmMigration` message to all the above components.

6. The Application Coordinator is now ready to migrate the individual component. However, before migration, the said component has to be frozen. To do so, the Application Coordinator sends a `freezeComponent` request to the MobileComponentID of the component.

7. On receipt of the `freezeComponent` request, the MobileComponentID implementation waits till all remote invocations are complete. It can do so by waiting till all ports being used are released via the `releasePort` method. All further `getPorts` are blocked. After all ports are eventually released, the MobileComponentID implementation sends a `componentFrozen` message to the Application Coordinator.

8. After receiving the `componentFrozen` message from the component, the Application Coordinator contacts the Master Storage service to receive a reference to an Individual Storage service. Subsequently, it sends a `storeComponentState` request to the MobileComponentID implementation, along with the reference to the Individual Storage service.

9. The MobileComponentID receives the `storeComponentState` request from the Application Coordinator and stores the complete state of the component into the Individual Storage service referenced by the message. Then it sends the storageID to the Application Coordinator as part of the `componentStateStored` message.

10. After the component state is stored, the Application Coordinator destroys the executing instance of the component, and re-instantiates it on the target Grid resource using its stored XML deployment descriptor. It then sends a `loadComponentState` state message to the MobileComponentID, along with the storageID and a reference for the Individual Storage service to be used.

11. On receiving the `loadComponentState` message, the MobileComponentID implementation loads the state of the component from the appropriate Individual Storage service, and sends a confirmation to the Application Coordinator. New references for the provides ports are registered with the Handle Resolver service.

12. The Application Coordinator then sends the `resumeExecution` message to the MobileComponentID, which resumes all threads of execution for the component.

13. For every component that is connected to the component being migrated, the Application Coordinator sends a `migrationComplete` message to the MobileComponentID.

14. On receiving the `migrationComplete` message, the MobileComponentID unblocks all the blocked `getPort` invocations for the concerned uses ports. Additionally, it uses the Handle Resolver service to retrieve the latest Grid Service References for the provides ports of the migrated component, that are being used. This will enable further communication with the migrated component.

15. The `migrateComponent` method is now complete, and control is returned to the end-user.

4.4.2 Discussion

In theory, there is no upper bound on the worst case performance of the migration algorithm because the Application Coordinator can wait indefinitely for components to stop using their uses ports. However, in practice, since the component persistence mechanisms are user-defined, the user can be expected to exhibit good programming discipline and not hold on indefinitely to a uses port that is obtained via a `getPort` invocation. If the components are loosely coupled, which is most often the case for distributed Grid applications, then the Application Coordinator would not wait very long for all uses ports to be released. Hence, in practice, most of the time is spent in the physical migration of the component - storing component state, re-instantiating on another Grid resource, and loading back the state. We discuss the performance of the migration algorithm for a typical application in Chapter 6.

When we presented the migration algorithm above, we did not discuss the implications of failures during the migration process. This is because we use a distributed checkpointing and restart mechanism to recover from failures of individual components, which will be described in Chapter 5. If any component fails during the migration process, we abort the migration of the component and restart the whole application from a globally consistent checkpoint. However, it is possible that the Application Coordinator fails during the migration process. To deal with this situation, the Application Coordinator can write the progress of the migration algorithm into stable storage and restart from it, if need be. The Application Coordinator is typically executing on a local workstation, rather than a Grid resource. This means that in the unlikely event of its failure during the migration algorithm, it can be restarted quickly. The components communicating with the Application Coordinator can keep retrying to send their messages, under the assumption that the Application Coordinator would be re-instantiated promptly.

We do not attempt to prove the correctness of the migration algorithm, since it is trivial. It is obvious that by blocking all communication to and from a component that is being migrated, we reduce the problem of migrating a component in a distributed application to one of a single component without any connections to any other component, as long as the communications can be resumed once the migration is complete. However, an important assumption we make in the algorithm is that no Grid clients may affect the state of a migrating component while it is being migrated. This is typically true as users would only query for component status during execution, and this does not cause any state changes.

5

Distributed Checkpointing and Restart

In this chapter, we present an approach for distributed checkpointing and restart for components within the XCAT3 framework. The distributed checkpointing and restart mechanisms use the concepts of component persistence presented in Chapter 4. We first describe the design goals, the architecture and the programming support, and then present the algorithms for checkpointing and restart for distributed components.

5.1 Design Goals

The primary functional goal of the distributed checkpointing and restart mechanisms in XCAT3 is to provide an ability to recover from failures during execution of long running applications. The goal is to produce *consistent global checkpoints* periodically, which can then be used to restart the execution of the components upon failures.

Failures can be broadly defined as deviation from *normal* program behavior. In distributed systems, these can be divided into *node failures* or *link failures*. Node failures are the ones in which the executing processes, that are part of the distributed system, fail to execute correctly. This might be caused by the process itself, or the resource that the process is executing on. Link failures are the ones in which the communication link between the executing processes deviates from normal behavior.

A node can exhibit *stopping failure* by stopping its execution in the middle of a computation. This is also referred to sometimes as *crash failure*, since it is typically caused by a node crashing. On the other hand, a node can also exhibit *Byzantine failure* wherein it displays random behavior, but continues its execution. This random behavior could possibly be malicious. On the other hand, links can fail by dropping messages sent on them.

During the execution of distributed components on the Grid, we can encounter both node and link failures. Stopping node failures could result from crashes of components or Grid resources, while Byzantine failures could result from several reasons, viz. programmer errors, resources being hacked, etc. Link failures could occur if communication links between the components crash. However, in this work, we do not handle Byzantine node failures, and link failures. Henceforth, whenever we use the term "failure", we imply stopping failures of nodes - the component processes or the computational resources that they are executing on.

In Chapter 4, we identified several design goals for component persistence: (1) mechanisms to capture component state, (2) portability, (3) checkpoint size, and (4) scalability. Apart from the above which are key design issues for distributed checkpointing and restart as well, some of the other goals are as follows:

- **Consistent Global State:** Since an application implemented using the XCAT3 framework consists of a set of distributed components, checkpointing the application calls for implementation of algorithms that produce consistent global checkpoints. Checkpointing components individually is not of great use because a set of individual checkpoints need not constitute a consistent global checkpoint. Additionally, the checkpoints generated by the implementation should have enough information to restart the application upon failures. This includes the data encapsulated by the component, as well as the state of the control flow.

  Furthermore, the checkpointing algorithm itself should be resistant to failures. In other words, it should ensure that any failures during the checkpointing process do not result in partial checkpoints that are not globally consistent.

- **Interoperability:** We wish to remain interoperable with standard Grid and Web service clients. However, we also do not wish to be tied in to any particular implementation of the messaging and transport layers. Hence, our algorithms and techniques should work at a higher level of abstraction, and make no assumptions or changes to any standard protocols and/or semantics.

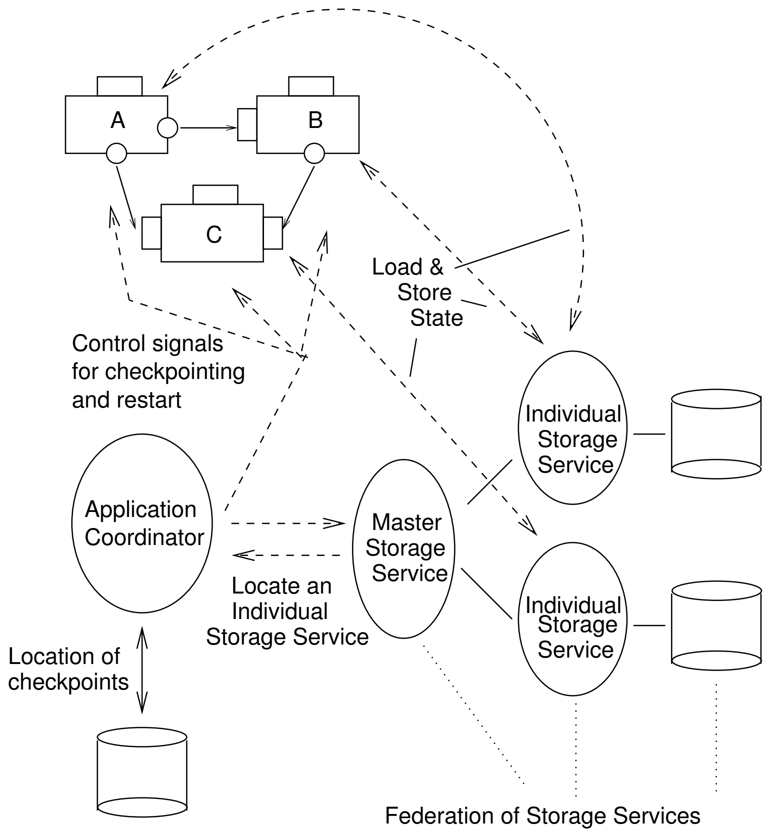

Figure 5.1: The big picture for distributed checkpointing and restart

- **Checkpoint Availability:** The checkpoints of components must not be stored on the Grid resources that they are executing on. The most common cause of failure of a distributed application is failure of the resource on which it is executing on. Since Grid resources belong to different administrative domains, it is not realistic to assume that resources that fail will be brought back up promptly. This necessitates storing the checkpoints at reliable remote locations that are highly available.

5.2 Architecture

The architecture for distributed checkpointing and restart is basically the same as the one described in Chapter 4 for component persistence and migration. The various members of the architecture and their interactions are shown in Figure 5.1.

It consists of the Application Coordinator which is responsible for providing an implementation for the distributed checkpointing and restart algorithms, the federation of Storage services for storing and retrieving component states, the components themselves which are responsible for generating and loading their local states whenever they are notified to do so, and the framework which is responsible for participating in the above algorithms by reacting to control messages and loading/storing these states from/to the Storage services.

As described in Chapter 4, the Application Coordinator maintains information about the set of components that constitute a distributed application within XCAT3. It contains enough information to locate the various components, and re-instantiate them if need be. In addition to the applicationID for an application and the list of ComponentInfo's for the various components, the Application Coordinator also needs to store the locations of the various checkpoints into stable storage. This is somewhat unlike the case for component migration, where the stored component state is only useful immediately when a component is migrated to another location. As far as distributed checkpointing is concerned, it should be possible to locate a consistent global checkpoint at any point of time, so that it could be used to restart execution upon failure.

The roles of the Storage Service and the components are similar to their roles in the migration of individual components. The Master Storage service is responsible for keeping track of Individual Storage services, and schedule storage requests on them. The Individual Storage services are responsible for storing component states and allowing access to them, if need be. The components participate in the distributed checkpointing and restart algorithms by interacting with the Application Coordinator and the Storage services, as we shall see in Sections 5.4 and 5.5.

5.3 Programming Support

As usual, we provide APIs for the component writer and the end-user to avail of the checkpointing and restart capabilities. Furthermore, some APIs are added inside the framework for supporting these capabilities. In this section, we present the important APIs that enable distributed checkpointing and restart.

In Chapter 4, we presented the operations added to the MobileComponent and MobileComponentID interfaces for component persistence and migration. These operations can be used even during distributed checkpointing and restart.

- **MobileComponent:** A component writer generates the minimal state required to restart a component using the `generateComponentState` method, and accepts the same via the `setComponentState` method upon restart. Control threads are started up via the `resumeExecution` method after the states of all components are set, as we shall see from the restart algorithm in Section 5.5.

- **MobileComponentID:** The `freezeComponent` method is used for freezing all outgoing communication for a component during distributed checkpointing. The `unfreezeComponent` method is used for enabling all outgoing communication to proceed after distributed checkpointing is complete. The `storeComponentState` method is used for generating and storing component state with the Individual Storage services, while the `loadComponentState` method is used for the converse. The `resumeExecution` method resumes control threads for a component on restart by invoking its namesake method of the MobileComponent interface.

The `requestMigration`, `confirmMigration`, and `migrationComplete` methods of the MobileComponentID are not used in distributed checkpointing or restart. These methods involve stalling and resuming communication for a particular uses port during migration of a connected provider component. In the case of migration, only the state of the migrating component needs to be stored. However during distributed checkpointing, the state of all components needs to be captured for a consistent global checkpoint. This implies that in this case all components (and all their uses ports) need to be frozen. This functionality is already provided by the `freezeComponent` and `unfreezeComponent` methods, which is hence used in lieu of the former set of methods.

The Application Coordinator serves as the entry point for the checkpointing and restart process, and provides the methods required for the same. The **checkpointComponents** method can be used to create a consistent global checkpoint for an application. It is responsible for ensuring that all the individual checkpoints are correctly stored into the Storage services, and also for storing locations for these checkpoints into stable storage so that they can always be accessible if need be. The **restartFromCheckpoint** method can then be used to restart execution of the application from the most recent global checkpoint. This method is responsible for re-instantiating all the components, loading their states from the Storage services using the stored locations of the checkpoints, and resuming their execution threads. We present the algorithms for these methods in Section 5.4 and 5.5 respectively.

5.4 Distributed Checkpointing

In this section, we present the distributed checkpointing algorithm implemented in the XCAT3 framework in order to generate a consistent global checkpoint.

As we have seen before in Chapter 2, there are several possible approaches that can be taken for producing consistent global checkpoints. The checkpointing algorithm could be **uncoordinated** or **coordinated**. We prefer the coordinated checkpointing approach so as to minimize storage overhead, and to be free from the *domino effect* that can occur in uncoordinated checkpointing. Even with coordinated checkpointing, we could use either **blocking** or **non-blocking** algorithms. However, our goal is to use standard commodity implementations for communicational purposes, and not make any changes to the underlying messaging layer. This helps us preserve interoperability with other standard implementations used on the Grid, and also enables us not to be tied in intrinsically with any such implementations. This means that we could switch the messaging and transport layers easily, if need be. This rules out the use of non-blocking algorithms which are typically communication-induced, and involve changes to the messaging implementation. Hence, we use a coordinated blocking algorithm to create consistent global checkpoints.

5.4.1 Checkpointing Algorithm

In short, in order to create a consistent global checkpoint, all components are first frozen, the individual checkpoints are taken, and all the components are then un-frozen. This is typically how a coordinated blocking checkpointing algorithm works. The detailed algorithm, as shown in Figure 5.2, is as follows.

1. An end-user initiates the distributed checkpointing process by invoking the `checkpointComponents` method on the Application Coordinator.
2. The Application Coordinator retrieves the references for all components that are part of the distributed application.

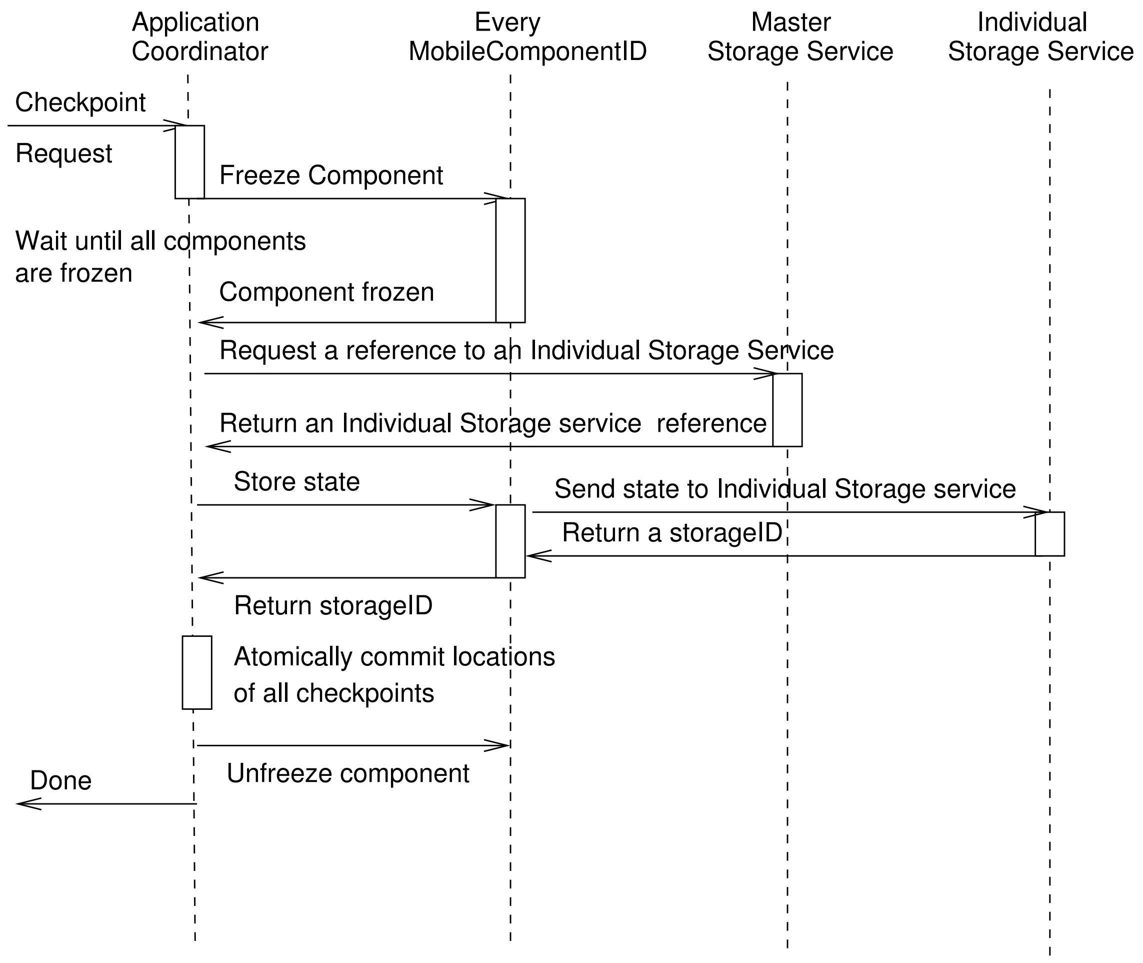
Application Coordinator
Every MobileComponentID
Master Storage Service
Individual Storage Service
Checkpoint Request
Freeze Component
Wait until all components are frozen
Component frozen
Request a reference to an Individual Storage Service
Return an Individual Storage service reference
Store state
Send state to Individual Storage service
Return a storageID
Return storageID
Atomically commit locations of all checkpoints
Unfreeze component
Done

Figure 5.2: The distributed checkpointing algorithm

3. For every component that is part of the distributed application, the Application Coordinator sends a `freezeComponent` request to the appropriate MobileComponentID.

4. 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.

5. 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 interfaces by invoking the `storeComponentState` method.

6. On invocation of the `storeComponentState` method, 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.

7. On receiving the storageID's from every MobileComponentID implementation, the Application Coordinator stores a list of $\{$*instanceHandle*, *individualStorageServiceURL*, *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. We have to ensure that this step either completes successfully in its entirety, or not at all. This is because 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. Hence, this step is performed *atomically* using transaction support provided by a MySQL database.

8. 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.

9. The `checkpointComponents` method is now complete, and control is returned to the user.

5.4.2 Discussion

In theory, just like the migration algorithm, there is no upper bound on the worst case performance of the distributed checkpointing algorithm because the Application Coordinator can wait indefinitely for components to stop communicating with each other. However, as it is with component migration, the component persistence mechanisms are user-defined. Hence, the components can be expected to react to the control signals as expected, and stop communicating with other components within a reasonable period of time. In practice, most of the time will be spent in generation and storage of the checkpoints.

The distributed checkpointing algorithm is implemented in parallel - new threads are created for every component that is part of the application, and the checkpoints are stored with the Storage services by the components independently. Hence, in theory, increasing the number of components should not affect the performance of the application as long the checkpoint sizes remain the same. However, in practice, there may be a few minor overheads for every additional component, viz. thread creation, extra handle resolutions, etc. The above statement is true as long as the Storage services scale well with the increasing number of components.

In addition, the performance of the algorithm also depends on the size of the checkpoints. In specific, the checkpoint time would depend on the time taken by the component with the largest checkpoint size to store its checkpoint, and is expected to be linear with respect to it. We discuss the performance of this algorithm for a typical application in detail in Chapter 6.

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. The assumption here is that state changes can only be made via method invocations over uses-provides connections. Grid clients accessing the components via standard mechanisms may not be allowed to affect component states during the checkpointing process. 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.

As we have mentioned before, we do not deal with link failures during checkpointing or during component execution. However, this can be dealt with at the messaging level by providing reliable communication mechanisms between uses and provides ports in the presence of failures, e.g. by using upcoming Web service standards such as WS-Reliable Messaging [36].

5.5 Restarting from Failures

In this section, we present the algorithm implemented in the XCAT3 framework for restart of distributed components from a consistent global checkpoint.

During restart, it is not sufficient to restart only failed components from the global checkpoint. This is because other components may have communicated with the failed component after a distributed checkpoint was taken. In this case, restarting only the failed component from the most recent checkpoint would create inconsistencies because the restarted component would have no information about the messages that were sent to it after the checkpoint was taken. Hence, it is necessary to roll back the states of *all* components to ones from the most recent global checkpoint; not just the ones that have failed.

Furthermore, when we restart all components from the global checkpoint, we have to ensure 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

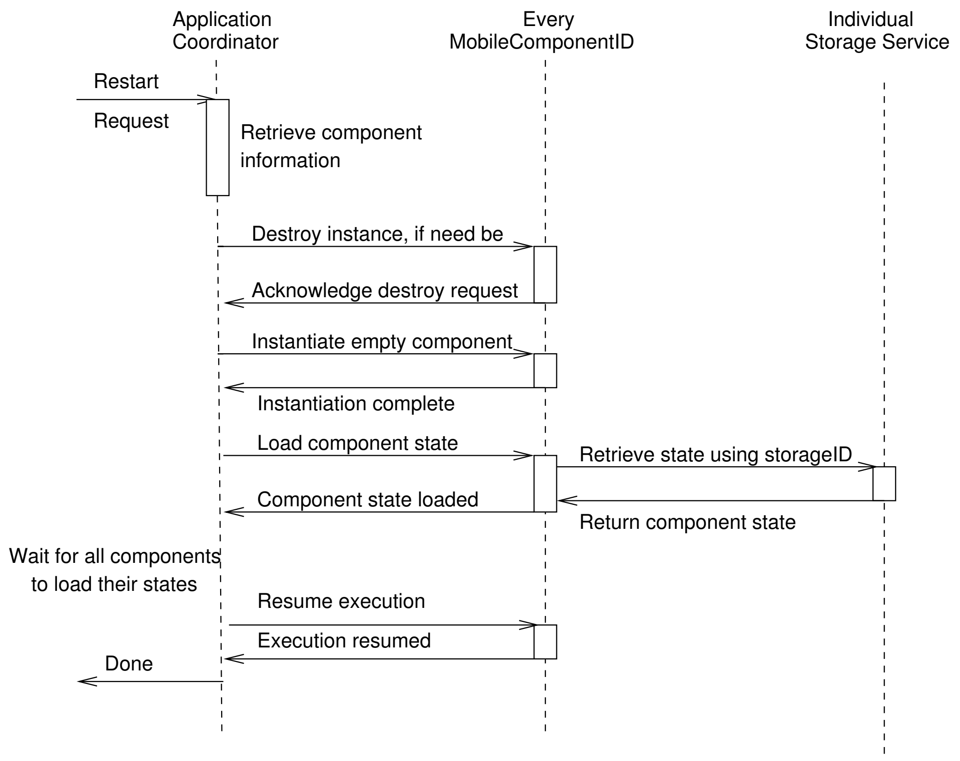
Application Coordinator
Every MobileComponentID
Individual Storage Service
Restart Request
Retrieve component information
Destroy instance, if need be
Acknowledge destroy request
Instantiate empty component
Instantiation complete
Load component state
Retrieve state using storageID
Component state loaded
Return component state
Wait for all components to load their states
Resume execution
Execution resumed
Done

Figure 5.3: The distributed restart algorithm

might cause inconsistencies if they interact with other components whose states have not yet been set. Hence, the restart algorithm should initially set the states of all the components before resuming execution of the control threads of any of them. This is the reason for separate `setComponentState` and `resumeExecution` methods in the MobileComponent and MobileComponentID interfaces.

5.5.1 Restart Algorithm

In short, in order to create a consistent global checkpoint, all the individual components are first instantiated with empty states, the states are then loaded from a consistent global checkpoint, and the control threads are started subsequently. The detailed algorithm, as shown in Figure 5.3, is as follows.

1. An end-user initiates the restart process by invoking the `restartFromCheckpoint` method on the Application Coordinator.

2. 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.

3. 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.

   The `setServices` method is not called for a component that is re-instantiated. The Services object will be set when the component state is loaded from a checkpoint. At the end of this step, the states of all the components are empty.

4. 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.

5. 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.

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

7. 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.

8. The restart process is now complete, and control is returned back to the user.

5.5.2 Discussion

The performance of the restart implementation is not as important as that of the distributed checkpointing. This is because the distributed checkpointing has to be performed when the components are executing, while application restart is performed when one or more components have already failed. Hence, the distributed checkpointing implementation affects the performance of a running application, while the restart implementation does not. However, if all components are instantiated in parallel, the performance should be theoretically independent of the number of components. However, there may be a few minor overheads, viz. thread creation, handle resolution, etc. In addition, the performance would also depend on the size of the checkpoints. In specific, it would depend on the time taken to load and initialize the largest checkpoint from the Storage services, since all the loads would happen in parallel.

We do not attempt to formally prove the correctness of the restart algorithm. However, two key arguments in favor of its correctness are:

- The restart algorithm uses the consistent global checkpoints produced by our distributed checkpointing algorithm, whose correctness we have substantiated earlier. Hence, if the states of the components are correctly set from these checkpoints, they should exhibit correct behavior.
- Control threads are resumed only after every component has loaded its state from the global checkpoint. This ensures that a component whose state has already been loaded from the global checkpoint does not start communicating prematurely with a component whose state is yet to be loaded.

In our current implementation of the XCAT3 framework, we do not provide any mechanism to monitor components in execution. We advocate the use of third party monitoring tools such as Autopilot [48] in order to learn about component failures. Distributed checkpointing and restart, with the help of monitoring tools, can be used to provide fault tolerance for a distributed Grid applications upon stopping failures of components or Grid resources.

6

Performance Analysis

In this chapter, we present a sample application that we use to analyze the performance of the component migration and distributed checkpointing capabilities. We describe the application in detail, and show how it uses the user-defined mechanisms for component persistence presented by the XCAT3 [40] framework. We also look at the performance of the migration and checkpointing implementations under various constraints.

6.1 Sample Application

In this section, we describe the application we use for our performance tests. We describe the flow of control and interactions between the various components, and the implementation details for providing component persistence.

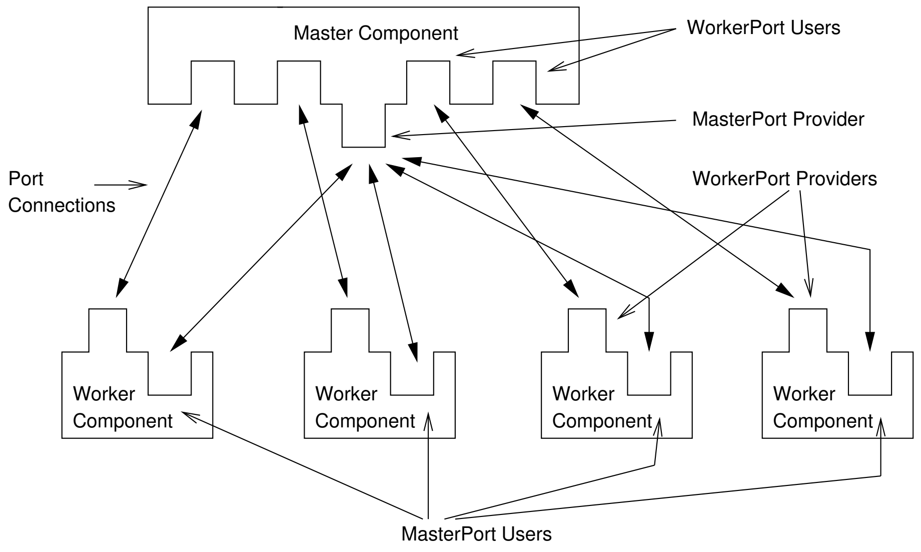

Figure 6.1: The big picture for the Master-Worker application

6.1.1 Overview

The sample application is modeled after the chemical engineering application described in Chapter 3. It is an implementation of a Master-Worker simulation implemented within the XCAT3 framework. The architecture of the application is shown in Figure 6.1.

The application consists of a single Master component, and a number of 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 the Workers. The work packet and the processed results are represented as an array of integers. The Worker component contains a *WorkerPort*, which is a provides port that is responsible for receiving work packets from the Master, also as an array of integers. In addition, the Master component contains a set of uses ports to connect to each of the Workers' WorkerPorts, and

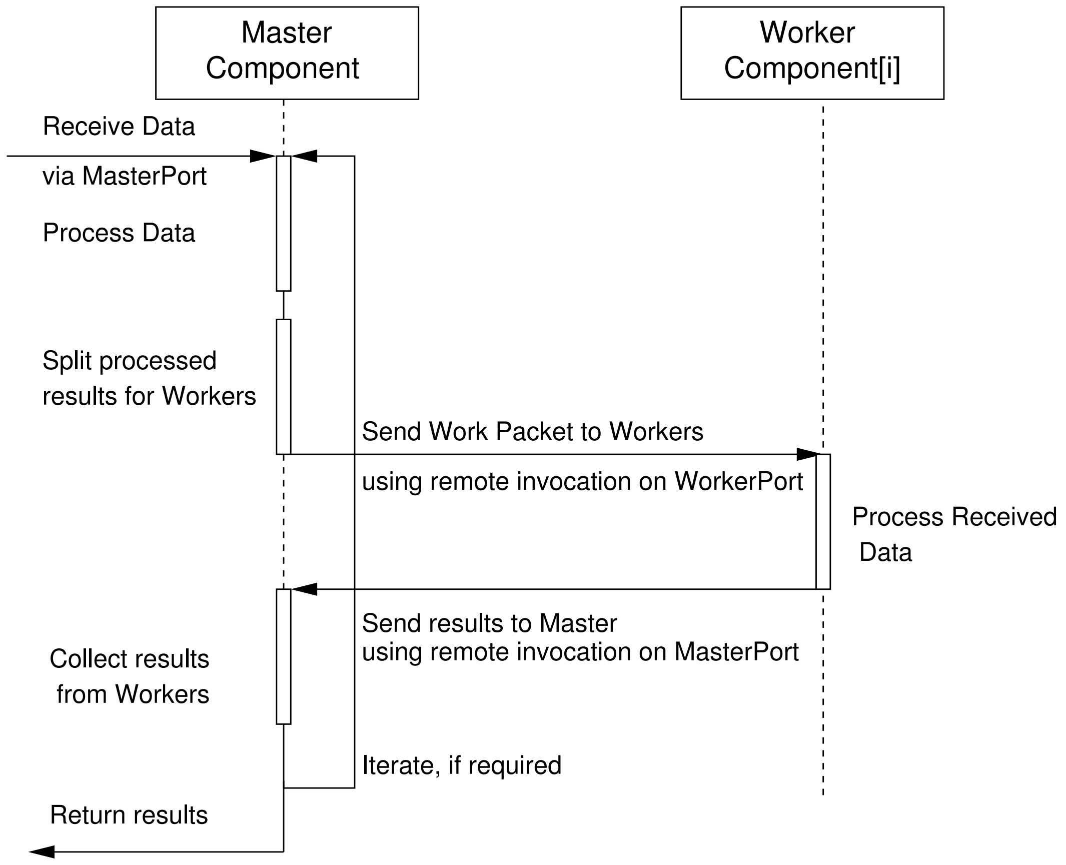
Master Component
Worker Component[i]
Receive Data
via MasterPort
Process Data
Split processed results for Workers
Send Work Packet to Workers
using remote invocation on WorkerPort
Process Received Data
Send results to Master
using remote invocation on MasterPort
Collect results from Workers
Iterate, if required
Return results

Figure 6.2: The interaction between the Master and Workers

every Worker component contains a single uses port to connect to the Master's MasterPort. The application can be scripted to vary the number of components, as well as the size of data being processed, dynamically at run-time.

The interactions between the Master and Worker components, as shown in Figure 6.2, is as follows:

1. The Master component receives a work packet from the user via its MasterPort to bootstrap execution.

2. The Master component proceeds to start processing received data. Since this is a simulation of an actual application, the processing is modeled by putting the Master component to sleep for a period of time.

3. After finishing the processing, the Master component splits the processed data equally for the Workers to process.

4. The Master component sends a work packet to every Worker component using its connections to their WorkerPorts.

5. On receiving the work packets from the Master component, the Worker Components start processing it. This is also modeled by putting the Worker components to sleep for a length of time.

6. After finishing the processing, the Worker components send back the processed results to the Master component via its MasterPort.

7. The Master component receives all the processed results from the Worker components, and combines them together.

8. If further processing needs to be done, the Master component iterates by going to Step 2. If not, the processing is complete.

It can be observed from the component interactions that the Master component and the set of Worker components could never be processing the data at the same time. If the Master component is processing the data, all data resides with it. If all the Worker components are processing the data, all data resides with them. However, it is possible that the data is distributed among the Master and Worker components when (a) all Worker Components have not received data from the Master component after the latter has finished processing, and when (b) the Master component has received processed results from one or more of the Worker components, but not from all of them since they may not finish processing at the same time. We will use the above observations to model the state of the application in Subsection 6.1.2. Furthermore, we shall also see how the distribution of data among the Master and Worker components affects the performance of the migration and checkpointing algorithms.

6.1.2 Component Persistence

In order to store the states of the Master and Worker components, 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 a component. Hence a super-state could be thought of as the state of the control thread of a component. Within a super-state, the physical state may vary depending on the state of the internal data structures at any point of execution.

From Figure 6.2, we infer that the Master component has the following super-states:

- **INITIALIZED**, where it has received its work packet from the user. It is also aware of all the Worker components and has connections to each of them via its uses ports.

- **PROCESSING_DATA**, where it is processing the data received from either the user or the Worker components
- **SENDING_DATA**, where it has finished processing and is sending its processed results to the Worker components. It could have sent work packets to zero or more Worker components in this super-state.
- **RECEIVING_DATA**, where it has finished sending work packets to the Worker components and is waiting to receive processed results from them. It could have received the processed results from zero or more Worker components in this super-state.

Similarly, the Worker component has the following super-states:

- **INITIALIZED**, where it has been initialized with connections to the Master component via its uses port.
- **PROCESSING_DATA**, where it is processing a work packet received from the Master component.

Unlike the Master component, the Worker component does not need SENDING_DATA and RECEIVING_DATA super-states. This is because all data is sent and received to/from the Master component using a single port invocation. Once these invocations are complete, the super-states can be switched immediately. However, the Master component interacts with multiple Worker components. Hence, it may have finished sending or receiving data to/from only a subset of all the Worker components. This necessitates the SENDING_DATA and RECEIVING_DATA super-states.

In order to modularize the code, and make it usable for generating and loading states, we map every super-state into a corresponding method inside the components, e.g. PROCESSING_DATA is mapped to the `processData` method, SENDING_DATA is mapped to the `sendDataToWorkers` method, and so on. In the Master component, all of these methods, except the one representing the final super-state (i.e. RECEIVING_DATA), invoke the methods representing the next super-states as their final statements. The method representing the final super-state, RECEIVING_DATA, simply returns when it is done, causing every other method preceding it to complete as well. This signifies the end of an exchange between the Master and the Worker components. A single control thread keeps continually invoking the method representing the first super-state, `processData`, until all processing is complete. On the other hand, for the Worker component, a control thread is started when it receives a work packet from the Master component. This thread simply invokes with the method corresponding to the PROCESSING_DATA super-state (i.e. `processData`), and terminates when the processing is complete to go back to its INITIALIZED super-state.

The above modularization is done so that the component executions can be resumed easily from a particular super-state during re-instantiation. After the physical state has been set, the execution can be re-instantiated by simply invoking the method representing the appropriate super-state.

In order to store the state of the components correctly, we have to implement the `generateComponentState` method of the *MobileComponent* interface. Both the Master and Worker components have implementations that generate the component state in XML format. Depending on the super-states, different physical states are generated. If the Master component is in super-state PROCESSING_DATA, then the state generated contains the array of integers being processed. If it is SENDING_DATA, then the state generated contains the data not yet sent to the Worker components, and so on for the rest of its super-states, and also for those of the Worker components. The only other thing to ensure is that the physical state returned is consistent, i.e. they do not change during the execution of the `generateComponentState` method, and if they do, they do so atomically. Hence, access to all data members should be thread-safe. If any of these data members need to be modified, they need to be done inside synchronized blocks.

Note that while conversion of an integer array into serialized XML form, every entry is represented as `<value>i</value>`, which is about 25 characters and 50 bytes long. Hence, the checkpoint size in bytes is a little over 50 times the number of integers being processed. This is a very naive implementation, and a BASE64 encoded format could be used for decreasing the checkpoint size.

The `setComponentState` and `resumeExecution` methods of the *MobileComponent* interface have to be implemented for restarting from a checkpoint. Both the Master and the Worker components have implementations of the `setComponentState` method that read in the physical states in XML format and set the appropriate data members from it. The `resumeExecution` methods use the control information that is part of the super-state to resume control threads from appropriate locations, e.g. if the super-state retrieved by the Master component is SENDING_DATA, then it resumes the execution thread such that it proceeds sending data to the rest of the Worker components, i.e. by invoking the `sendDataToWorkers` method.

6.2 Component Migration

In this section, we discuss the performance of the component migration algorithm under various conditions. For the Worker components and the Storage services, we use 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 (OS version: 2.4.26) and are connected via 1Gb/s ethernet. The version of Java used is Sun's JDK 1.4.2_04. For the Master component, we use a Dell Optiplex GX270 with 2.8GHz Pentium 4 processor, 2GB of memory running Gentoo Linux (OS version: 2.6.8), and connected to the ethernet via a 1Gb/s interface. The version of Java used is the same as above.

The storage services are made up by eight Individual Storage services, which for the purposes of this experiment store the component states in memory, but simulate a data write time of 10MB/s and a read time of 100MB/s to and from persistent storage. A simple non-parallel implementation of a GSX Handle Resolver service is used to map a GSH to a GSR, and is also run on one of the nodes of the 8-node cluster.

6.2.1 Performance

We first measure the performance of migrating a Worker component when it is processing data. In this case, all the data is distributed evenly among the Worker components, who are processing it independently from each other.

Figure 6.3 shows the performance of the migration of a Worker component against the total number of Workers, keeping the data size constant. We can observe that the performance of the algorithm is practically constant, and seems independent of the number of Workers. This is so because a Worker component is connected only to the Master component, and to no other Worker. Hence, increasing the number of Workers has no effect on its migration performance. Increasing the data size per Worker increases the migration time as expected, because of the additional time required to generate and store the checkpoints for the larger data size.

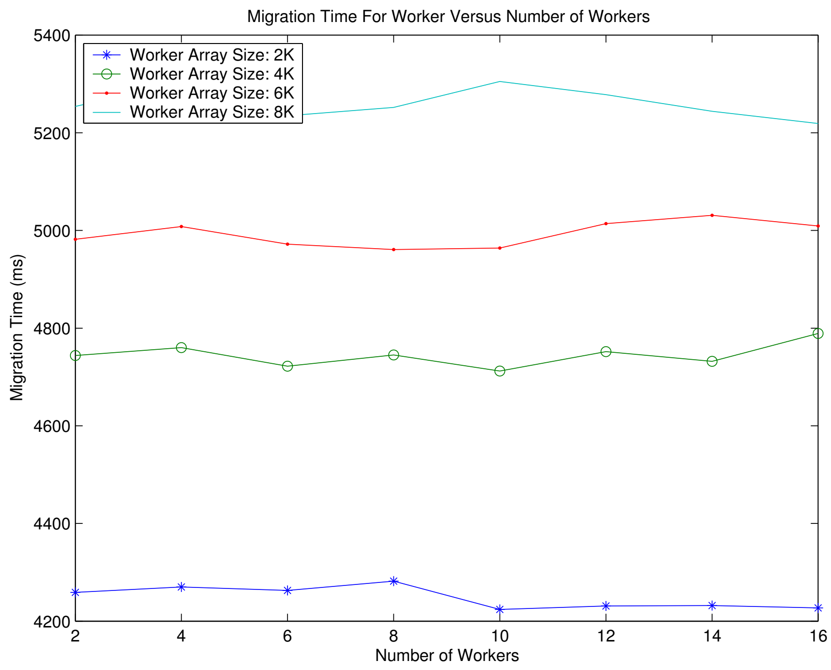

Figure 6.3: Migration of a Worker component when it is processing data

We now measure the performance of migrating the Master component when it is processing data, i.e. when it is in super-state PROCESSING_DATA. In this case, the Master component contains all the data in the system, which equals the product of the number of Worker components and the data size per Worker.

Figure 6.4 shows the performance of the migration of the Master component against the number of Workers that are part of the application, keeping the data size per Worker constant. It can observed that the migration time is not independent of the number of Workers anymore, and grows linearly with the number of Workers. This is because of two reasons - minor overheards such as handle resolution for every Worker component, and more importantly the increase in the data being processed by the Master component as the

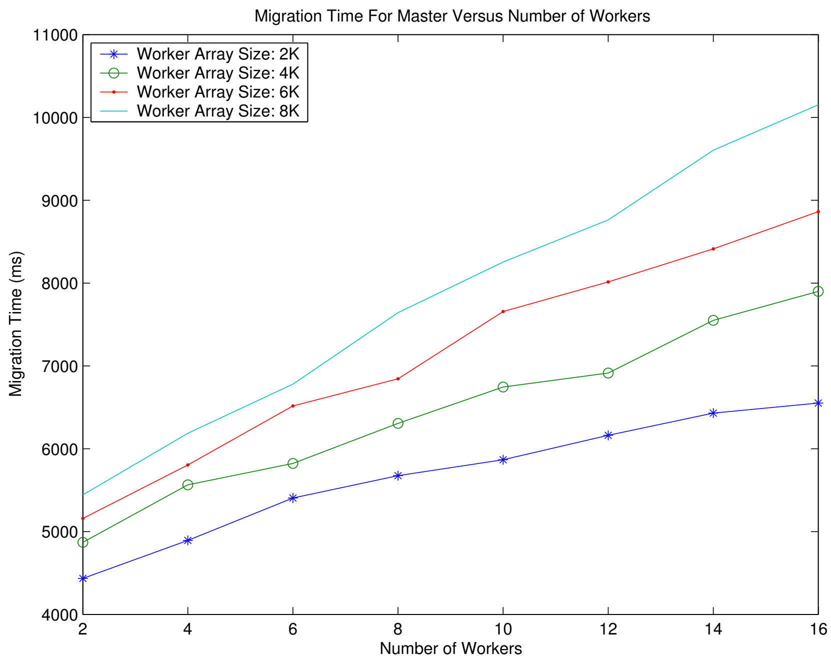

Figure 6.4: Migration of the Master component when it is processing data

number of Workers increase.

It can also be observed that the slopes of the graphs increase with increasing data sizes per Worker component, e.g. in the case where the data size per worker is 2K integers the migration time increases by about 150ms per additional component, and in the case where the data size per worker is 4K integers the migration time increases by about 215ms per additional component. This is because the data being processed by the Master component at this super-state increases linearly with the number of Worker components. Hence, the slopes of the graphs also increase with the data sizes per Worker component. The slopes of the graphs would be constant and independent of the data sizes per Worker component, if the data being processed by the Master component increased at a constant rate independent of the number of Workers.

The migration time includes the time to generate and store component state, time to reinstantiate the component, and the time to load the state back from the Storage services. Component instantiation time is around 3s with our setup, and is constant irrespective of the data sizes and number of components. In the above experiment, for data size 8K integers per Worker and 16 Worker components, the observed checkpoint size is about 8.2MB. It takes about 2.6s to generate and store the component state with the Storage service using an XSOAP invocation. In addition, it takes around 3.6s to retrieve the stored state from the Storage service, and parse and convert it into the actual component state. The times to generate, load, retrieve, and parse component states grow with the data sizes. This performance is slow due to (a) conversion of component state to and from XML, and (b) the use of XSOAP for transferring large messages. Since it takes more time to parse XML than to generate it, the time to generate and store the component state is lesser than the time to retrieve and set it.

6.3 Distributed Checkpointing

In this section, we discuss the performance of the distributed checkpointing algorithm under various conditions. All the components, as well as the storage services, execute under the same environment that is described in Section 6.2.

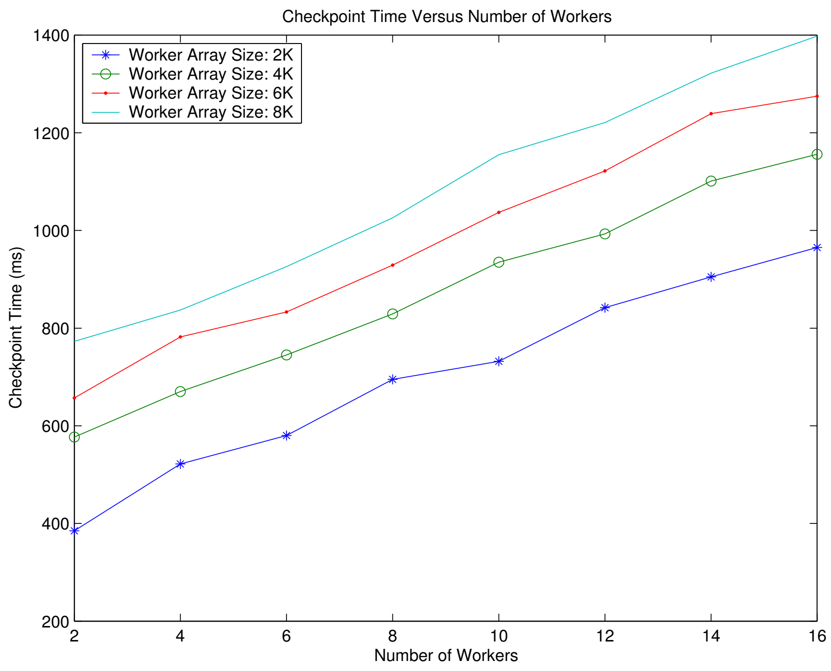

Figure 6.5: Distributed Checkpointing - Workers are processing data

6.3.1 Performance

We measure the performance when the Worker components are processing all the data, and also when the Master Component is processing all the data. In the former case, the times taken by the Worker components to store their states should be fairly even and parallel, and no component should dominate the storage time. In the latter case, the time taken to store the Master component state dominates the time taken to store the Worker states, and hence is not perfectly parallel.

Figure 6.5 shows the performance of the checkpointing algorithm against the number of Workers, when the Workers are processing the data and the data size per Worker is constant. In theory, the performance of the checkpointing algorithm should be independent of the number of Workers for the same data sizes. However, it can be seen from the figure that there is an overhead of about 40ms per additional Worker component, independent of the data sizes. One of the reasons is that a single handle resolution, which has to be performed for every additional component, takes about 25-30ms. As we have mentioned earlier, we use an existing simple implementation of a Handle Resolver service provided by the GSX toolkit. Parallelizing the Handle Resolver service should reduce the slope of the graph, resulting in only minor overheads for every additional component. The slope of the graphs is independent of the data sizes per Worker because the checkpoints are being transferred in parallel to the Storage services. However, as expected, larger data sizes result in slower performance.

Figure 6.6 illustrates the performance of the checkpointing algorithm against the number of Workers, when the data size per Worker is constant and the Master component is processing all data. In this case, we notice that the slopes of the graphs are not independent of the number of Workers, e.g. in the case where the data size per Worker is 2K integers, the checkpoint time increases by about 75ms with every additional Worker component, whereas in the case where the data size per Worker is 4K integers, the checkpoint time increases by 110ms for every additional Worker component. This is so because increasing the number of Workers increases the data being processed by the Master component, since

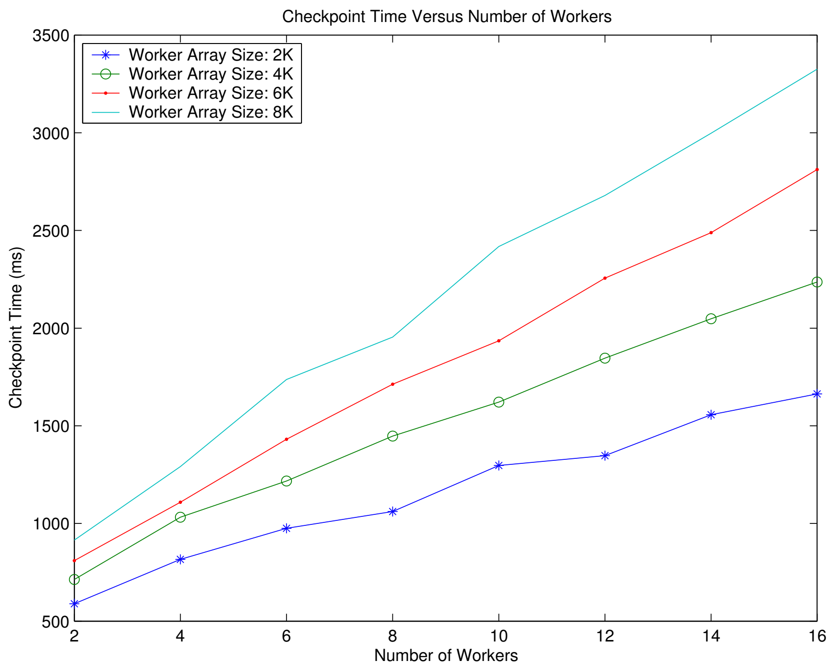

Figure 6.6: Distributed Checkpointing - Master is processing data

it equals the product of the number of Workers and the data size per Worker. This is in addition to the constant overhead per additional Worker component that we observed in Figure 6.5. Also, as we can observe from the graphs, the time to checkpoint components is significantly lower than the time to migrate them. This is because checkpointing only involves storage of component states into the Storage services, and does not involve reinstantiation and retrieval of state.

Thus, from the graphs we can observe that the checkpointing performance is better if the data is evenly distributed among the components. This is because the generation and storage of component states can then be easily parallelized. The performance of the algorithm is directly proportional to the time required to generate and store the largest checkpoint in the system. On the other hand, even though increasing the number of components should theoretically not affect the performance of the algorithm for the same checkpoint size, we observe that there are a few minor overheads that get added for every additional component. The above statement assumes that the federation of Storage services can handle concurrent requests from different components scalably. If this is not true, increasing the number of components would deteriorate the performance of the algorithm considerably.

7

Conclusions and Future Work

In this dissertation, we addressed three key problems in Grid computing - programming mechanisms for orchestration of complex long running distributed applications, adaptability of these applications to the inherently dynamic availabilities of Grid resources, and their ability to recover from resource failures.

We proposed that long running distributed applications on the Grid should be orchestrated by composition of individual simpler components, both in space and time. We also proposed component migration as a mechanism to deal with variable resource availabilities, and distributed checkpointing and restart as a mechanism for providing basic fault tolerance and recovery from failures of Grid resources.

To substantiate the above claims, we presented XCAT3 [40], a framework for CCA-compatible components consistent with current Grid standards, which enables orchestration of complex distributed applications on the Grid. We also presented the architecture for persistence of components within the XCAT3 framework, and showed how it can be used for providing component migration. We also discussed the distributed checkpointing and restart capabilities that are useful for fault tolerance purposes. We analyzed the performance of both of the above capabilities with the help of a sample application, and showed that it scales well with respect to the number of components as well as the checkpoint sizes.

7.1 Contributions

We made the following key contributions in this dissertation.

- **A CCA framework for the Grid:** The XCAT3 framework is an implementation of the Common Component Architecture (CCA) specification such that it is consistent with the Open Grid Services Infrastructure (OGSI). It uses a novel way to map CCA components to a set of Grid services, that are accessible via standard Grid mechanisms. Conforming to the CCA specification enables the use of standard component composition techniques for creating distributed applications (composition in space), whereas conforming to the Web services based OGSI specification enables exploitation of desirable features for the Grid, such as multiple level naming, dynamic service introspection, interoperability, and the use of Web services based workflow tools for composition in time.

- **Component migration:** The XCAT3 framework provides a capability to migrate individual components across Grid resources, despite connections to and from them. Typical XCAT3 based applications consist of a set of distributed components, with direct uses-provides connections between them. The framework not only provides user-defined mechanisms for component persistence, but also provides an algorithm for migrating individual components even if they may be communicating with other components that are part of the application. We use user-defined mechanisms for components, as opposed to the ones that are system-level, for reasons of portability and smaller checkpoint sizes.
- **Distributed checkpointing and restart:** The XCAT3 framework also provides a capability for distributed checkpointing of a set of components, which can then be used for restart upon failures. The framework implements a coordinated blocking algorithm for producing a consistent global checkpoint for a distributed application, using the aforementioned component persistence mechanisms. The global checkpoint can be retrieved from stable storage to restart an application, if need be. The only failures we handle are stopping failures of nodes. We do not address Byzantine failures of nodes, or link failures.

Complex long running distributed applications can be implemented effectively using the composition capabilities of the XCAT3 framework. The component migration and checkpointing capabilities can be then used for adapting to dynamic Grid environments.

7.2 Future Work

The work done in this dissertation can spin off further research is several directions. We divide these into three main categories, viz. the framework, component migration, and fault tolerance.

7.2.1 Framework Issues

Some of the future work for the XCAT3 framework are as follows:

- **WSRF compatibility:** The Web Service Resource Framework (WSRF) has recently superseded the Open Grid Services Infrastructure as the next de facto standard for the Grid. However, at the time of writing this dissertation, WSRF is still under active revision, and no stable implementations are available for use. We plan to be compatible with WSRF once it is more stable and widely accepted.
- **Use of alternate protocols:** Although SOAP is the standard protocol used in Grid computing, it is well known to be extremely inefficient. The Proteus multi-protocol library [14] allows exposing an endpoint via multiple protocols. Thus, we could still use SOAP as the basic protocol for interoperability, but switch to more efficient ones if they are supported on both the client and the server sides. We plan to use Proteus in future versions of XCAT3 for enabling high performance communication between components, if need be.

- **Asynchronous communication:** At present, the communication between components is via uses-provides connections between them. This is synchronous in nature. We plan to add asynchronous communication between components as an additional feature, by using implementations of popular Web services based publish-subscribe systems such as Web Services Notification [4].

7.2.2 Component Migration

In XCAT3, we address *how* to migrate individual components if required. However, we do not attempt to answer *why* and *where* these components should migrate. This is part of our future work.

- **Policies:** Migration can be triggered by violations of policies specified by either the component writer or the resource provider. Several projects use policy based resource scheduling, e.g. Condor uses *ClassAds* for specification of application and resource policies, while Cactus uses *Performance Contracts* for specification of application requirements. If these policies are violated during execution of an application, it is migrated to another resource where these policies are no longer violated. In the future, we plan to use a similar mechanism for specification and evaluation of policies for the XCAT3 framework.
- **Co-scheduling:** Since an application in the XCAT3 framework is distributed, migrating one component may have an adverse effect on other components that are

communicating with it. Hence, the said policies would have be evaluated for the whole application, and not just for individual components. The framework has to ensure that the components are scheduled such that all the policies are satisfied simultaneously. If any of them fails to be satisfied during execution, the affected component needs to be migrated such that the all the other policies remain satisfied after migration.

7.2.3 Fault Tolerance

In our dissertation, we presented an approach to distributed checkpointing and restart. In the future, we plan to evaluate optimizations during checkpointing, fault monitoring in order to restart applications automatically, and strategies to deal with link failures.

- **Checkpoint optimizations:** Currently when component states are stored with the Individual Storage services, the complete states of the components are generated and transferred to them. We plan to investigate techniques that may be applicable in order to reduce the time required to generate and transfer these checkpoints, e.g. incremental checkpointing. Since we use a user-defined approach to component persistence, we also plan to evaluate the effects of these techniques on programmer overhead.

We also plan on investigating techniques to optimize the transfer of state to the Storage services. At this point, the Individual Storage services are chosen in a round-robin fashion. The location of the components could be used to find the closest Individual Storage service from a component for faster loads and stores. Additionally, better algorithms can be chosen for improving the load-balancing between the Storage services.

- **Fault monitoring:** In XCAT3 we only provide an ability to recover from faults by restarting an application. We do not provide a capability to detect these faults, as they occur. We plan to use one of the several available tools in order to do so, e.g Autopilot [48], Network Weather Service [56], etc. If the faults are detected reliably as and when they occur, the applications can be restarted promptly upon failures.
- **Link failures:** In this dissertation, we do not handle link failures. However, we could use a reliable messaging layer in order to ensure that messages between components are reliably delivered even in the presence of link failures. The Web Services Reliable Messaging (WSRM) [36] is an upcoming specification for reliable delivery of Web services based messages in the presence of failures. We plan to investigate WSRM for reliable port invocations between components.

Bibliography

[1] The Global Grid Forum, 2004. http://www.ggf.org.

[2] Universal Description Discovery and Integration of Business for the Web (UDDI), Dec 2003. http://www.uddi.org/specification.html.

[3] The Blocks Extensible Exchange Protocol Core (BEEP), March 2001. http://www.ietf.org/rfc/rfc3080.txt.

[4] Akamai, C.A.I., Fujitsu, Globus, HP, IBM, SAP AG, Sonic, and TIBCO. Web Services Notification, June 2004. http://www-106.ibm.com/developerworks/library/specification/ws-notification/.

[5] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience 14(5)*, 2002.

[6] G. Allen, D. Angulo, I. Foster, G. Lanfermann, C. Liu, T. Radke, E. Seidel, and J. Shalf. The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *The International Journal of High Performance Computing Applications*, 15(4), 2001.

[7] D. Bartlett. CORBA Component Model (CCM): Introducing next generation CORBA, April 2001. http://www-106.ibm.com/developerworks/webservices/library/co-cjct6.

[8] J. Basney, M. Livny, and T. Tannenbaum. High Throughput Computing with Condor. In *HPCU news, Volume 1(2)*, June 1997.

[9] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A component based services architecture for building distributed applications. In *Proceedings of Ninth IEEE International Symposium on High Performance Distributed Computing Conference, Pittsburgh*, August 1-4 2000.

[10] Center for Applied Scientific Computing (CASC), LLNL. The Babel Project, 2004. http://www.llnl.gov/CASC/components/babel.html.

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

[12] D. Chess, C. Harrison, and A. Kershenbaum. Mobile Agents: Are They a Good Idea? *Lecture Notes In Computer Science*, 1222, 1997.

[13] K. Chiu. An Architecture for Concurrent, Peer-to-Peer Components, 2001. PhD Dissertation, Dept of Computer Science, Indiana University.

[14] K. Chiu, M. Govindaraju, and D. Gannon. The Proteus Multiprotocol Library. In *Supercomputing 2002*, November 2002.

[15] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *IPPS/SPDP 98, Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[16] T.O. Drews, S. Krishnan, J. Alameda, D. Gannon, R.D. Braatz, and R.C. Alkire. Multi-scale Simulations of Copper Electrodeposition onto a Resistive Substrate. *IBM Journal of Research and Development*, 2004.

[17] N. Elliott, S. Kohn, and B. Smolinski. Language Interoperability for High-Performance Parallel Scientific Components. In *International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE 1999)*, September 29 - October 2 1999. San Francisco, CA.

[18] 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.

[19] A. Bosworth et al. Web Services Addressing, March 2004. http://www-106.ibm.com/developerworks/library/specification/ws-add/.

[20] D. Box et al. Simple Object Access Protocol 1.1, Dec 2003. http://www.w3.org/TR/SOAP.

[21] 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.

[22] E. Christensen et al. Web Services Description Language (WSDL) 1.1, 2003. http://www.w3.org/TR/wsdl.

[23] F. Curbera et al. Business Process Execution Language for Web Services, Version 1.0, July 2002. http://www-106.ibm.com/developerworks/library/ws-bpel.

[24] I. Foster et al. Modeling Stateful Resources with Web Services, March 2004. http://www.ibm.com/developerworks/library/ws-resource/ws-modelingresources.html.

[25] K. Czajkowski et al. WS-Resource Framework, May 2004. http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf.

[26] 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.

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

[28] Global Grid Forum. Grid Checkpointing and Recovery Working Group, June 2004. http://gridcpr.psc.edu/GGF.

[29] I. Foster. What is the Grid? A Three Point Checklist, July 2002. www-fp.mcs.anl.gov/ foster/Articles/WhatIsTheGrid.pdf.

[30] I. Foster and C. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

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

[32] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *J. Parallel and Distributed Computing*, 37:70–82, 1996.

[33] D. Gannon, R. Ananthakrishnan, S. Krishnan, M. Govindaraju, L. Ramakrishnan, and A. Slominski. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 9, Grid Web Services and Application Factories. Wiley, 2003.

[34] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. XCAT 2.0: Design and Implementation of Component based Web Services. Technical report, Department of Computer Science, Indiana University, June 2002. TR562.

[35] S. Graham, S. Simeonov, T. Boubez, D. Davis, G. Daniels, Y. Nakamura, and R. Neyama. *Building Web Services with Java*. SAMS, 2002.

[36] IBM, BEA, Microsoft, and TIBCO Software. Web Services Reliable Messaging, June 2004. http://www-106.ibm.com/developerworks/webservices/library/ws-rm.

[37] Albert Einstein Institute. The Cactus Project, 2003. http://www.cactuscode.org.

[38] C. Johnson and S. Parker. The SCIRun Parallel Scientific Computing Problem Solving Environment. In *9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[39] J. Kohl, P. Papadopoulos, and G. Geist. Cumulvs: Collaborative infrastructure for developing distributed simulations. In *8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[40] 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.

[41] Argonne National Lab. The Globus Toolkit, 2004. http://www.globus.org.

[42] M.J. Lewis and A. Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, Aug 1996.

[43] W.J. Li and J.J. Tsay. Checkpointing Message-Passing Interface (MPI) Parallel Programs. In *Pacific Rim International Symposium on Fault Tolerant Systems*, 1997.

[44] F. Manola and E. Miller. RDF Primer, January 2003. http://www.w3.org/TR/rdf-primer.

[45] Microsoft. COM, 2003. http://www.microsoft.com/com.

[46] Microsoft. DCOM, 2003. http://www.microsoft.com/com/tech/dcom.asp.

[47] Sun Microsystems. EJB, 2003. http://java.sun.com/products/ejb.

[48] R.L. Ribler, J.S. Vetter, H. Simitci, and D.A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *7th IEEE Symposium on High-Performance Distributed Computing*, 1998.

[49] S. Sankaran, J.M. Squyres, B. Barrett, and A. Lumsdaine. Checkpoint/Restart System Services Interface (SSI) Modules for LAM/MPI. Technical report, Department of Computer Science, Indiana University, 2003. TR578.

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

[51] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. 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.

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

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

[54] G. von Laszewski, J. Gawor, S. Krishnan, and K. Jackson. *Grid Computing: Making the Global Infrastructure a Reality*, chapter 25, Commodity Grid Kits - Middleware for Building Grid Computing Environments. Wiley, 2003.

[55] F. Vraalsen, R.A. Aydt, C.L. Mendes, and D.A. Reed. Performance Contracts: Predicting and Monitoring Grid Application Behavior. In *2nd International Workshop on Grid Computing/LNCS (GRID 2001)*, Nov 2001.

[56] R. Wolski, N.T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 1999.

[57] S. Zhou, B. Womack, and G. Higgins. Grid-enabled Earth System Models. In *NASA's Earth Science Technology Conference 2004*, June 2004.

Curriculum Vitae

Sriram Krishnan was born in Mumbai, India on June 30, 1978, He received his B.E. in Computer Engineering from Vivekanand Education Society's Institute of Technology (VESIT), Mumbai, India in 1999 which is affiliated to the University of Mumbai, India. He subsequently received his M.S. in Computer Science from Indiana University, Bloomington in 2001.