A SCALABLE, DECENTRALIZED COORDINATION INFRASTRUCTURE FOR GRID ENVIRONMENTS

BY ZHEN LI

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

May, 2007

ABSTRACT OF THE DISSERTATION

A Scalable, Decentralized Coordination Infrastructure for Grid Environments

by Zhen Li

Dissertation Director: Professor Manish Parashar

While Grid computing is rapidly emerging as the dominant paradigm for distributed problem solving for a wide range of application domains, the heterogeneity, dynamism, and uncertainty of Grid environments result in significant application coordination challenges. A key challenge is managing the runtime dependencies and interactions among the elements. These dependencies and interactions can be complex and various and both system entities and interactions between them can be ad hoc and opportunistic. As a result, realizing these coordinations becomes extremely challenging.

This research investigates a shared-space based decentralized architecture model for addressing scalable and robust coordination for Grid applications. This model employs fully decentralized architecture and provides a global virtual shared-space abstraction that can be associatively accessed by all peers in the system. In this research, we design and develop Comet coordination infrastructure to demonstrate the conceptual architecture model. The architecture of the Comet is based on a content-based distributed hash table, which employs a locality preserving mapping to map the multi-dimensional information space used by the coordinating entities to the one-dimensional peer node index space. The resulting shared-space maintains content locality and guarantees that content-based information queries are delivered with bounded costs. The key

contribution of this thesis is a conceptual architecture model and an implementation infrastructure for realizing coordination abstractions that support dynamic, scalable, and asynchronous application interactions on wide-area Grid environments.

The developed architecture model and the Comet infrastructure are used to support coordination and computation in Grid environments. Two prototype systems have been implemented and evaluated. The first prototype, CometG, provides a decentralized (peer-to-peer) computational infrastructure that extends Desktop Grids to support parallel asynchronous applications. CometG constructs decentralized coordination spaces and programming abstractions for parallel asynchronous iterative computations and asynchronous formulation of the replica exchange algorithm for molecular dynamics applications. The second prototype, Rudder coordination framework provides agent abstractions and coordination protocols for supporting dynamic composition of Grid applications. Experimental evaluations of these prototypes demonstrate the flexibility, scalability and effectiveness of the infrastructure, as well as its ability to support complex coordination requirements of Grid applications.

Acknowledgements

I would like to gratefully thank my advisor Professor Manish Parashar for his enthusiasm, his inspiration, his encouragement, his sound advice and great efforts during my research years in The Applied Software Systems Laboratory (TASSL). I am very thankful to Professor Ivan Marsic, Professor Hoang Pham, Professor Deborah Silver, and Professor Yanyong Zhang for being on my thesis committee and for their advice and suggestions regarding the thesis and beyond. I am grateful to my colleagues at TASSL and other friends at Rutgers University for their emotional support and help, which makes my study at Rutgers enjoyable and fruitful. I would like to thank the staff at the Center for Advanced Information Processing (CAIP) and Department of Electrical and Computer Engineering for their assistance and support. Finally, I wish to thank my parents, my brother, and my husband for their understanding, endless encouragement, and love. To them, I dedicate this thesis.

The research presented in this thesis was supported in part by National Science Foundation via grants numbers CNS 0305495, CNS 0426354, IIS 0430826 and ANI 0335244, and by Department of Energy via the grant number DE-FG02-06ER54857. I thank these funding agencies for their generous financial support.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

The wide-area Grid computing, which is based on the aggregation of large numbers of independent hardware, software and information resources, is rapidly emerging as the dominant paradigm for distributed problem solving for a wide range of application domains. The Grid computing environment, such as pervasive information systems and computational Grids, has enabled a new generation of applications that are based on seamless access, aggregation and interaction. Grid applications, combining intellectual and physical resources spanning multiple organizations and disciplines, provide vastly more effective solutions to scientific, engineering, business and government problems [70]. For example, it is possible to conceive a new generation of scientific and engineering simulations of complex physical phenomena that symbiotically and opportunistically combine computations, experiments, observations, and real-time data, and can provide important insights into complex systems such as interacting black holes and neutron stars, formations of galaxies, and subsurface flows in oil reservoirs and aquifers etc.

The underlying Grid computing environment is inherently large (scaling to thousands of nodes), complex, heterogeneous and dynamic, globally aggregating large numbers of independent computing and communication resources, data stores and sensor networks. As a result, maintaining global knowledge about the current system nodes and the interaction entities in the applications is infeasible. Furthermore, the emerging applications are similarly complex and highly dynamic in their behaviors and interactions. The defining characteristics of these emerging systems and applications are [70]:

- *Heterogeneity*: The environments aggregate large numbers of independent and geographically distributed computational, communication and information resources. Also, the capabilities (e.g., storage, processing power, bandwidth), of the nodes are different, and they may choose to share only a fraction of their resources. Furthermore, the network interconnections among the nodes do not conform to a single architecture or technology. Similarly, applications typically combine multiple independent distributed software elements, such as components, services and data sources, with different executing operating systems, versions, configurations, etc.

- *Dynamism*: The computation, communication and information environment is continuously changing during the lifetime of an application, including the availability and state of resources and services. The nodes can join, leave the system, or fail at any time. The applications similarly have dynamic runtime behaviors, where the organization and interactions of the elements can change based on context, content and state.

- *Uncertainty*: This is caused by multiple factors including: (1) dynamism, which introduces unpredictable and changing behaviors that can only be detected and resolved at runtime; (2) failures, which have an increasing probability of occurrence as the system scales increase; and (3) incomplete knowledge of global system state, which is intrinsic to large decentralized and asynchronous distributed environments.

Together, the characteristics of Grids listed above result in significant application development and management challenges that span all levels, including the programming models, run time systems, middleware, and operating systems. Enabling flexible and robust coordination becomes a key issue.

1.2 Problem Description

The scale, heterogeneity, and dynamism of emerging Grid environments make coordination a significant and challenging problem. Coordination can be defined as *managing*

the runtime dependencies and interactions among the elements in the system. These dependencies and interactions can be complex and various (e.g. peer-to-peer, client-server, producer-consumer, collaborative, at-most/at-least/exactly, etc.). Further, both the coordinated entities and their interactions can be ad hoc and opportunistic. As a result, realizing these coordination behaviors using low-level protocols becomes extremely difficult. To address this issue, coordination infrastructures that provide high-level abstractions to address scalable flexible coordinations are necessary. These infrastructures should provide middleware services to deal with data communication and synchronization, as well as process cooperation and competition. They should also allow application developers to distinguish the coordination concerns from the computation concerns, and allow these concerns to be separately specified and implemented.

Clearly, designing and developing a coordination infrastructure is non-trivial. A key issue in the design and development of such a coordination infrastructure is the choice of the underlying coordination model. Models based on direct communication, such as Remote Procedure Call (RPC), imply a strict coupling in time, place and name between the interacting entities, which is not suitable for the large decentralized systems. This is because maintaining common knowledge about the names/identifiers, addresses of an end-point as well as the syntax and semantics of the interfaces in these systems is infeasible. In contrast, models based on shared-space abstractions, which provide temporal and spatial decoupling and associative data access mechanisms, can deal with the incomplete knowledge and system dynamism and heterogeneity. As a result, this approach has been popularly adopted for coordination in distributed environments. These include middlewares for supporting the coordination among system entities or software agents (e.g., TuCSoN [67] and MARS [29]), for dealing with physical system dynamism and uncertainties (e.g., Lime [63] and PeerWare [35]), and for exchanging data between heterogeneous components using XML tuples (e.g., XMLSpaces [88] and XMARS [30]). While these systems (discussed in more detail in the following Chapter) have successfully demonstrated the power and feasibility of shared-space based middleware, scalability and resilience in distributed wide-area environments remain open issues.

1.3 Overview of Comet

The overall goal of this research is to design, implement and evaluate a coordination infrastructure that enables the communications and interactions of heterogeneous entities in large decentralized distributed Grid environments. The specific objectives includes: (i) design a conceptual architecture model, which provides scalable, resilient, and simple coordination and programming abstractions; (ii) develop an coordination infrastructure to demonstrate the conceptual architecture model; (iii) develop application systems to illustrate the effectiveness and feasibility of using the infrastructure for supporting wide-area Grid applications.

This thesis presents Comet, a fully decentralized coordination infrastructure for Grid environments. Comet provides a scalable, decentralized tuple space abstraction to address communication and synchronization of distributed processes and software elements. Comet defines a fully decentralized conceptual architecture model, which provides a global virtual shared-space constructed from the semantic information space used by entities for coordination and communication. Comet adapts Squid [79] information discovery scheme and implements an associative Distributed Hash Table (DHT), which deterministically maps the information space using Hilbert Space Filling Curve (SFC) [62] onto the dynamic set of peer nodes in the Grid system. The locality preserving nature of Hilbert SFC enables the Comet to maintain content locality and efficiently resolve content-based lookups. The Comet architecture model provides communication, coordination, and application abstractions, which allow these programming concerns to be separately addressed during system and application development.

The Comet infrastructure implements the conceptual architecture model. A peer node in Comet has 3 layers: a communication layer which provides scalable content-based messaging services and manages system heterogeneity and dynamism; a coordination layer which provides Linda-like [31] primitives and supports a shared-space coordination model; and an application layer which provides programming frameworks and mechanisms. The Comet infrastructure has been deployed and evaluated on a wide-area environment using the PlanetLab [7] testbed, as well as a campus network

at Rutgers University.

The developed Comet infrastructure is used to support coordination and computation in Grid environments. Two prototype systems have been implemented and evaluated. The first prototype, CometG [56], provides a decentralized computational infrastructure that extends Desktop Grid environments to support parallel asynchronous applications. CometG constructs decentralized coordination spaces and application-level programming abstractions for parallel asynchronous iterative computations as well as asynchronous formulation of the replica exchange algorithm for molecular dynamics applications. The second prototype, Rudder [55, 53, 54, 58] coordination framework provides agent abstractions and coordination protocols for supporting dynamic composition and coordination of Grid applications. Experimental evaluations of these prototype implementations demonstrate the flexibility, scalability and effectiveness of the infrastructure, as well as its ability to support complex coordination requirements of Grid applications.

1.4 Contributions

This research investigates coordination infrastructures for addressing the heterogeneity, dynamism, scalability, and uncertainty of the Grid environments. The key contribution of this work is that it lays out a conceptual architecture model and provides a practical implementation of a coordination infrastructure that facilitates scalable, robust, and efficient interaction and communication for Grid applications. The main components of this research include:

- Design of the Comet conceptual architecture model, which provides a global virtual shared-space abstraction that can be associatively accessed by all system peers. Comet adapts the Squid information discovery scheme and employs a locality preserving mapping to map the multi-dimensional information space used by the coordinating entities to the one-dimensional peer node index space. The resulting shared-space space maintains content locality and guarantees that content-based information queries are delivered with bounded costs. The Comet

architecture model provides separated abstractions to address the communication, coordination, and application programming concerns.

- Design and develop Comet decentralized coordination infrastructure to demonstrate the conceptual model. Each peer node in Comet consists of 3 layers: the communication layer which provides associative messaging services and manages system dynamism using a self-organizing overlay; the coordination layer which implements Linda-like coordination primitives, by which all peers can associatively access the space without knowing the physical location or identifiers of the hosts; and the application layer which provides programming mechanisms to enable application formulations and executions.

- Develop application programming systems using Comet for solving practical problems. Two developed prototype systems include CometG computational infrastructure that extends Desktop Grid to support parallel asynchronous applications and Rudder coordination framework that provides software agents and coordination protocols to address dynamic composition of application workflows.

- Deploy and evaluate Comet, Rudder, and CometG on wide-area Grid testbed and Rutgers University campus networks. The experiments evaluate the performance of Comet coordination primitives, the Rudder coordination protocols, and the CometG computation system. The experimental results demonstrate the scalability and efficiency of these systems as well as the feasibility of using Comet to support wide-area Grid deployment.

1.5 Thesis Outline

The rest of the thesis is organized as follows.

Chapter 2 gives an overview of shared-space coordination model and existing coordination systems as well as their applications. It discusses the limitations of these existing systems with respect to the requirements of Grid computing. Furthermore, this chapter compares the key differences between Comet with the related systems.

Chapter 3 presents the Comet conceptual architecture model and the development of Comet infrastructure. The Comet model employs fully decentralized architecture and provides a global virtual shared-space abstraction that can be associatively accessed by all peers in the system. The Comet infrastructure implements and demonstrates the conceptual model. The design, implementation, deployment, and evaluation of Comet are presented.

Chapter 4 describes the CometG computational infrastructure, which extends Desktop Grid for parallel asynchronous applications. The effectiveness of CometG has been illustrated using parallel asynchronous iterative computations and parallel asynchronous replica exchange simulations. This chapter presents the implementation, deployment, and evaluation of CometG as well as CometG-based applications. Experimental results demonstrate the efficiency and scalability of CometG and its ability to support wide-area deployments of Desktop Grid applications based on parallel asynchronous algorithms.

Chapter 5 presents the Rudder coordination framework, which provides agent abstractions and coordination protocols for supporting dynamic composition and interaction of Grid applications. It describes the Rudder agents, the agent coordination protocols, and the Rudder-based dynamic application workflow compositions. This chapter also describes the implementation, deployment, and evaluation of Rudder.

Chapter 6 concludes the thesis and gives out future research directions.

Chapter 2

Background and Related Work

This chapter describes the shared-space coordination model and investigates existing shared-space based coordination systems and their applications. The existing shared-space coordination systems are described based on their target applications and computing environments. A comparison of Comet with the related existing systems is presented.

2.1 Shared-space Coordination Model

The shared-space coordination model was made popular by Linda [31], which defines a centralized tuple space as a shared message repository to exploit generative communication [40] model. The key attributes of Linda included:

(i) Asynchronous communication that decouples senders and receivers in space and time. An inserted tuple will exist independently in the tuple space until it is explicitly removed by a receiver and tuples are equally accessible to all receivers but bound to none.

(ii) An associative multicast medium through which multiple receivers can read a tuple written by a single sender using a pattern-matching mechanism instead of the name and location of the producer.

(iii) A small set of operators (write, read, and remove) providing a simple and uniform interface to the tuple space.

In Linda, a tuple is an ordered sequence of typed fields and a tuple space is a multiset of tuples that can be accessed concurrently by several processes using simple primitives. Tuples are inserted into the tuple space by executing the *out(t)* operation, extracted using the destructive primitive *in($\bar{t}$)* and read using the non-destructive primitive *rd($\bar{t}$)*,

where t is the tuple and $\bar{t}$ is a template that matches the tuple. If multiple tuples match a template, one tuple will be nondeterministically returned. Both *in* and *rd* are blocking operations. The template $\bar{t}$ may contain wildcards, denoted by "?", which is matched against actual values in a tuple during the associative matching process. For example, a tuple $\langle \text{“}task\text{”}, 12 \rangle$ will be matched by the template $\langle \text{“}task\text{”}, ?Interger \rangle$. Details can be found in [31].

The shared-space model provides a very flexible and powerful mechanism for supporting extremely dynamic coordination patterns. In this model, processes interact using an associative shared tuple space. The message sender formulates the message as a tuple and places it into the tuple space. The receiver(s) can associatively look up relevant tuples using pattern-matching on the tuple fields. This associative asynchronous communication model automatically supports dynamic communication and interaction between the coordinating entities. Recently shared-space coordination model has been adopted to build coordination infrastructures to address the challenges in different computing environments as described below.

2.2 Shared-space Coordination Infrastructures and Applications

Several research projects and commercial products have successfully adopted the shared-space coordination model to construct coordination systems. Examples include JavaSpace (Sun, 2000) [36], TSpaces (IBM, 1998) [52], MARS [29], XMARS [30], XMLSpaces [88], TuCSoN [67], and MARS [29]. These systems have enhanced the original Linda model with language expressiveness, control flexibility, and implementation architectures to support applications on various computing environments. These systems are summarized in Table 2.1 and described below.

Several projects employ shared-space coordination model and Java technology to support interactive Internet distributed applications. These systems provide a tuple space embedded in the Java run-time environment and offer loosely coupled communication with the paradigm of distributed shared memory. JavaSpaces [36] depends on the Jini services, where all operations are invoked using a local smart proxy and Java

Table 2.1: Shared-space coordination infrastructures for various computing environments.

System	**Java-based distributed computing**
JavaSpaces [36]	JavaSpaces provides a Java-based distributed computing environment. Each JavaSpace is provided by a single server, which can be either local or remote. JavaSpaces operations are invoked using Java Remote Method Invocation (RMI).
TSpaces [52]	TSpaces acts as a communication buffer with database capabilities. It offers applications a number of services, including group communication, database access, URL-based file transfer, and event notification. The TSpaces platform is implemented using a centralized architecture.
PageSpace [32]	PageSpace introduces the notion of active Web pages, which are capable of executing code, coordinates the interactions between these active pages. It implements a Java tuple space shared between different processes. The tuple space can be local or remote and employs a client-server architecture.
System	**Mobile agent computing**
MARS [29]	MARS provides local tuple spaces for mobile agent communication and interaction. The tuple space is enhanced with reactive programming capabilities, which allow an agent to respond to actions performed on the tuple space. The MARS agents can only coordinate with other agents that reside on the same node.
TuCSoN [67]	TuCSoN introduces the notion of a programmable data space, which provides the possibility to program the way in which the coordination medium reacts to the execution of coordination primitives. The implementation of TuCSoN employs a centralized architecture.
System	**Peer-to-peer computing**
PeerSpace [27]	PeerSpace provides a JXTA-based distributed peer-to-peer data space. Each peer node maintains a list of the neighbors, which are the tuple spaces known to the node. The tuple distribution and search is achieved by permitting the interactions among neighbor data spaces.
PeerWare [35]	PeerWare realizes a virtual global space using a forest of tree data structure, in which each peer holds its own data structure expressed in terms of a tree of documents. PeerWare also supports event-based publish/subscribe primitives. However, the information about the location of system components is required by the primitives.
System	**Wireless computing**
Lime [63]	Lime implements a virtual global space on top of shared local spaces, which are located on a group of physical mobile hosts. Lime supports reactive programming and provides distributed transactions. Its coordination primitives require the location information of the mobile hosts.
TOTA [59]	TOTA relies on spatially distributed tuples, which are injected in the network and propagated according to application-specific patterns. The tuple propagation mechanism can not guarantee the tuple lookup operation.

Remote Method Invocation (RMI). Although JavaSpaces can provide a flexible second-tier of "middleware", it is not database-centric and has no security model. TSpaces [52] acts as a network communication buffer with database capabilities. It targets the communications between applications and devices in a network of heterogeneous computers and operating systems. PageSpace [32] aims to support distributed applications which require active processing elements. It introduces the notion of active Web pages which are capable of executing code. In common, these Java-based tuple space systems can operate across a range of hardware platforms and operating systems. However, the centralized architectures limit their scalability for supporting large scale applications.

Some projects have employed shared-space infrastructures for mobile agent computing. These infrastructures, exemplified by MARS [29] and TuCSoN [67], enhance tuple spaces with reactive programming which allows a mobile agent to respond to actions performed on the tuple space. MARS [29] provides a local tuple space at each node, which is accessed by agents residing on it. An agent can only coordinate with other agents that reside on the same node. Agent migration is required for inter-node communication. TuCSoN [67] introduces the notion of programmable data space, which enables the tuple space reacts to the executions of the coordination primitives. These spaces are still based on centralized architecture and specifically designed for mobile agent computing environments.

Shared-space systems, such as PeerSpace [27] and PeerWare [35], have been proposed to address the dynamism and complexities of resource coordination in peer-to-peer environments. PeerSpace [27] builds on the JXTA peer-to-peer technology. In PeerSpace, each peer node maintains a list of neighbors and interacts with its neighbor nodes to distribute the tuples, which has no guaranteed costs and scalability. PeerWare [35] adopts a global virtual tree data structure to address large system scales, in which each peer shares a tree of documents with the other peers. PeerWare also provides publish/subscribe primitives to support event-based communication. However, the coordination primitives of PeerWare requires the location information of system components (nodes, hosts or agents), which is hard to maintain in Grid environments.

Several coordination middlewares, e.g., Lime [63] and TOTA [59], employed the

shared-space model to address the coordinating requirements among mobile devices. Lime [63] utilizes logically mobile agents running on physically mobile hosts to offer a global shared space that supports reactive programming and event notifications. Lime enforces distributed transactions with strong atomicity, which presents a scalability problem. Moreover, the Lime coordination primitives require the host location information. TOTA [59] relies on spatially distributed tuples, which are injected in the network and propagated accordingly to specific patterns. The tuple propagation patterns are dynamically reshaped by the TOTA middleware to implicitly reflect network and application dynamics. Each TOTA node holds references to a limited set of neighboring nodes and the system structure is dynamically maintained and updated by the nodes. However, the tuple distributed mechanism of TOTA can not guarantee the tuple lookup operations.

This research investigates Comet, a decentralized shared-space based coordination infrastructure for Grid environments. Comet provides a global virtual shared-space that can be associatively accessed by all peers in the system, and access is independent of the physical location of the tuples or identifiers of the host. Different from the centralized tuple space systems described above, Comet employs a fully decentralized architecture and a distributed hash scheme to achieve scalable data distribution performance. The two systems most related to this research are Lime and PeerWare, which build on the concept of Global Virtual Data Structures (GVDS). Lime [63] is designed for mobile environments. It exploits a flat data structure that transiently builds share spaces upon a set of hosts. PeerWare [35] realizes a forest of trees, composed of nodes and documents, which are contributed by each peer. However, Lime and PeerWare implicitly employ the *context-aware* programming style where information about the location of system components (e.g., nodes, hosts or agents) is required by the coordination primitives. However, maintaining such a global knowledge about location in large and highly dynamic distributed systems is infeasible. In contrast, we use the *context-transparent* approach and realize a GVDS as an associative distributed hash table, where all operations only use tuple content and are independent of the current state of the system and the mapping of content to these peers.

Chapter 3

Comet, A Scalable Decentralized Coordination Infrastructure for Grid Environments

This chapter presents the conceptual architecture model and implementation of Comet, a scalable decentralized coordination infrastructure. The Comet conceptual architecture model is based on a global virtual shared-space constructed from a semantic information space that is used by entities for coordination and communication. Comet adapts Squid [79] information discovery scheme to deterministically map the information space onto the dynamic set of peer nodes in the Grid system. The resulting structure is a locality preserving semantic distributed hash table (DHT) on top of a self-organizing structured overlay. The resulting decentralized tuple space maintains content locality and guarantees that content-based tuple queries, using flexible content descriptors in the form of keywords, partial keywords and wildcards, are delivered with bounded costs. The Comet space can be associatively accessed by all system peers without requiring the location information of tuples and host identifiers. The Comet also provides transient spaces that enable applications to explicitly exploit context locality.

3.1 The Comet Coordination Architecture

The Comet is constructed from a semantic multi-dimensional information space defined by the coordinated entities. This information space is deterministically mapped onto a dynamic set of peer nodes in the system using a locality preserving mapping. Comet is composed of layered abstractions prompted by a fundamental separation of communication and coordination concerns. A schematic overview of the system architecture is shown in Figure 3.1. The communication layer provides scalable content-based messaging and manages system heterogeneity and dynamism. The coordination

layer provides Linda-like associative primitives and supports a shared-space coordination model. Dynamically constructed transient spaces are also supported in Comet to allow the applications explicitly exploit context locality for improving system performance. The application layer provides programming abstractions and mechanisms to enable application formulation and execution.

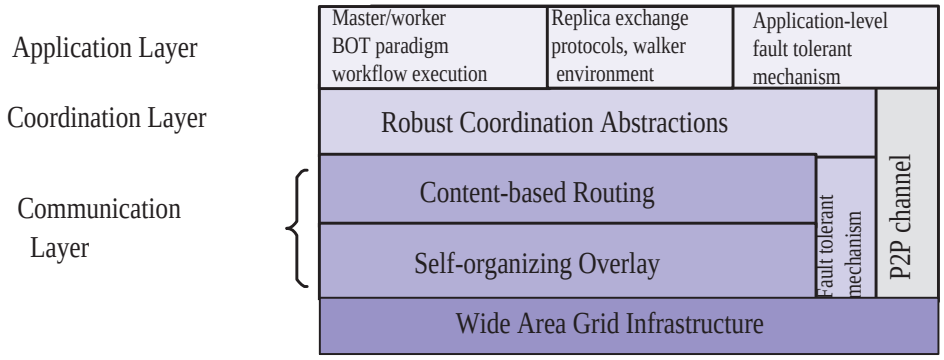


Figure 3.1: A schematic overview of the Comet.

3.1.1 Tuples and Template Tuples

In Comet, a tuple is a simple XML string, where the first element is the tuple's tag and is followed by an ordered list of elements containing the tuple's fields. Each field has a name followed by its value. The tag, field names, and values must be actual data for a tuple and can contain wildcards ("*") for a template tuple. This lightweight format is flexible enough to represent the information for a wide range of applications, and can support rich matching relationships [88]. Further, the cross-platform nature of XML makes this format suitable for information exchange in distributed heterogeneous environments.

A tuple in Comet can be retrieved if it exactly or approximately matches a template tuple. Exact matching requires the tag and field names of the template tuple to be specified without any wildcard, as in Linda. However, this strict matching pattern must be relaxed in highly dynamic environments, since applications (e.g., service discovery) may not know exact tuple structures. Comet supports tuple retrievals with incomplete

structure information using approximate matching, which only requires the tag of the template tuple be specified using a keyword or a partial keyword. Examples are shown in Figure 3.2. In this figure, tuple (a) tagged "contact" has fields "name, phone, email, dep" with values "Smith, 7324451000, smith@gmail.com, ece", can be retrieved using tuple template (b) or (c).

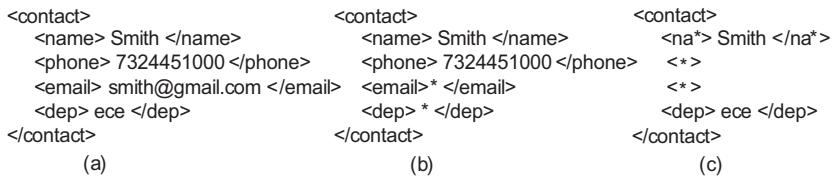


Figure 3.2: Examples of tuples in Comet.

3.1.2 Tuple Distribution and Retrieval

Comet adapts Squid information discovery scheme and employs the Hilbert Space-Filling Curve (SFC) [62] to map tuples from a semantic information space to a linear node index. The semantic information space, consisting of based-10 numbers and English words, is defined by application users. For example, a computational storage resource may belong to the 3D storage space with coordinates "space", "band width", and "cost". In Comet, each tuple is associated with k keywords selected from its tag and field names, which are the **keys** of a tuple. For example, the keys of tuple (a) in Figure 3.2 can be "name, phone" in a 2D student information space. Tuples are local in the information space if their keys are lexicographically close, or if they have common keywords. The selection of keys can be specified by the applications.

A Hilbert SFC is a locality preserving continuous mapping from a k-dimensional (kD) space to a 1D space. It is locality preserving in that points that are close on the curve are mapped from close points in the kD space. The Hilbert curve readily extends to any number of dimensions. Its locality preserving property enables the tuple space to maintain content locality in the index space. In Comet, the peer nodes form a 1-dimensional overlay, which is indexed by a Hilbert SFC. Applying the Hilbert mapping, the tuples are mapped from the multi-dimensional information space to the linear peer index space. As a result, the Comet uses the Hilbert SFC constructs the distribute

hash table (DHT) for tuple distribution and lookup. If the keys of a tuple only include complete keywords, the tuple is mapped as a point in the information space and located on at most one node. If its keys consist of partial keywords, wildcards, or ranges, the tuple identifies a region in the information space. This region is mapped to a collection of segments on the SFC and corresponds to a set of points in the index space. Each node stores the keys that map to the segment of the curve between itself and the predecessor node. For example, as shown in Figure 3.3, five nodes (with id shown in solid circle) are indexed using SFC from 0 to 63, the tuple defined as the point (2,1) is mapped to index 7 on the SFC and corresponds to node 13, and the tuple defined as the region (2-3,1-5) is mapped to 2 segments on the SFC and corresponds to nodes 13 and 32.

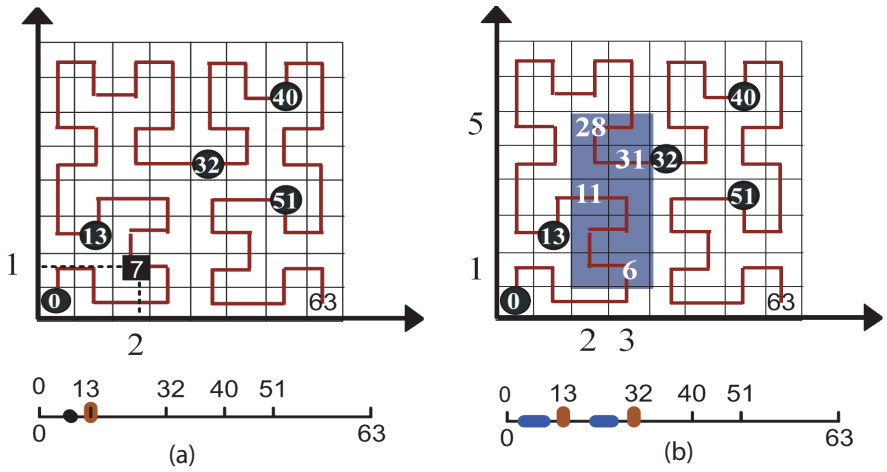


Figure 3.3: Examples of mapping tuples from 2D information space to 1D index space.

3.1.3 The Communication Abstraction

The Comet communication abstraction provides an associative messaging service, which guarantees that content-based information queries, specified using flexible content descriptors, are served with bounded costs. This abstraction provides a single operator: **post** ($\mathcal{M}$). The message $\mathcal{M}$ consists of (1) a semantic selector that is flexibly defined using keywords, partial keywords and wildcards from the information space, and specifies a region in this space, and (2) a payload consisting of the data and operation to be performed at the destinations. The operations can be "store", "delete", "read",

etc. This operator forwards the message $\mathcal{M}$ to all destination nodes containing content that lies in the region defined by the semantic selection, i.e., that matches the selector. Note that the resolution of this operator depends on the current information existing in this system. Further note that, unlike low-level messaging protocols that send/receive messages to/from specific destinations, the destination(s) in this case are dynamically determined based on the state of the system.

Tuple space operations can directly build on this operator. For example, the operator **out**(t) can be implemented as **post** ($\mathcal{M}$), where the semantic selector is defined by the keys of the tuple and the payload includes the tuple and the action "store". The **rd**($\bar{t}$) can be similarly implemented using the template $\bar{t}$ to define the semantic selector and the payload including the action "read". Note that the **post** can return one, all or some of the matched tuples.

3.1.4 The Coordination Abstraction

Coordination Primitives

The coordination layer provides tuple operation primitives to support the shared-space based coordination model. The basic coordination primitives are listed below:

- **out**(ts, t): a non-blocking operation that inserts tuple t into space ts.
- **in**(ts, $\bar{t}$): a blocking operation that removes a tuple t matching template $\bar{t}$ from the space ts and returns it.
- **rd**(ts, $\bar{t}$): a blocking operation that returns a tuple t matching template $\bar{t}$ from the space ts. The tuple is not removed from the space.

The above primitives retain the Linda semantics, i.e., if multiple matching tuples are found, one of them is arbitrarily returned (and removed). Furthermore, advanced primitives, such as **rdall(ts,** $\bar{t}$**)** that returns all the matching tuples, can also be supported to address specific application requirements.

Transient Spaces

The above uniform operators do not distinguish between local and remote spaces, and consequently Comet is naturally suitable for context-transparent applications. However, this abstraction does not maintain geographic locality between peer nodes, and may have a detrimental effect on the efficiency of the applications imposing context-awareness, e.g., mobile applications. These applications require that context locality be maintained in addition to content locality, i.e., they impose requirements for context-awareness. To address this issue, Comet supports dynamically constructed transient spaces that have a specific scope definition (e.g., within the same geographical region or the same physical subnet). The global space is accessible to all peer nodes and acts as the default coordination platform. Membership and authentication mechanisms are adopted to restrict access to the transient spaces. The structure of the transient space is exactly the same as the global space. An application can switch between spaces at runtime and can simultaneously use multiple spaces.

3.1.5 The Application Abstraction

The Comet application layer provides programming abstractions to support the coordinations and computations in Grid environments. Specifically, it provides the CometG computational system for parallel asynchronous Grid computations, and Rudder coordination framework for composing component-based applications. These systems are introduced below and described in more detail in Chapter 4 and Chapter 5.

The CometG provides coordination space abstractions and programming modules to support master-worker/Bag-Of-Task(BOT) parallel formulations of asynchronous computations. The coordination spaces support dynamic task distribution and management as well as inter-task communications. The programming modules support application-specific computational components that define computations and provide task retrieval/submission mechanisms as well as interaction/negotiation protocols. Specifically, prototypes of parallel asynchronous iterative application and parallel asynchronous replica exchange simulations have been developed to demonstrate the CometG system.

The Rudder framework provides agent abstractions and coordination protocols for supporting dynamic composition and coordination of applications. It defines agents to represent and control discrete software elements. An element can be an application, service or resource unit (e.g., computer, instrument, and data store). Agents advertise element capabilities, provide uniform access to elements, configure an element based on its execution context, and control an element execution. Transiently generated composition agents dynamically discover and compose elements to realize applications.

The following section describes the implementation of the communication and coordination layers of Comet and the experimental evaluation of the Comet system using a campus-network at Rutgers University and the PlanetLab [7] testbed. The implementation, evaluation, and illustrative applications of CometG and Rudder are presented in Chapter 4 and Chapter 5.

3.2 Implementation of the Comet Infrastructure

The Comet infrastructure is implemented on Project JXTA [73], a platform independent peer-to-peer framework, where peers can self-organize into peergroups, discover resources, and communicate with each other. Comet is provided as a JXTA peergroup service and can be concurrently exploited by multiple applications. The JXTA peergroup provides a secure environment where only member peers can access the service instances running on peers of the group. If any peer fails, the collective peergroup service is not affected and the service is still available from other peer members. Transient spaces are also implemented based on the peergroup concept.

3.2.1 The Communication Layer

The communication layer provides an associative communication service and guarantees that content-based messages, specified using flexible content descriptors, are served with bounded cost. The two main components of this layer are a structured one-dimensional self-organizing overlay and a content-based routing engine.

The overlay is composed of peer nodes, which may be any node in the system (e.g.,

gateways, access points, message relay nodes, servers, or end-user computers). The peer nodes can join or leave the network at any time. While the Comet architecture is based on a structured overlay, it is not tied to any specific overlay topology. The overlay provides a simple operator to the layers above: **lookup**(identifier). Given an identifier, this operator locates the node that is responsible for it, i.e., the node with an identifier that is the closest identifier greater than or equal to the queried identifier. In the current implementation of Comet, we use Chord [84], which has a ring topology, primarily due to its guaranteed performance, efficient adaptation as nodes join and leave the system, and the simplicity of its implementation. In principle, the Chord overly could be replaced by other structured overlays.

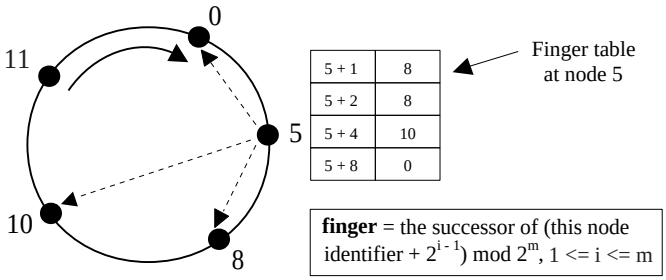


Figure 3.4: Example of the Chord overlay.

In the Chord overlay network, each node has a unique identifier ranging from 0 to 2^m-1. These identifiers are arranged as a circle modulo 2^m, where each node maintains information about its successor and predecessor. In addition, each node maintains the identifiers of (at most) m other neighbors, called fingers, in a finger table. The i^{th} finger node is the first node that succeeds the current node by at least 2^{i-1}, where $1 \leq i \leq m$. The finger table is used for efficient routing. An example of a Chord overlay network with 5 nodes is shown in Figure 3.4. Each node constructs its finger table when it joins the overlay and finger tables are updated any time a node joins or leaves the system. The lookup algorithm in Chord enables the efficient data routing with O(Log N) cost [84], where N is the number of nodes in the system.

The routing engine provides a decentralized information discovery and associative messaging service. It implements the Hilbert SFC mapping to effectively map a multi-dimensional information space to a peer index space and to the current peer nodes in

the system, which form a structured overlay. The resulting peer-to-peer information system supports flexible content-based routing and complex queries containing partial keywords, wildcards, or ranges, and guarantees that all existing data elements that match a query will be found. The engine has a single operator: **post**(keys, data), where keys form a semantic selector and data is the message payload provided by the layers above. If the keys only include complete keywords, the engine routes the message using the overlay lookup mechanism. If the keys contain partial keywords or wildcards, the message identifies a region in the information space. The region is defined as a cluster [62] if the SFC enters and exits it once. A cluster might be mapped to one or more adjacent overlay nodes and one node can store multiple clusters. While resolving a query by sending a message to each cluster is not scalable, since the number of nodes can be very high. In Comet, the optimization scheme [79] provided by Squid is used to improve the scalability. This scheme embeds the query tree into the overlay network and distributes the cluster refinement at each node to allow the query to be resolved in a decentralized manner.

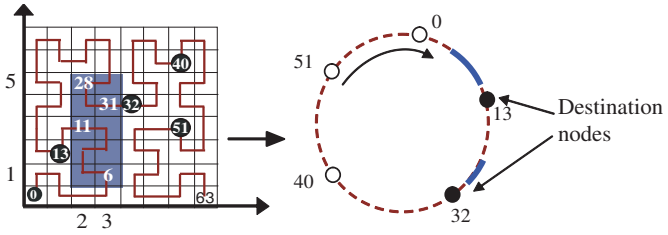


Figure 3.5: Example of content-based routing in Comet.

Content-based routing using Comet is achieved as follows. SFCs are used to generate a 1D index space from the multi-dimensional keyword space. A simple query composed of only keywords is mapped to a point on the SFC. A complex query containing partial keywords or wildcards is mapped to regions in the keyword space and to corresponding clusters (segments of the curve) on the SFC. The 1D index space generated from the entire information space is mapped onto the 1D identifier space used by the overlay network. As a result, using the SFC mapping any query request can be resolved. For example, the tuple in Figure 3.3(a) defined as a point (2,1) in a 2D space is mapped to index 7 on SFC and routed on Chord (an overlay with 5 nodes and an identifier

space from 0 to 2^6-1) to node 13, the successor of the index 7. Similarly, the tuple in Figure 3.3(b) defined as a region (2-3,1-5) in the 2D space is mapped to 2 segments on the SFC, and routed to node 13 and node 32 on Chord, is shown in Figure 3.5.

3.2.2 The Coordination Layer

The coordination layer implements tuple operation primitives to support the shared-space based coordination model, and realizes transient spaces to allow applications explicitly exploit context locality. The main components of this layer include a data repository for storing pending requests, and retrieving tuples, a flexible matching engine, and a message dispatcher that interfaces with the communication layer to convert the coordination primitives to messaging operations and vice versa. As described in Section 3.1, tuples are represented as simple XML strings as they provide small-sized flexible formats that are suitable for efficient information exchange in distributed heterogeneous environments. The data repository stores tuples as DOM level 2 objects [2]. It employs a hash structure to perform associative lookup in a constant time in memory.

Implementation of Coordination Primitives

In Comet, it is assumed that all peer nodes agree on the structure and dimension of the information space for an application. The **out**, **rd** and **in** operators are implemented using the **post** operator provided by the communication layer. Using the keys associated with each tuple, each tuple is routed to an overlay peer node and a template tuple may be routed to a set of nodes. If the keys of a template are completely specified (only contain complete keywords), the template will be routed to the node that would store matched tuples using the overlay lookup protocol. The tuple distribution and exact retrieval processes using **out** and **in/rd** operators are illustrated in Figure 3.6 (a) and (b) respectively.

The process illustrated in the above figures consists of the following steps: (1) Keywords are extracted from the tuple and used to create the keys for the **post** operation. The payload of the message includes the tuple data and the coordination operation. (2) The query engine uses the SFC mapping to identify the indices corresponding to the

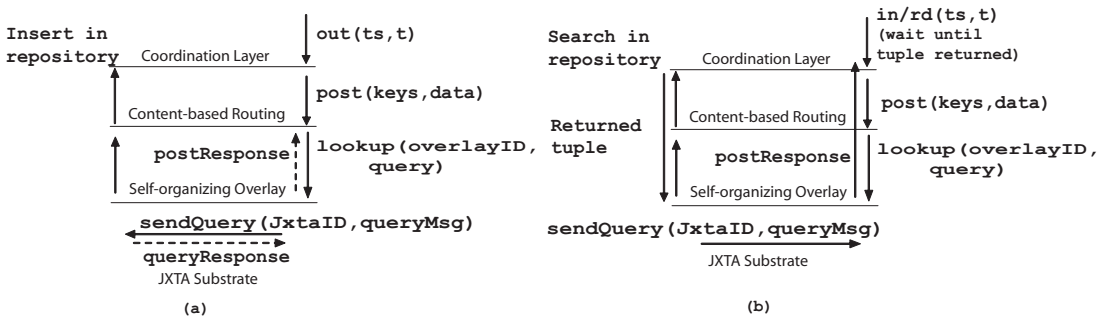


Figure 3.6: Comet tuple space operation: (a) Tuple distribution using the **out** operator. (b) Exact tuple retrieval using the **in/rd** operator.

keys and the corresponding peer id(s), to which these identifiers are mapped. (3) The overlay **lookup** operation is used to route the tuple to the appropriate peer nodes. This operator maps the logical peer identifer to the JxtaID of the node, and forwards the tuple using the JXTA Resolver Protocol. The **out** operation only returns after receiving the Resolver Query Response from the destination to guarantee tuple delivery. In case of **in** and **rd** operations, templates are routed to the peer nodes in a similar manner. The **in** and **rd** operations block until a matched tuple is returned by the destination.

The approximate retrieval process is similar. A retrieval request may be sent to multiple nodes in this case, and each of them may return a matching tuple. However, the **in** and **rd** operations are implemented differently. In case of **rd**, the first tuple that is returned is accepted and forwarded to the application, and subsequent tuples returned are ignored. In case of an **in** operation, one of the matching tuple must be deleted and this is coordinated by the requesting node. For each matching tuple found, the node with the matching tuple sends it to the requesting node and waits for a delete confirmation. The requesting node responds with a delete confirmation to the first matching tuple that it receives and responds with an ignore message to all other returned tuples.

Implementation of Transient Spaces

Comet supports dynamically constructed transient spaces, which are implemented based on JXTA peergroup. Figure 3.7 illustrates the implementation and operation of transient spaces in Comet. In the figure, the global space includes five nodes and a transient space is constructed using two nodes, 5 and 10. The transient space interface provides operations for creating, switching between and destroying spaces. The creating process consists of *coordination service initialization*, in which a peergroup is created and instantiated at each involved peer node, and *finger table stabilization*, in which the peer node joins the group. Peer nodes can belong to several tuple spaces and the switching operation enables an application to dynamically switch between coordination services associated with these spaces. To destroy the transient space, each peer node in the peergroup stops the associated service and deletes its local instance of the space.

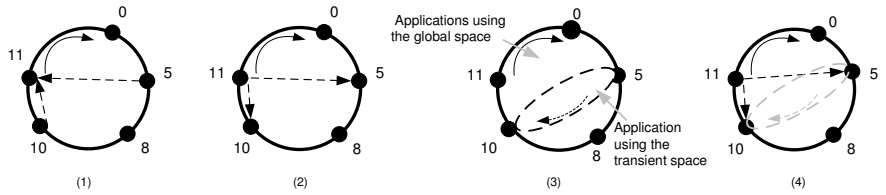


Figure 3.7: Transient tuple spaces in Comet. (1) All the peer nodes join in the global space. Peer node 5 and peer node 10 add requests to a create transient space. (2) The requests are initialized, peer node 5 and 10 are notified that a new space need to be created. (3) Peer node 5 and peer node 10 get the message, execute the join protocol to instantiate a new space service. (4) Once the application is done, the leave protocol will be executed to notify the nodes and destroy the transient space.

3.3 System Operation and Evaluation

The overall operation of Comet consists of two phases: *bootstrap* and *running*. The *bootstrap* phase is used to setup a coordination group. During this phase, peer nodes join the Comet JXTA peergroup and exchange messages with the rest of the group. Each joining peer attempts to discover an existing peer in the system and to construct the overlay and setup its routing table. It also sends discovery messages to the group. If the message is unanswered after a pre-defined time interval (in the order of seconds), the peer assumes that it is the first one in the system. If a peer responds to the message,

the joining peer queries this bootstrapping peer according to the join protocol of the overlay, and updates routing tables in the overlay to reflect the join.

The *running* phase consists of *stabilization* and *user* modes. In the *stabilization* mode, peer nodes manage the structure of the overlay. In this mode, peer nodes respond to periodic queries from other peers to ensure that routing tables are up-to-date and to verify that other peer nodes in the group have not failed or left the system. In the *user* mode, peer nodes participate in user applications. In this mode, application developers can configure the system, setup application parameters such as relevant spaces, and initiate the application program.

3.3.1 Experimental Evaluation

The Comet infrastructure has been deployed on a wide-area environment using the PlanetLab [7] testbed, as well as a campus network at Rutgers University. The objective of the experiments presented below is to evaluate system performance and scalability, and demonstrate the feasibility of using Comet to support wide-area deployments.

The first sets of experiments were conducted on the Rutgers campus network. Each machine was a peer node in the Comet overlay and the machines formed a single Comet peergroup. This set of experiments evaluated the costs of basic tuple insertion and exact retrieval operations. The tuples in the experiments were fixed at 200 bytes. A ping-pong like process was used in the experiments, in which an application process inserted a tuple into the space using the **out** operator, read the same tuple using the **rd** operator, and deleted it using the **in** operator. In these experiments, the **out** and exact matching **in/rd** operators used a 3D information space. For an **out** operation, the measured time corresponded to the time interval between when the tuple was posted into the space and when the response from the destination was received, i.e., the time between Post and PostResponse in Figure 3.6(a). For an **in** or **rd** operation, the measured time was the time interval between when the template was posted into the space and when the matching tuple was returned to the application, assuming that a matching tuple existed in the space, i.e., the time between Post and receiving the tuple in Figure 3.6(b). This time included the time for routing the template, matching tuples in the repository, and

returning the matching tuple. The average performances were measured for different system sizes. Figure 3.8 plots the average measured performance and shows that the system scales well with increasing number of peer nodes. When the number of peer nodes increases 32 times, i.e., from 2 to 64, the average round trip time increases only about 1.5 times, due to the logarithmic complexity of the routing algorithm of the Chord overlay. **rd** and **in** operations exhibit similar performance, as seen in the figure. To further study the **in/rd** operator, the average time for **in/rd** was measured using increasing number of tuples. Figure 3.9 shows that the performance of **in/rd** is largely independent of the number of tuples in the system, where the average time is approximately 105ms for the scenario with 2000 to 12000 tuples.

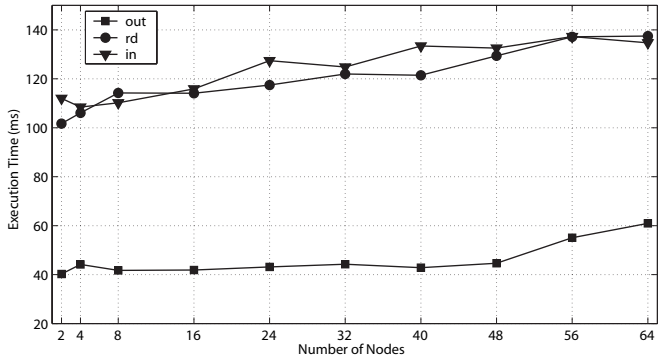


Figure 3.8: Average time for **out**, **in**, and **rd** operators for increasing system sizes on Rutgers campus network.

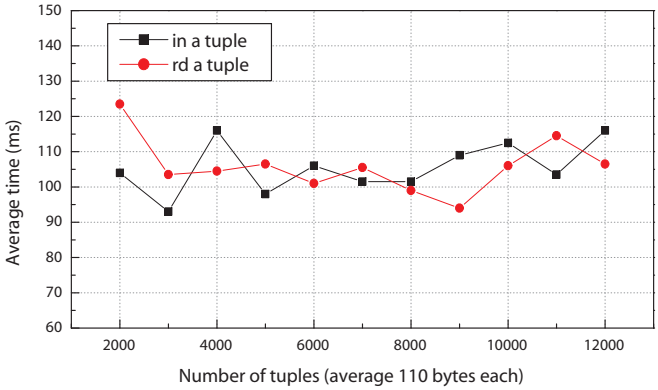


Figure 3.9: Average time for **in** and **rd** operations with increasing number of tuples on Rutgers campus network. System size fixed at 4 nodes.

The second set of experiments evaluated the performance of Comet using PlanetLab [7] (a large scale heterogeneous distributed environment composed of interconnected sites with various resources). Once again, each machine ran an instance of Comet, serving as a peer node in the overlay. Since PlanetLab nodes are distributed across over the globe, communication latencies can vary significantly over time and with node location. As a result, the following methods were adopted in the experiments: (1) In each experiment, at least one node was selected from each continent, including Asia, Europe, Australia, and North America. (2) Nodes randomly joined the Comet system during the bootstrap phase, resulting in a different physical construction of the ring overlay in each run. (3) The experiments were conducted at different time of the day during a 4-week period, and each experiment ran continuously for about 3 hours. The experiments measured the average run time for each of the primitives and abstractions provided by Comet, including the tuple insertion operation **out**, exact retrieval operation **rd/in**, approximate retrieval operation **rd/in**, and transient space performances.

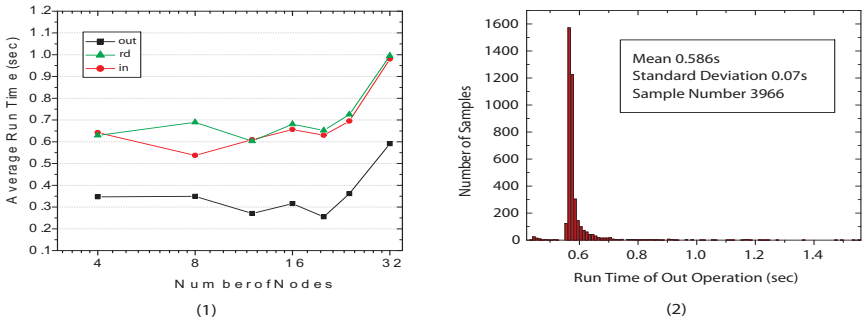


Figure 3.10: Average time for **out**, **rd**, and **in** operators for increasing system sizes on PlanetLab. (1) Average run time for **out** and exact **rd/in** operators. (2) Histogram of **out** operator run time on 32 nodes.

First, the same ping-pong process was used to test the basic tuple operators. The average run time of the operations for increasing number of nodes is plotted in Figure 3.10 (1). Each value plotted is averaged over 3 experiments. The X-axis is plotted on a logarithm scale with base 2 of the number of used nodes. The Y-axis is the average run time in seconds. As seen in Figure 3.10 (1), the operation time increases by a factor of about 2 when the system size grows by a factor of 8. As mentioned before,

this is expected as the complexity of the underlying Chord lookup protocol is O(Log N), where N is the number of nodes in the system. Note that the costs plotted in the figure have the same order as the cost of the Chord lookup protocol presented in [84]. This protocol has been shown to scale to 10^4 nodes using simulations in [84]. As a result, we believe that Comet will exhibit similar scalability. Figure 3.10 (2) plots the histogram of **out** operation run time on 32 nodes in one experiment. This plot shows that 95% of the samples are less than 0.65 second, one standard deviation away from the mean.

Second, we evaluated the approximate **rd/in** operations using 2D and 3D information spaces. The templates used in the experiments are: (Case 1) one keyword and at least one partial keyword, e.g., (contact, na*), (contact, na*, *), and (Case 2) one keyword or partial keyword, e.g., (con*, *), (contact, *, *). For a **in/rd** operation, the measured time is the time interval between when the template is posted into the space and when the matched tuple is returned. The average time over about 500 operations is shown in Figure 3.11. The figure shows that while the operation time increases with the node number and the space dimensions, and the rate of increase is much smaller than the rate of increase of the system size. Further, simulations have shown the number of query processing nodes is a small fraction of the total nodes [79], i.e., below 8% in 2D and 20% in 3D for (Case 2) when system size increases from 1000 to 5000 nodes. As a result, we conclude that the Comet approximate retrieval operations can also scale to large systems.

Finally, we evaluated the transient spaces in Comet. The time for creating a transient space is about 100 seconds per node, including the time to initialize a peergroup and execute the join protocol. The insertion and exact retrieval performance for a 4-node space scenario varies with geographical locations, as shown in Figure 3.12. The average run time for the case where the nodes are located within a LAN at Rutgers University is about 4 times smaller than the case where these nodes are within America, and 6 times smaller than the case where the nodes are distributed across 4 continents. Creation is a one-time cost, and from the results it can be concluded that using transient spaces can improve system performance in terms of operation latencies.

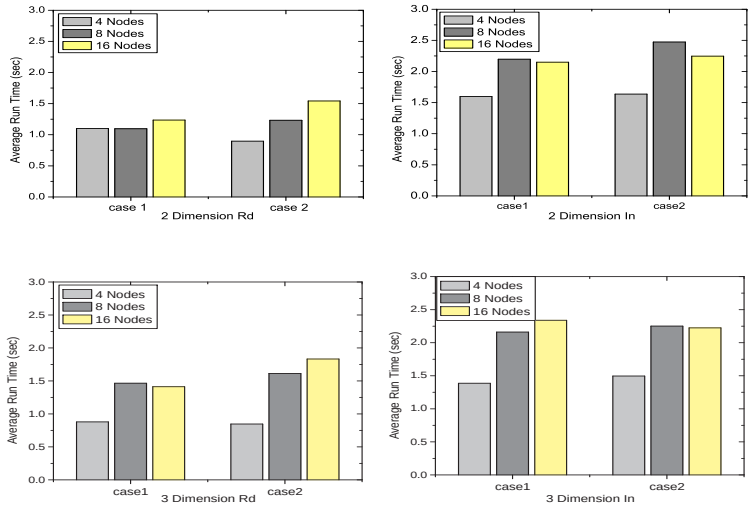


Figure 3.11: Average run time for approximate **rd/in** operations on PlanetLab.

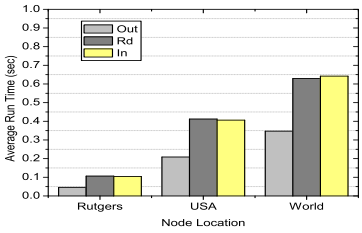


Figure 3.12: The system performance using transient spaces with different geographical scopes.

This chapter presented the conceptual architecture model and implementation of Comet coordination infrastructure. The Comet provides a global virtual shared-space constructed from a semantic information space that is deterministically mapped onto the dynamic set of peer nodes in the Grid system. The Comet space can be associatively accessed by all system peers without requiring the location information of tuples and host identifiers. The Comet also provides transient spaces that enable applications to explicitly exploit context locality. The Comet was deployed and evaluated using Rutgers campus networks and PlanetLab testbed. The experiments using campus network show that the Comet scales well with increasing number of peer nodes and increasing number of tuples. The experiments on PlanetLab demonstrate the feasibility of using Comet to support wide-area deployments and its capability to address the heterogeneity, dynamism, and uncertainty of Grid.

Chapter 4

CometG, A Decentralized Computational Infrastructure for Grid-based Parallel Asynchronous Applications

This chapter presents CometG [56], a decentralized (peer-to-peer) computational infrastructure that extends Desktop Grid environments to support parallel asynchronous formulations of general iterative computation and replica exchange simulations. CometG builds on top of the Comet tuple space and provides efficient, scalable communication and coordination abstractions. Furthermore, it provides programming abstractions to implement robust parallel asynchronous applications. Two prototype systems have been implemented and evaluated on top of CometG to demonstrate the effectiveness of this infrastructure. The design and implementation of CometG as well as the two prototype applications are presented as below.

4.1 CometG Computational Infrastructure for Grid-based Asynchronous Applications

4.1.1 Motivation

Grid computing, based on the aggregation of large numbers of independent hardware, software and information resources spanning multiple organizations, is rapidly emerging as the dominant paradigm for distributed problem solving for a wide range of application domains. Complementary to Grid virtual organizations, Desktop Grids [48] leverage Internet connected computers to support large computations. Desktop Grid systems have been successfully used to address large applications in science and engineering with significant computational requirements, including global climate predication (Climatprediction.net) [1], molecular sequence analysis (Folding@Home) [3],

protein structure prediction (Predictor@Home) [6], search for extraterrestrial intelligence (SETI@Home) [8], gravitational wave detection (Einstein@Home), and cosmic rays study (XtremWeb) [9].

While the successes of the above applications do demonstrate the potential of Desktop Grids, current implementations are limited to *embarrassingly parallel* [90] applications based on the Bag-Of-Task (BOT) paradigm, where the individual tasks are independent and do not require inter-task communications. As a result, these implementations cannot support more general scientific and engineering applications, such as those based on parallel iterative computations and replica exchange simulations for structural biology and drug design, as the parallel formulations of these applications require synchronization and inter-task communications. Parallel asynchronous formulations of computation algorithms relax synchronization and communication requirements, and can tolerate heterogeneous computation powers and unreliable communication channels. These formulations have been proposed to extend Desktop Grids beyond *embarrassingly parallel* applications and support parallel applications, such as computing the lowest eigenvalue and eigenvector of stochastic matrices for Google pageranks [77] and solving linear systems [25].

Parallel asynchronous applications can definitely benefit from the potentially large numbers of processors available on Grid. However, developing and executing Grid-based implementations requires addressing the complexity of the Grid environment, including its heterogeneity in computational, storage and communication capabilities, its dynamism and its unreliability. While some Java-based platform independent communication libraries, such as mpiJava [5] and JavaPVM [92] and been developed to support parallel Grid applications, these libraries have targeted relatively tightly coupled, similarly configured, and simultaneously available Grid environments such as multi-site inter-connected clusters [64, 44]. Consequently, supporting the synchronization and communication requirements of general scientific application in heterogeneous, dynamic and unreliable wide-area environment continues to present significant difficulties.

Clearly, this development complexity of Grid applications must be abstracted from

the application scientists/engineers and effectively addressed by a computational infrastructure. Such an infrastructure should support *dynamic* and *anonymous* task management, allowing application execution to be independent of system configuration and promoting the simplicity and convenience of the Bag-Of-Task (BOT) paradigm. Further, it should provide appropriate coordination and communication mechanisms to support dynamic dependencies and interactions. Specifically, the task coordination and communication mechanisms should be: (1) *asynchronous* to enable decoupled (in time and space) and dynamic task allocation and inter-processor communication; (2) *associative* to allow interactions to be anonymous and based on content rather than defined in terms of addresses or names of end-points', since maintaining common knowledge about names and addresses in dynamic Grid environments is infeasible and can pose security risks [25]; (3) *scalable* to address increasing system size (number of nodes) and application problem size; and (4) *failure-resilient* to reduce the loss of application computational effort when system or application failures occur. The tuple space paradigm, which supports an asynchronous associative communication model and provides simple programming abstractions, presents an attractive approach for addressing the issues outlined above.

The tuple space paradigm, made popular by Linda [40], addresses many of the requirements outlined above. Its key features include: asynchronous communication that decouples senders and receivers in space and time; an associative multicast medium through which multiple receivers can read a tuple written by a single sender using pattern-matching mechanisms instead of names and locations; and a small set of operators (write, read, and remove) providing a simple and uniform interface to the tuple space. Additionally, resilience to process failures can be simply provided by a stable tuple space [19] where failed processes can be recovered on any host. Further, tuple space paradigm naturally supports BOT solutions for parallel applications using the master worker model - the master inserts task tuples into the space and collects result tuples, and the workers extract task tuples from the space and insert result tuples. While sufficiently scalable distributed tuple space implementation, where the tuple retrieval performance is proportional to at least the logarithm of the system size [65],

can effectively address the requirements outlined above, such implementations in Grid environments remain a challenge. Further, the original Linda model must be enhanced and customized to support specific asynchronous algorithms. First, tuple insertion and retrieval are unordered and non-deterministic. As a result, the programmer must implement "latest version" retrieval semantics (e.g., by adding a sequence number field to the tuple) and guarantee processing of all tasks (e.g., by using a global counter tuple). Second, associative communications implemented using pattern-matching mechanisms are inherently inefficient for large data transfers [83]. This inefficiency is further amplified if the tuple delivery requires multiple routing steps, as large message sizes increase transmission time as well as probability of failure at each step. This research investigates CometG, a tuple space based computational infrastructure, which addresses the issues discussed above and extends Desktop Grid environments to support parallel asynchronous applications.

4.1.2 CometG Architecture

The CometG computational infrastructure builds on Comet, described in Chapter 3, a scalable decentralized tuple space that spans the nodes of the Desktop Grid. The Comet space is essentially a global virtual shared-space constructed from the semantic information space used by entities for coordination and communication. This information space is deterministically mapped, using a locality preserving mapping Hilbert Space Filling Curve (SFC), onto the dynamic set of peer nodes in the Grid system. The resulting structure is a locality preserving semantic Distributed Hash Table (DHT) built on top of a self-organizing structured overlay. CometG provides abstractions and mechanisms on top of Comet to construct services for dynamic and anonymous task distribution, task execution, decoupled communication and data exchange required by the application.

The architecture of CometG consists of 3 key layers. The communication layer provides scalable content-based messaging services as well as channels for direct communication, and manages system heterogeneity and dynamism. The coordination layer provides Linda-like primitives and supports the tuple space coordination model. The

application layer provides abstractions and services for asynchronous computations, which are implemented using the communication and coordination layers. While the CometG architecture can support scalable tuple distribution, failure of nodes can result in tuple loss. This is addressed by the CometG application layer using timeout regeneration and checkpointing-restart mechanisms.

4.1.3 The Design and Implementation of CometG

As described in Chapter 3, the tuple space provided by Comet is a global virtual semantic shared-space constructed from the semantic information space used by the coordinating entities. The application computational entities employ this space to retrieve computation tasks and interact with each other. In CometG, an application defines a k-dimensional semantic information space with user specified dimensions and coordinates; and a computational task defines a tuple in this space. Note that it is assumed that the information space is known to participating nodes.

In CometG, a task tuple is implemented as a Comet tuple, i.e., a simple XML string, where the first element is the tuple's tag and is followed by an ordered list of elements containing the tuple's fields. Figure 4.1 show an example of tuples that match exactly. The task tuple in Figure 4.1(a), tagged "Task", has fields *BlockID, TotalBlocks, Partition, Solver, Precision, MaxIteration, MasterNetName* and *DataPort* with values *5, 10, strips, Jacobi, 0.0001, Inf, foo.cs.bar.edu, 9914* respectively, and can be retrieved using the template in Figure 4.1(b).

```
<Task>
   <BlockID> 5 </BlockID>
   <TotalBlocks> 10 </TotalBlocks>
   <Partition> strips </Partition>
   <Solver> Jacobi </Solver>
   <Precision> 0.0001 </Precision>
   <MaxIteration> Inf </MaxIteration>
   <MasterNetName> foo.cs. bar.edu
   </MasterNetName>
   <DataPort> 9914 </DataPort>
</Task>
```

(a)

```
<Task>
  <BlockID> * </BlockID>
  <TotalBlocks> * </TotalBlocks>
  <Partition> * </Partition>
  <Solver> * </Solver>
  <Precision>* </Precision>
  <MaxIteration> * </MaxIteration>
  <MasterNetName> * </MasterNetName>
  <DataPort> * </DataPort>
</Task>
```

(b)

Figure 4.1: An example of a tuple and a template in CometG: (a) A task tuple. (b) A task template.

The CometG communication layer provides an associative communication service

and guarantees that content-based messages, specified using flexible content descriptors, are served with bounded cost. This layer is implemented based on the communication layer of Comet and includes a content-based routing engine and the 1-dimensional structured self-organizing overlay. The routing engine implements the Hilbert SFC mapping and supports flexible content-based routing and complex querying using partial keywords, wildcards, or ranges. It also guarantees that all peer nodes with data elements that match a query/message will be located. The overlay is composed of peer nodes, which may be any node in the Desktop Grid system (e.g., end-user computers, servers, or message relay nodes). The coordination layer supports a tuple space coordination model and provides the following primitives:

- *out(ts, t)*: a non-blocking operation that inserts tuple t into space ts.
- *in(ts, $\bar{t}$, timeout)*: a blocking operation that removes a tuple t matching template $\bar{t}$ from the space ts and returns it. If no matching tuple is found, the calling process blocks until a matching tuple is inserted or the *timeout* expires. In the latter case, *null* is returned.
- *rd(ts, $\bar{t}$, timeout)*: a blocking operation that returns a tuple t matching template $\bar{t}$ from the space ts. If no matching tuple is found, the calling process blocks until a matching tuple is inserted or the *timeout* expires. In the latter case, *null* is returned. This method performs exactly like the *In* operation except that the tuple is not removed from the space.

The implementation and operation of the above tuple insertion and retrieval primitives are based on the content-based routing provided by Comet, which are illustrated in Figures 4.2 and 4.3 respectively.

4.2 Parallel Asynchronous Iterative Computations in Grid Environments

Parallel asynchronous formulations of iterative algorithms [23, 38] relax synchronization and communication requirements, and can tolerate heterogeneous computation powers and unreliable communication channels. These formulations have been proposed to

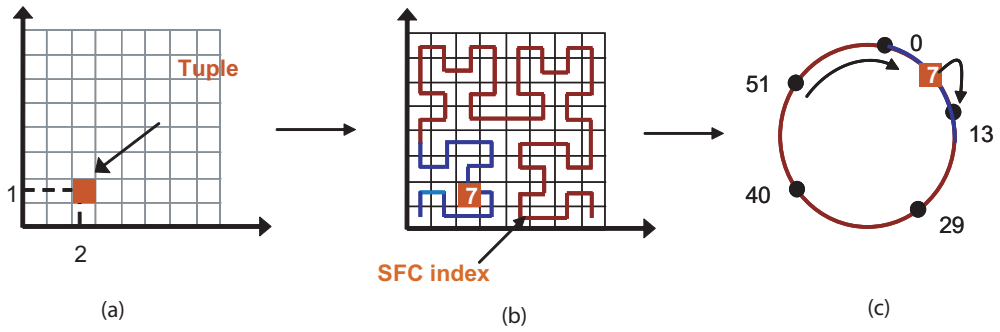


Figure 4.2: Example of tuple insertion in CometG: (a) a tuple is represented in a 2D keyword space, as the point (2, 1); (b) the point (2, 1) is mapped to index 7 using the Hilbert SFC; (c) the tuple is inserted at node 13 (the successor of SFC index 7).

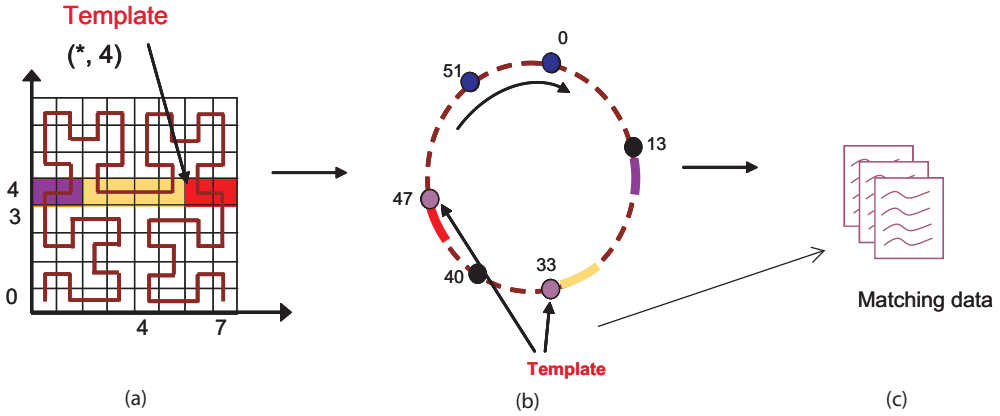


Figure 4.3: Example of tuple retrieval in CometG: (a) the template defines a rectangular region in the 2D space consisting of 3 clusters; (b) the nodes that store the clusters are queried; (c) results of the query are sent to the requesting node.

extend Desktop Grids beyond *embarrassingly parallel* applications and support parallel iterative applications, such as computing the lowest eigenvalue and eigenvector of stochastic matrices for Google pageranks [77] and solving linear systems [25]. However, current implementations of these algorithms are limited to tightly coupled clusters and local area networks, and scalable wide-area implementations remain a challenge.

Asynchronous Iterative Algorithms and Applications

Iterative algorithms are generally of the form: $x^{k+1} = f(x^k), k = 0, 1, ..,$ where x^0 is given, x^k is an n-dimensional vector, and f is a function from $R^n \rightarrow R^n$. The sequence x^k generated by the above iteration converges to some x^*, and if f is continuous then x^* is a fixed point of f. These algorithms are typically parallelized using the block-decomposition paradigm, where the x^k is decomposed as m components and f is partitioned conformally. The entire problem can be solved in parallel by m processors and the iteration vector at each step is $x^k = [x_1^k, x_2^k, ..., x_m^k]$, each component of which can be processed by a single processor.

Iterative algorithms can be categorized as *synchronous* or *asynchronous* based on their requirements for global data synchronization. *Synchronous* iterative algorithms have an implicit barrier at the end of each iteration step, and require that all communications be completed and all messages become available before the next iteration starts. *Asynchronous* iterative algorithms relax this requirement for global synchronization, and allow processors to continue computing using only partial information from other processors. This allows these algorithms to tolerate variances in computational power and communication delay, which are typical in Grid environments. Note that, as expected, the convergence of asynchronous iterative algorithms is delayed due to the unsynchronized data. However, in spite of this, these algorithms have the potential of outperforming synchronous algorithms as they avoid synchronization overheads, which can be significant in Grid environments.

Potential applications of parallel asynchronous iterative computation span a range of scientific and engineering disciplines, such as high-performance linear algebra and optimization problems. Examples include: (1) computation of eigen-systems, which

are used in the study of nuclear reactor dynamics, dynamic finite element analysis of structural models, and the next generation particle accelerators [93]; (2) solution of large sparse linear systems of equations obtained from the discretization of partial differential equations (PDE) [49], which are used for aircraft simulation, computer graphics, weather prediction, fluid flow, gravitational fields, and electromagnetic field description; and (3) variational inequalities that can be viewed as generalization of both constrained optimization problems and systems of equations, which are used as models for equilibrium studies ranging from economics to traffic engineering [23].

Note that while there has been significant work on parallel asynchronous iterative computations in recent years, these efforts have focused on algorithmic and implementation issues such as convergence rate, termination detection, and load balancing [16, 17, 20, 22, 23]. This research leverages these efforts and focuses on the development and execution of applications based on these algorithms on Desktop Grid environments with Internet-scale connectivity.

Existing Parallel Asynchronous Iterative Computational Systems

Related research efforts that focus on supporting asynchronous parallel applications in peer-to-peer systems include P^3 [66], Jace [18], and parallel iterative computing using associative broadcast [25]. P^3 proposes a peer-to-peer network platform for high performance parallel computing in an Internet-based environment. It uses a distributed file system for inter-process communication and synchronization. Scalability in P^3 is achieved using dynamic load balancing between computing nodes, P2P communication and dynamically changing sets of manager nodes. However, the P^3 network implementation is still ongoing to the best of our knowledge.

Jace [18] is a Java based distributed programming environment designed specifically for distributed asynchronous iterative computations. It provides a parallel virtual machine to implement computing tasks using message passing. However, it does not allow nodes to dynamically join and/or leave the system, and the application data is statically partitioned across and stored at the participating nodes. Further, fault-tolerance issues are not addressed by Jace.

Parallel iterative computing using associative broadcast [25] is most closely related to the research presented in this thesis. In [25], the programming models and implementation issues for executing parallel computations on Desktop Grids are discussed, and combining associative interactions with parallel asynchronous iterative algorithms is proposed as an effective approach. Specifically, asynchronous data communications between the parallel computation tasks is achieved using the associative broadcast mechanism. The implementation of associative broadcast, however, does not currently address scalability to Grid environments. Further, this system does not support dynamic task distribution. CometG implements a scalable tuple space to support the associative communication model, and also provides support for dynamic task distribution and fault-tolerance.

4.2.1 CometG-based Parallel Asynchronous Iterative Computations

Programming Abstractions

The CometG application layer provides coordination space abstractions and programming modules to support master-worker/BOT parallel formulations of asynchronous iterative computations. Specifically, two customized coordination spaces, *TaskSpace* and *BorderSpace*, are defined and implemented separately. *TaskSpace* stores task tuples representing application tasks and specifying the masters that are responsible for the tasks. This space implements First-In-First-Out (FIFO) semantics for tuple and template operations, and provides a queue abstraction for task distribution and management. An example of a task tuple is shown in Figure 4.1. *BorderSpace* is used for exchanging data tuples between neighboring tasks. This space enforces *over-write* semantics during tuple insertion, where tuples in the space always store the latest content, resulting the latest messaging semantics. A border tuple has a border id field and an associated binary data block. The data block is not used for content-based distribution, lookup, and pattern-matching.

The programming modules include masters and workers. A worker module contains an application-specific computational component that can locally compute a retrieved task. The worker uses the tuple space abstractions to retrieve tasks and exchange

borders. Task retrieval consists of two steps - removing a task description from the *TaskSpace* and downloading the task data from the corresponding master. A master module is responsible for partitioning the application data, generating tasks, collecting results, and terminating the application when it completes. CometG provides single master mode as well as multiple master mode. In multiple master mode, hierarchical or decentralized termination algorithms [17] are supported based on the organization of the masters. A master module has five components:

- The *configuration manager thread*, which reads the application configuration (including whether it is a single or a member of multiple master organization) and the data partitioning strategy.
- The *task generator thread*, which generates application tasks based on the partitioning strategy, encapsulates task descriptions as tuples and inserts the task tuples into *TaskSpace*.
- The *data transfer thread*, which uses the direct communication channel to process requests for task data retrieval and for result submission from workers, as well as coordination messages (e.g., "convergence" message) between masters.
- The *terminator thread*, which checks for convergence among tasks that the master is responsible for, monitors convergence messages from other masters, and terminates when overall convergence is achieved.
- The *task monitor*, which maintains a table of tasks the master is responsible for, and records the current state of the tasks in this table. The state of a task can be *generated*, *retrieved*, *computing*, *submitting* or *completed*.

Supporting Large Application/System Scales

CometG supports large application/system scales using multiple coordination groups. A coordination group includes one *TaskSpace*, one *BorderSpace*, and a group of masters and workers. A group can support multiple applications with logically separate semantic spaces. An application can also span multiple groups, each of which handles a part of the application. The application is hierarchically partitioned, first across coordination groups, and then across masters within each coordination group. Task with communication dependencies should be mapped to the same coordination group if possible as

communications across groups can be expensive. Workers within a coordination group communicate using the shared *BorderSpace*. Masters within and across coordination group communicate using direct communication channels.

Using coordination groups thus distributes the load of *TaskSpace* and reduces the size of *BorderSpace*, effectively improving the scalability of the system. Nevertheless, it may not always be possible to partition the application to eliminate inter-group communications. However, as the number of these communications is relatively small, these communications can simply be ignored in the case of asynchronous applications. While ignoring them will effect convergence, we have observed that the improvement in overall application performance using this approach outweights these effects. In cases where the number of inter-group communications is large, or when task dependencies are complex, data exchange can be coordinated through a single node in each group [72].

Addressing Grid Unreliability

The CometG computational infrastructure provides application level fault tolerance mechanisms to address the unreliability inherent in Grid environments. These mechanisms assume a fail-stop failure model and timed communication behavior [34, 33]. Under these assumptions, possible failures include border tuple communication failure, master failure, and task loss. These failures are address below:

Border tuple communication failures are simply handled by *Rd* timeouts, due to the resilient nature of asynchronous algorithms. *Master failures* are handled using checkpoint-restart. The runtime system periodically checkpoints the local state of each master, including its task table and current intermediate results, to a stable storage. Users are currently responsible for the detecting the failure of a master node. When a master fails, users can recover its state from the stable storage and resume the computation. Finally, *task loss* is handled using timeout-regeneration and a retrieval-submission protocol. It is well known that detecting this kind of failure in tuple spaces is very difficult because there can be multiple reasons for the failure, including *TaskSpace* crashes, message losses, communication link failures, failures of workers with unfinished tasks, etc. In CometG, the loss of un-retrieved and retrieved tasks, are handled separately as

follows.

Un-retrieved task loss occurs only when the relevant *TaskSpace* node crashes since task tuple insertions are guaranteed. Masters can detect this failure using a keep-alive mechanism, and can handle it by regenerating unfinished tasks. The regenerated tasks will be deterministically routed to an operational *TaskSpace* node on the DHT due to the resilience of the overlay (e.g., the Chord routing around failure functionality [84]).

Retrieved task loss is detected using the task tables at the masters. Each task in the table is associated with a timer, which is initialized when the task is retrieved by a worker. If the results for a task are not returned before the timer expires, the task is considered as lost. The master regenerates the lost task and updates the task table. The value of the task timer depends on the computational requirements of the specific application as well as the current performance of the system. In CometG, this value is dynamically determined based on a user specified threshhold and the observed maximum task processing time, which is the time interval from when a task is retrieved to when the corresponding results are returned.

Note that task regeneration can lead to the problem of *duplicated tasks* where the same task may be allocated to multiple workers. This can be addressed using a simple retrieval-submission protocol where the master refuses all data transfer requests and result submissions for a task that it has tagged as completed in its task table. Further, redundancy in storage and routing can be embedded at the overlay layer as described in [78], where a group of nodes act as one CometG peer. However, the tuple space consistency problems as well as group synchronization issues, such as redundancy degree, group memberships, group communication protocols, etc. [34] must be considered.

4.2.2 Grid-based Parallel Asynchronous Iterative Applications Using CometG

This section illustrates the use of the CometG computational infrastructure to implement and execute a Grid-based PDE application. The application uses parallel asynchronous Jacobi iterations for solving the heat distribution problem [23]. In this illustrative application, the temperature at the edges of a square sheet are known, and

the temperature at a point in the interior surface of the sheet is computed based on the temperatures around it. The square sheet is discretized as a 2-dimensional grid and represented as a 2-dimensional array of points. In each iteration, the value of each point in the interior of the array is computed as an average of four points around it. The computation is repeated until the stop criterion is satisfied, i.e., the difference in temperature values at a point between iterations is less than a prescribed threshold, or the bound on the number of iterations is reached.

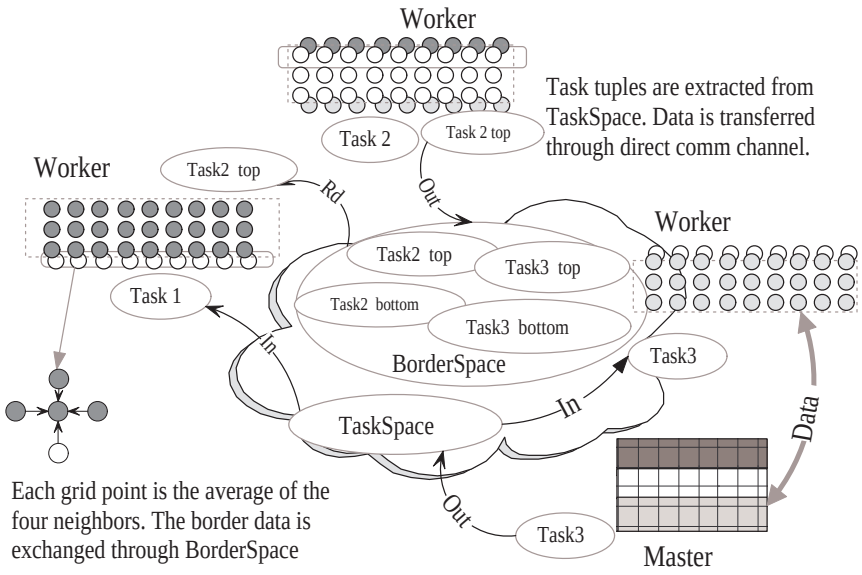


Figure 4.4: CometG-based implementation of the heat distribution problem using parallel asynchronous Jacobi iterations.

Assuming that the application uses strip partitioning, the grid points are divided into blocks of rows. Each block defines a task and is processed by one worker. Since each point needs its four immediate neighbors, each worker needs to exchange data in the rows at the top and bottom of the block with workers processing neighboring blocks. The workers assigned the top most and bottom most rows are exceptions and need to exchange data in only one row. A conceptual overview of the CometG based implementation of this application is shown in Figure 4.4.

Flow charts for the operation of master and worker nodes are presented in Figure 4.5, and are described below. Once a worker is initiated, it repeats the following steps until explicitly terminated: (1) extract a task tuple from *TaskSpace*, (2) read the required

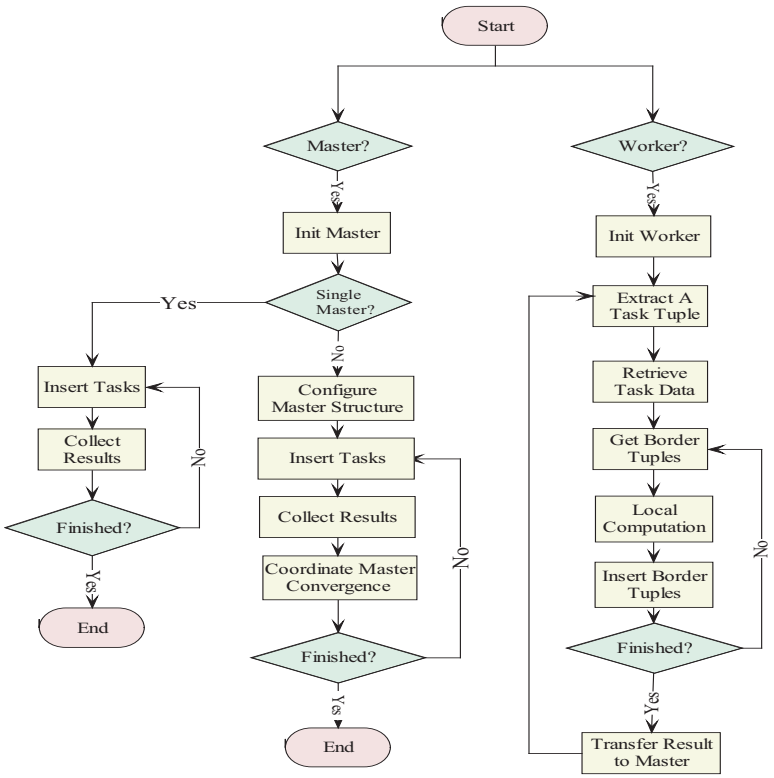


Figure 4.5: Operation of master and worker nodes for the CometG-based implementation of the heat distribution problem.

top and/or bottom border rows from *BorderSpace*, (3) locally compute temperature, (4) insert updated border rows into *BorderSpace*, (5) repeat steps (2)-(4) until the stop criterion specified in the task tuple is reached, and (6) send results to the master corresponding to the task using a direct communication channel.

When the master is launched, it uses user inputs to configure the application (e.g., setup the number of coordination group and master organization, etc.) and initiates the *BorderSpace*. If a single master is used, that master is responsible for the entire grid. The master first partitions the grid into blocks and inserts corresponding tasks into *TaskSpace*. When a task is assigned to a worker, the worker obtains task data from the master using the direct communication channel. When the task completes, the work submits the results to the master also using the direct communication channel. After all its tasks have completed, the master checks if the stop criterion is satisfied by the computed data, since the overall application may not satisfy the stop criterion even though each task locally satisfies its stop criterion. If the overall stop criterion is

not satisfied, the master repartitions the grid to create new tasks and inserts them into *TaskSpace*. This process constitutes one global application iteration, and is repeated until the overall stop criterion is satisfied, at which point, the master terminates the application.

If multiple masters are used, the user must define the organization of the masters, e.g., a hierarchical structure [18], and the termination detection algorithm to be used. The grid is uniformly partitioned across the masters in this case, and each master locally partitions its sub-grid into blocks and inserts corresponding tasks into *TaskSpace*. The operation at each master then proceeds as in the single master case described above. When a master detects local termination, it coordinates with the other masters to establish global convergence. In case of a hierarchical master organization, it sends a "converge" message up the hierarchy to the root node. If the master stays in a "converged" state, no further messages are sent, otherwise, a "diverge" message is sent to the root. The root node checks the messages received from all the masters at the end of each iteration, and if all of them are in the converged state for a specified number of iterations, it broadcasts a "stop" message to the masters, which causes them to terminate the application.

4.2.3 Experimental Evaluation

CometG and the PDE application have been deployed on a wide-area environment using PlanetLab [7] test bed, as well as a campus network at Rutgers. The objective of the experiments presented in this section is to evaluate and demonstrate system performance and scalability, its ability to tolerate faults, and its ability to support wide-area deployments of parallel asynchronous iterative applications. The experiments use a horizontal block partitioning strategy and vary the size of the problem as listed in Table 4.1. In the multiple master mode, a hierarchical organization of the masters was used and measurements were made at the root node. The different experiments and the results obtained are described below.

The first set of experiments were conducted on a Grid consisting of 70 heterogeneous Linux-based computers on the Rutgers campus network. Each machine was a peer node

in CometG overlay and the machines formed a single CometG group. The evaluation of basic tuple insertion and exact retrieval operations were presented in Chapter 3. The tuples in the experiments were fixed at 200 bytes, which is roughly equal to the size of a task tuple. Increasing the size of border tuples can cause message transmission delays. However, the message routing time remains the dominant factor as the system size increases. Note that the JXTA 2.3 Resolver Protocol used to implement CometG has been shown to effectively transfer message of size up to 128 Kbyte [15], which is sufficient for supporting border tuple communications for the current application.

The first experiment measured overall application performance using a problem of size 3000x3000 grid points and precision thresholds of 10^{-3}, 10^{-5}, and 10^{-7}. The grid was partitioned into 100 blocks and uniformly distributed across 10 master nodes. The masters were organized as a hierarchy with one root using the algorithm in [18]. All other nodes served as worker nodes, each hosting 2 worker instances. The total execution time is plotted in Figure 4.6. In this plot, the X-axis represents the number of workers plotted using a logarithmic scale with base 10. The plots show the overall application performance improvements and demonstrate that, as expected, the improvements are more significant when there is more computation (e.g., when the precision threshold is smaller).

The second experiment demonstrates the CometG fault tolerance mechanisms for handling task losses due to worker dynamism. The experiment was conducted on 32 machines and used a problem of size 2000x2000 grid points and a precision threshold of 10^{-5}. The grid was partitioned into 100 tasks distributed across 4 master nodes. The user defined task timeout threshhold was set to 50s. All the other nodes served as workers and hosted multiple worker instances. Tuple losses were simulated by having workers that have retrieved a task tuple fail with a probability of 25%. A global

Table 4.1: Problem sizes used in the experimental evaluation.

Problem Size	Partitions	Block Size	Border Tuple Size
2000x2000	100	0.32M	16.026K
3000x3000	100	0.7M	24.026K
8000x2000	200	0.64M	16.026K

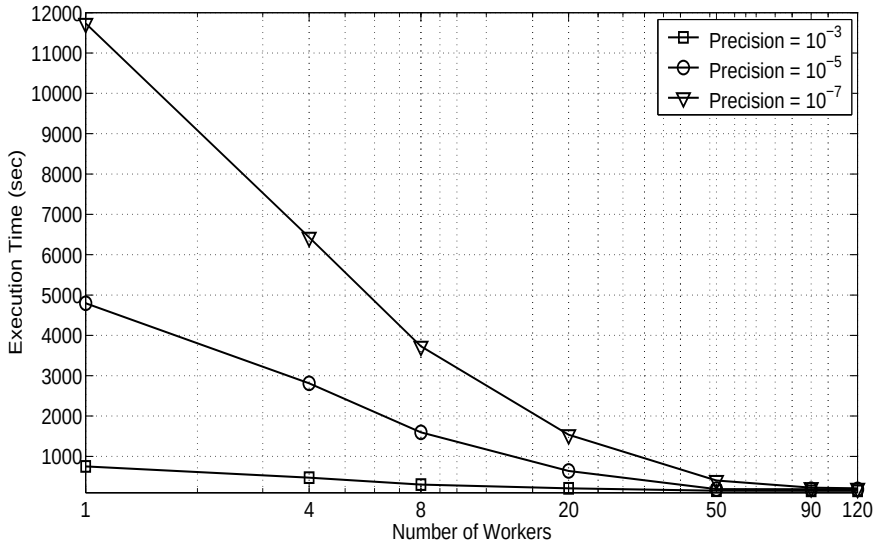


Figure 4.6: Overall application execution time for a problem size of 3000x3000 and 100 partitions on the Rutgers campus network.

monitor process was used to calculate the number of alive workers in the system each time worker was started or failed. In the experiment, 20 workers were initially started on randomly selected nodes. As the application progressed, workers failed and the lost task tuples were regenerated. Meanwhile, 20 new workers were started at 285s and 439s after the start of the application, at the rate of one worker every 3s. The results of this experiment are plotted in Figure 4.7. Figure 4.7 (a). plots the fluctuations in the number of workers during the lifetime of the application. Of the 100 total tasks in the application, 22% were regenerated once and 3% were regenerated twice due to worker failures. Figure 4.7 (b). illustrates the life-cycles of tasks including timeouts and the resulting task regenerations. For clarity, this figure only shows a subset of tasks with id between 80 and 90. Plots for other tasks are similar.

The second set of experiments were setup on the wide-area PlanetLab [7] test bed. PlanetLab is a large scale heterogeneous distributed environment composed of inter-connected sites on a global scale. The goal of the experiment is to demonstrate the ability of CometG to support application even in an unreliable and highly dynamic environments such as PlanetLab, which essentially represents an extreme case for a Desktop Grid environment. The CometG is currently deployed on 234 machines on PlanetLab, which have been used in the experiment presented below. In the experiment,

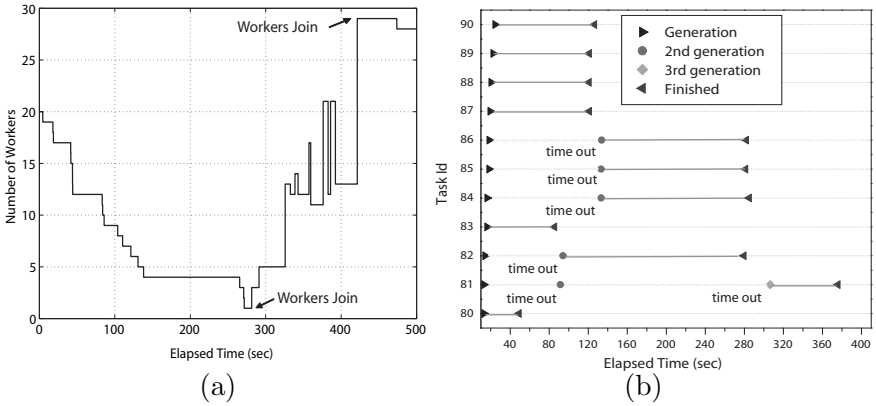


Figure 4.7: Evaluation of CometG fault tolerance mechanisms for tolerating loss of task tuples. (a) Fluctuations in the number of worker due to failures. (b) Life-cycles of tasks 80 through 90.

each machine ran an instance of the CometG stack, randomly joined the CometG overlay during bootstrap phase, and served as a master or worker node with one worker instance per node.

The experiment used a problem of size 8000x2000 and a precision of 10^{-5}. The problem was partitioned into 200 tasks, which were uniformly distributed and across 4 CometG coordination groups. Each group had about 60 peer nodes, of which 5 nodes acted as masters and others served as workers. The task timeout threshhold was set to 500s and the border tuple read timeout was set to 100s. The experiment was conducted on December 9, 2006, and lasted more than three hours, including the infrastructure setup, bootstrap, application deployment, configuration, and execution. The application terminated after two global iterations, during which multiple worker nodes left the system or failed and were handled by the CometG fault tolerance mechanisms. One master in coordination group 3 also failed and was restarted manually. The task tables of all the masters were collected and summarized in Figure 4.8 and Figure 4.9. Figure 4.8 separately plots the retrieval, computation, and result submission times for all the tasks for each of the two global iterations. The X-axis in these plots represents the task id, and the Y-axis represents the execution time of each phase. Note that the computation time for the second iteration is significantly smaller and the first, as expected. Figure 4.9 plots the total execution time for each iteration. The X-axis once again represents the task id. The variation in the execution time for different tasks

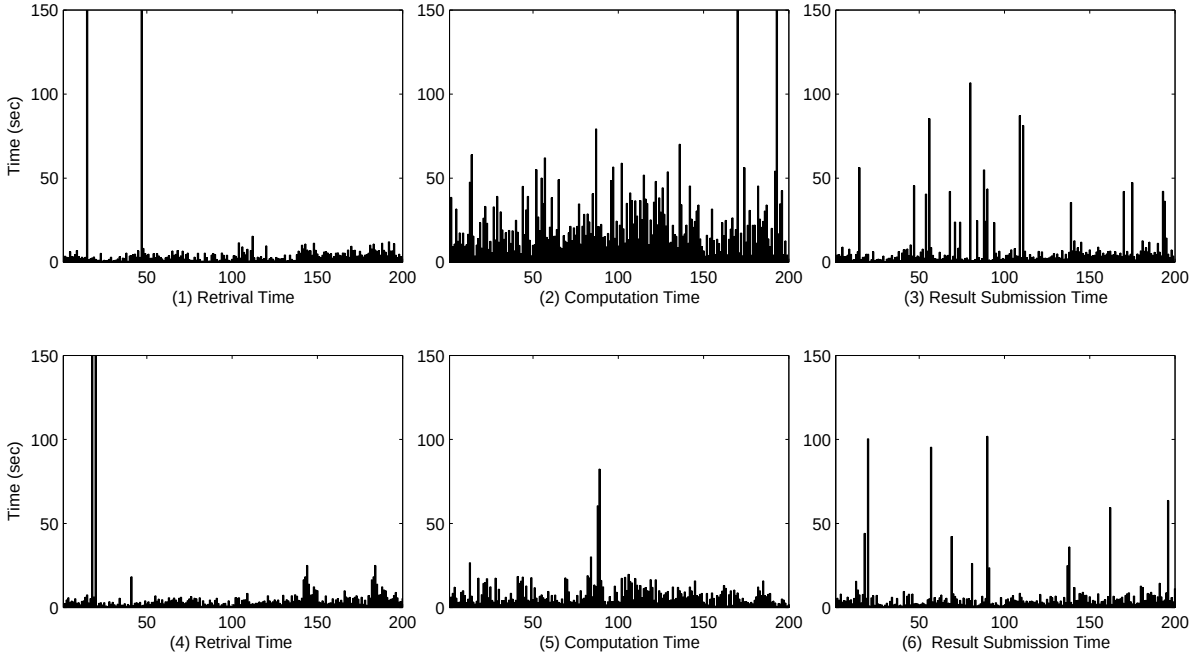


Figure 4.8: Execution time of each CometG phase on PlanetLab using a problem size of 8000x2000 and 200 tasks. (1)-(3)phases of the first global iteration. (4)-(5) phases of the second global iteration.

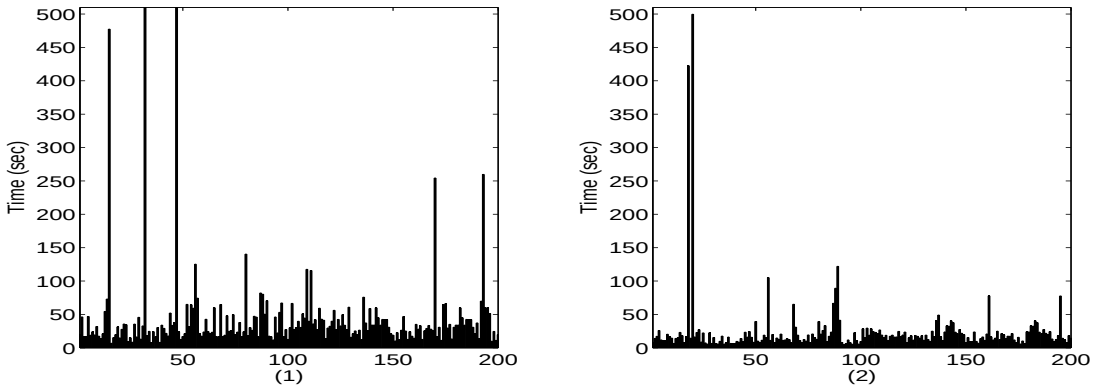


Figure 4.9: Overall execution time for two global iterations on PlanetLab using a problem size of 8000x2000 and 200 tasks. (1) Total execution time of the first global iteration. (2) Total execution time of the second global iteration.

illustrates the heterogeneity of the workers and the PlanetLab test bed. These experiments demonstrate that CometG system can effectively support parallel asynchronous iterative applications on an extreme case of a wide-area Desktop Grid environment with very high heterogeneity, dynamism, and uncertainty.

The experimental results presented in this section demonstrate both, the efficiency/scalability of CometG and its ability to support wide-area deployments of Desktop Grid applications based on parallel asynchronous iterative algorithms.

4.3 Asynchronous Replica Exchange for Grid-based Molecular Dynamics Applications

Replica exchange is a powerful sampling algorithm that preserves canonical distributions and allows for efficient crossing of high energy barriers that separate thermodynamically stable states. In this algorithm, several copies or replicas, of the system of interest are simulated in parallel at different temperatures using "walkers". These walkers occasionally swap temperatures and other parameters to allow them to bypass enthalpic barriers by moving to a higher temperature. The replica exchange algorithm has several advantages over formulations based on constant temperature, and has the potential for significantly impacting the fields of structural biology and drug design - specifically, the problems of structure based drug design and the study of the molecular basis of human diseases associated with protein misfolding.

While these replica exchange simulations can definitely benefit from the potentially large numbers of processors available in a Desktop Grid environment, general formulations of the replica exchange algorithm require complex coordination and communication patterns between the walkers. Coupled with the complexity of the Grid environment, including its scale, its heterogeneity in computational, storage and communication capabilities, its dynamism and its unreliability, Grid-based replica exchange simulations present significant challenges. It is probably for this reason that, to the best of our knowledge, all the current parallel/distributed implementations of replica

exchange simulations in use by the structural biology community target small homogenous systems. Further, these implementations are based on a simplified formulation of the algorithm that limits the potential power of the technique in two important aspects: (1) the only parameter exchanged between the replicas is the temperature of each replica, and (2) the exchanges occur in a centralized and totally synchronous manner, and only between replicas with adjacent temperatures. The former limits the effectiveness of the method and impedes temperature mixing, while the latter limits its scalability to a small number of homogeneous and relatively tightly coupled processors.

Clearly, the complexity of developing Grid-based replica exchange must be abstracted from the application scientists/engineers and effectively addressed by a computational infrastructure. Such an infrastructure should support dynamic walker management and efficient, robust and scalable exchanges to enable large scale simulations of the structure, function, folding, and dynamics of proteins. This thesis presents the design, implementation and evaluation of such a computational infrastructure. It consists of two components: (1) an asynchronous formulation of replica exchange that is more suited to Grid environments and (2) a Grid-based asynchronous replica exchange engine (GARE). The asynchronous replica exchange formulation builds on our initial algorithm proposed in project Salsa [51] and has the following characteristics: (1) the exchanged parameters and the overall parameter ranges used by the simulation are determined at the beginning of the simulation and are known to all the walkers; (2) the parameters assigned to a walker only change when the walker performs an exchange; (3) exchanges can occur between walkers on different nodes; and (4) the walkers can dynamically join or leave the system. The first two observations allow individual walkers to locally determine the ranges of interest and enable exchange decisions to be made in a decentralized and decoupled manner. The third allows actual exchanges to occur between pairs of walkers in parallel. The last observation enables the replica exchange to deal with the environment and system dynamism.

The Grid-based asynchronous replica exchange engine builds on CometG and extends it to provide the abstractions and mechanisms required by asynchronous replica exchange, including mechanisms for dynamic and anonymous task distribution, task

coordination and execution, decoupled communication and data exchange. It provides a virtual shared space abstraction that can be associatively accessed by all walkers without knowledge of the physical locations of the hosts over which the space is distributed. The walkers can use this space to dynamically discover exchange partners, negotiate with them, and exchange data. Walkers periodically post temperature ranges that are of current interest for exchange to the space. If this range overlaps with the range of interest posted by another walker, an exchange can occur. The actual exchange is then negotiated and completed by the individual walkers in a peer-to-peer manner. As a result, exchanges are decoupled, dynamically and asynchronously determined, and not limited to neighboring temperatures.

4.3.1 Parallel Replica Exchange for Structural Biology and Drug Design

The sequencing of the human genome, in conjunction with rapidly increasing efforts in structural genomics, is producing an explosion in the number of available high resolution protein structures. Molecular simulations of protein structural changes and drug binding to proteins depend critically on the design of highly efficient algorithms to search over the very rough energy landscapes which govern protein folding and binding. Scalable parallel replica exchange implementations can potentially address these molecular search problems and can significantly impact structure based drug design applications.

The Replica Exchange Algorithm

Replica exchange is an advanced canonical conformational sampling algorithm designed to help overcome the sampling problem encountered in biomolecular simulations. The method had been proposed independently on several occasions in various disciplines [86, 41, 60, 45]. In this method, several copies, or replicas, of the system of interest are simulated in parallel at different temperatures using walkers. These walkers occasionally swap temperatures based on a proposal probability that maintains detailed balance [42]. Note that general formulations of replica exchange simulations

allow walkers to exchange multiple parameters, e.g., temperature plus energy. However current implementations only exchange temperatures.

These exchanges allow individual replicas to bypass enthalpic barriers by moving to high temperatures. A parallel version of this algorithm was proposed by Hukushima and Nemoto [45]. The replica exchange algorithm is easy to implement and does not require time-consuming preparatory procedures. Further, it can decrease the sampling time by factors of 20 or more, as compared to constant temperature molecular dynamics when applied to peptides at room temperature [76]. Details of the algorithm can be found in [42] and application examples can be found in [85, 13].

The molecular dynamics replica exchange canonical sampling method has been implemented in the IMPACT (Integrated Modeling Program, Applied Chemical Theory) molecular mechanics program [46], and is the molecular simulation method used in this research. The implementation follows the approach proposed by Sugita and Okamoto [85]. The method consists of running a series of simulations at fixed specified temperatures. Each replica corresponds to a temperature. An exchange of temperatures between replicas i and j at temperatures T_m and T_n is attempted periodically and is accepted according to the following Metropolis transition probability [85]:

$$W = \min\{1, \exp[-(\beta_m - \beta_n)(E_j - E_i)]\} \tag{4.1}$$

where $\beta = 1/kT$ and E_i and E_j are the potential energies of replicas i and j, respectively. After a successful exchange, the velocities of replicas i and j are rescaled at the new temperature.

Existing Parallel Implementations of Replica Exchange-based Molecular Dynamics Simulations

Molecular dynamics programs are essentially loops over a large number of integration steps, each of which advances the time forward for one step. Replica exchange is attempted periodically after a chosen interval of steps. As mentioned in the introduction, existing parallel implementations of replica exchange are MPI [4]-based, centralized

and synchronous, and target relative small tightly coupled homogenous systems. For example, in the existing implementation in IMPACT, a central master node collects temperature data about all the replicas from the walker nodes, and then broadcasts the collected data array to the walkers. Each walker node receives this data array and sorts the array locally. Neighboring temperatures in the sorted array are potential partners for temperature exchange. The master node randomly selects between two modes of exchange. One is to exchange with upper neighboring temperature and the other is to exchange with lower neighboring temperature. The master notifies the walkers about the selected mode, and walkers can then mutually exchange temperatures based on this information. During the actual exchange, one of the two walker nodes with neighboring temperatures in the sorted array that are paired up for temperature exchange, acts as a temporary server. This walker collects temperature and potential energy data from the other node, determines whether the exchange is feasible based on the transition probability given in Eq. (4.1), and replies with either the new temperature, if the exchange is successful, or with a notice of denial otherwise.

The parallel replica exchange implementation described above has several limitations. First, the scheme limits the exchange to only neighboring temperatures. This limitation is not a concern when the number of replicas is small and there is a small chance of exchange between non-nearest temperatures. However, as the number of processors (and correspondingly walkers) increases, the difference between target temperatures becomes small enough to allow exchanges between non-nearest neighbor replicas. In such cases, more flexible schemes which allows non-nearest neighbor temperature exchange are desirable. Second, the implementation is based on a centralized master that gathers and scatters data system wide. Gathering data from all the nodes on a single node may be infeasible in large systems, and a centralized master can quickly become a bottleneck. Further, gather and scatter operations are synchronous and expensive. Also, since the master node also participates in the simulation as a walker, there is a load imbalance which can lead to additional synchronization overheads.

The Folding@home [3] project at Stanford University has proposed a multiplexed replica exchange algorithm. The algorithm uses multiplexed-replicas with a number of

independent molecular dynamics runs at each temperature, and attempts exchanges of configurations between these multiplexed-replicas. In this formulation, the efficiency of the simulation is enhanced as a number of independent molecular dynamics simulation replicas are run at each temperature and there are a larger number of potential exchange partners available. Further, the multiplexing between replicas is arranged in such a way that the discrepancy between exchange partners is reduced. In contrast, Salsa improves simulation efficiency by eliminating the limitation of nearest neighbor exchanges, instead of introducing redundant computations. Both algorithms, however, use parallelism to improve the efficiency of the simulation.

To address the above limitations, we proposed an initial asynchronous realization of the replica exchange algorithm in project Salsa [51]. This formulation distinguished itself from existing implementations in two aspects: (1) it allows arbitrary walkers with mutual interesting range to exchange temperature; (2) it enables the temperature exchanges in an asynchronous and parallel manner, and is used in this project. However, Salsa still targeted closely coupled and reliable cluster environments and only supported a single walker per node. This research targets Grid environments, and builds on the initial Salsa asynchronous replica exchange formulation to address platform heterogeneity, environment unreliability, and dynamic walker management.

4.3.2 GARE/CometG, A Grid-based Asynchronous Replica Exchange Engine

GARE/CometG is a Grid-based asynchronous replica exchange engine (GARE) builds on CometG [56]. GARE/CometG provides abstractions and mechanisms to support scalable parallel implementations of the general replica exchange formulation, where walkers can exchange non-nearest neighbor temperatures in a decoupled, decentralized, and asynchronous manner. Figure 4.10 presents a conceptual overview of the GARE/CometG infrastructure. It provides a virtual decentralized shared space abstraction that can be associatively accessed by all walkers. Walkers can use this space to dynamically discover exchange partners and negotiate with them, and exchange data. Walkers periodically post temperature ranges that are of current interest to the space.

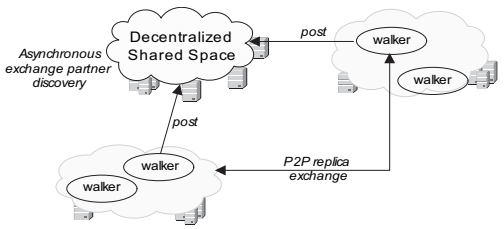


Figure 4.10: A conceptual overview of the GARE/CometG infrastructure.

If this range overlaps with the range of interest posted by another walker, an exchange can occur. Then the individual walkers negotiate and complete the actual data exchange in a peer-to-peer manner. As a result, exchanges are decoupled, dynamically and asynchronously determined, and not limited to neighboring temperatures.

Programming Abstractions

The system builds on CometG and consists of three main layers. The communication layer provides an associative communication service and a direct communication channel to efficiently support data transfers between peer nodes. The content-based routing service provided by this layer maps the multi-dimensional information space to the linear node index space. Note that in the case of replica exchange implementations that only use temperature exchange, i.e., k = 1, CometG uses simple hashing where the index is directly derived from the overall temperature range used by the simulation. The overlay network is composed of peer nodes, which may be any node in the Desktop Grid system (e.g., end-user computers, servers, or message relay nodes).

The coordination layer enables discovery of potential exchange partners between walkers. This layer supports the tuple space coordination abstractions [31], including *out*, *in*, and *rd* operators. The primary components of the coordination layer are a data repository for storing posted replica ranges, a local matching engine, and a message dispatcher that interfaces with the communication layer to translate the coordination primitives to content-based routing operations at communication layer and vice versa. A CometG service daemon running at each node is responsible for handling these exchange interest postings and storing them locally, and for detecting matches with existing postings of exchange interest at the node.

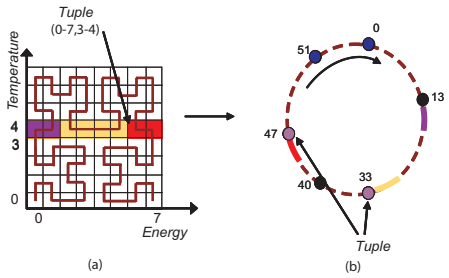


Figure 4.11: Example of tuple post in 2D space: (a) the tuple defines a rectangular region in the 2D space consisting of 3 clusters; (b) the region is mapped to the overlay index and the tuples are routed to the node 33 and 47.

The application layer provides an environment for dynamically managing walkers and protocols for asynchronous exchange. Walkers are extensions of CometG computational tasks and the walker environment configures, initiates, monitors, and manages the local walkers. Walkers are dynamically dispatched to CometG nodes during node initiation. The protocol provides three operators for implementing the asynchronous replica exchange algorithm:

- *post (t)*: inserts a tuple t into space. This operator is used by a walker to express its desire to exchange. Tuple t contains the details of the exchange including a specification of the parameters and ranges (upper and lower bounds) of interest.

- *query (walkerid, timeout)*: sends a query message to a potential partner. The calling walker blocks until receiving a "confirm" or "refuse" response or the specified *timeout* period expires.

- *getp (walkerid, d, timeout)*: exchanges a walker's data d with a selected partner. If the attempt to exchange is successful, the calling walker blocks until the exchange is finished or the specified *timeout* expires. If the attempt fails, *getp* returns with a failure code.

The *post* operator is implemented using the *out* operation provided by the coordination layer. Figure 4.11 illustrates the *post* operation in a 2D replica exchange simulation. In this example, the tuple specifies ranges of interest for the two parameters, which define a rectangular region in the information space. The Hilbert SFC is used to map this region to appropriate index spans in the linear index space and the corresponding peer

nodes to which these index spans have been mapped. The tuple is then routed on the overlay to these nodes. To guarantee data insertion, each *post* request is confirmed by responses from the corresponding destination peer nodes. The *getp* and *query* operators are implemented using the peer-to-peer communication channel to ensure efficient data exchange.

Addressing Grid Unreliability

The GARE/CometG provides fault tolerance mechanisms to address the dynamism and unreliability inherent in Grid environments. These mechanisms assume a fail-stop failure model and timed communication behavior [33]. Under these assumptions, possible failures include walker failure, posted data loss, and negotiation failure. These failures are addressed as follows:

- *Walker failures* are handled using checkpoint-restart. The walker environment periodically checkpoints the local state of each walker, such as its current exchange parameters, to a stable storage. When a walker fails, it can be restarted using this checkpoint. Currently, the detection of walker failures and walker restarts are manual. Note that due to the asynchronous nature of the algorithm, other walkers are not affected by the failure and restart of a walker except that any attempt to exchange with this walker will not succeed.

- *Loss of posted data* occurs only when the node at which a tuple is stored fails since tuple insertions are guaranteed. From a walker's point of view, the impact of this failure is that its attempt to exchange will not succeed and it will repost its request tuple in the next exchange cycle. The resilience of the overlay (e.g., Chord's ability to route around failures [84]) guarantees that the repost will be routed to an operation peer node.

- *Negotiation failures* may result due to the failure of a walker, loss of a message, failure of a communication link, or failure or departure of a node. These failures are handled using timeouts for the *query* and *getp* operations. Once again, due to the resilient nature of asynchronous algorithms, the application is not affected.

Further, redundancy in storage and routing can be incorporated within the overlay as described in [78], where a group of nodes act as one peer. In this case, the consistency of the tuple space as well as issues of group synchronization, such as degree of redundancy, group membership, group communication protocol, etc., must be addressed.

4.3.3 Grid-based Asynchronous Replica Exchange Using CometG

The operation of asynchronous replica exchange using GARE/CometG is illustrated using a temperature exchange example. The operation consists of three phases: (1) the *post* phase in which, candidate exchange partners are identified and notified; (2) the *query* phase in which, potential exchange partners negotiate and agree to exchange; and (3) the *getp* phase in which, confirmed partners attempt to exchange data. When a walker attempts to exchange its current temperature, it computes the target temperature range that it is willing to exchange with, and posts this range using the *post* operator. Based on the temperature range posted, the request is routed to all the nodes whose index ranges overlap with the posted range. When a remote *post* request is received by a peer node, it first checks its local repository for potential exchange partners that have previous posted interests with overlapping temperature ranges. If one or more potential exchange partners are found, the corresponding walkers are notified. Otherwise, the incoming request is stored.

The process is illustrated in Figure 4.12. In this figure, the ranges of interest of $walker_1$, $walker_2$, and $walker_3$ overlap. When the relevant node receives the post from $walker_3$, it discovers that $walker_1$ is a potential exchange partner and notifies the two walkers. Then, $walker_3$ queries $walker_1$ to see whether it is available for an exchange. This is necessary because, even though $walker_1$ has expressed a desire to exchange, it may have already partnered with another walker or may have decided to give up and continue with its computations. On receiving this query, $walker_1$ checks its local state, which can be either "free" or "pending". A walker is available for an exchange only if it is in the "free" state. The "pending" state indicates that the walker is either exchanging with another walker or has committed to exchange with another

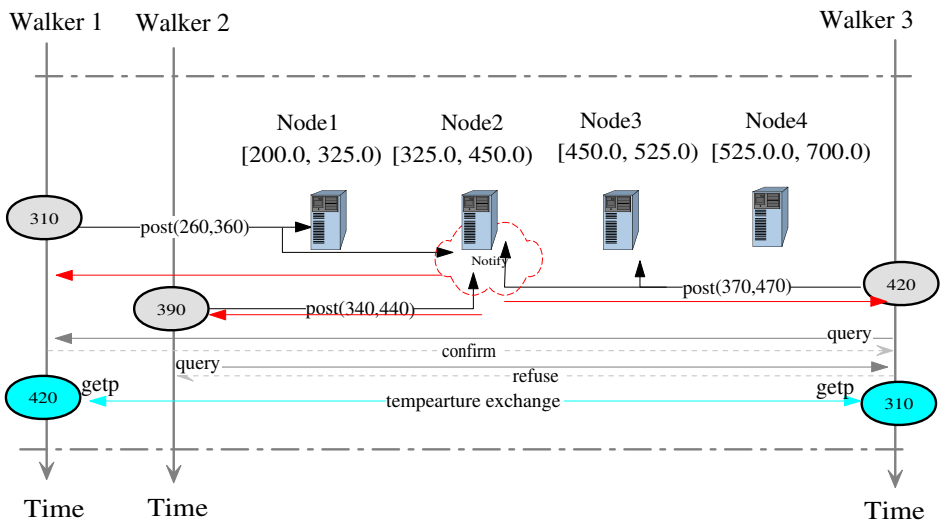


Figure 4.12: Operation of a sample asynchronous replica exchange.

walker but the exchange has not yet occurred. Since $walker_1$ is in the "free" state, it responds affirmatively to $walker_3$ and commits to the exchange. Once both walkers confirm their intents to exchange, they change their states to "pending" and perform data exchange using the *getp* operator. Since $walker_3$'s post also matches $walker_2$'s interest, $walker_3$ receives a query from $walker_2$. Since $walker_3$ has already committed to exchange with $walker_1$ and is in the "pending" state, it refuses this request. In this example, $walker_2$, rather than attempting an exchange with another potential partner, continues computing using its current data and waits until the next exchange cycle to attempt an exchange.

Once a pair of walkers agree to exchange, they initiate the actual exchange by invoking the *getp* operator, which proceeds as follows. One of the walkers sends its current data (e.g. temperature and energy) to its potential partner. The potential partner determines whether the exchange can be completed based on the data it receives and its own data. This step is necessary since the exchange happens asynchronously and in parallel with the computation, and a walker's data (i.e., energy) may have changed since it posted its exchange interest. If the walker decides to continue with the exchange, it will send an exchange acceptance to its partner along with its current local data. It will then wait for a similar acceptance from the partner to complete the exchange. Note

that an exchange is between a pair of walkers and multiple exchanges between different pairs of walkers can proceed in parallel. After the exchange is completed, both walkers remove posted tuples from the space since these tuples and the data they contain are no longer valid.

Since a *post* request typically maps to multiple peer nodes and each node may find more than one partner, it is possible that a requesting walker is notified of multiple candidates located on different nodes. In this case, the first notification that reaches the requesting walker is accepted. In the algorithm, a walker specifies the ranges for each parameter that it is interested in exchanging as part of the *post* operator. Usually, the larger the range is, the higher is the probability of finding an exchange partner and results in better solution quality. However, a larger range will also map to a large number of nodes, which in turn increases communication overheads as well as the load at the nodes, and reduces system performance. In the current system, the *post* operator randomly selects a subset of the nodes to which the interval is mapped, and forwards the post request to these nodes. The size of this subset can be configured by users to achieve desired tradeoffs between solution quality and simulation performance.

If the parameter ranges are not evenly distributed, the posted ranges will result in load balancing issues. In the current implementation, the fact that the parameter ranges are known is used to define a simple load balancing protocol. The distribution of parameter ranges within the linear index space can be analyzed and this analysis can be used while partitioning the index space across the nodes to ensure that the system is load-balanced. Since more general replica exchange formulations may use dynamically defined ranges, we are working on a dynamic load-balancing protocol.

4.3.4 Experimental Evaluation

GARE/CometG has been deployed on a wide-area environment using PlanetLab [7] test bed, as well as a campus network at Rutgers. The objectives of the experiments presented in this section are to demonstrate the ability of GARE/CometG to support wide-area deployments of replica exchange applications and to evaluate performance and scalability.

Table 4.2: Number of temperature cross-walk events.

Number of walkers	8	16	32	64	128
Posted temperature range	decentralized simulation				
$[-200K, 200K]$	49	91	115	251	582
$[-100K, 100K]$	11	43	82	113	262
Posted temperature range	centralized simulation				
$[-200K, 200K]$	74	119	150	178	202
$[-100K, 100K]$	26	55	81	92	126

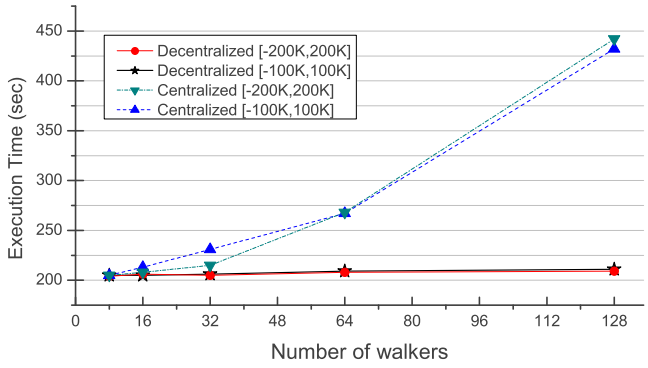


Figure 4.13: Average wall-clock execution time for different numbers of walkers.

A temperature replica exchange simulation, which is based on the IMPACT framework, is used in the experiments presented below. In this evaluation, all the experiments are configured to run for 10,000 sampling cycles, and exchange is attempted every 25 cycles. Temperatures are distributed eventually within the 200-700 K range. At equilibrium each walker should visit each temperature with equal probability. The rate of temperature equilibration is measured by the number of "cross-walks", where a walker originally within the low temperature range (200 K $\leq$ T $\leq$ 250 K) reaches the upper temperature range (650 K $\leq$ T $\leq$ 700 K) and then returns to the lower temperature range. The number of "cross-walks" is measured in the experiments to evaluate the system performance - the larger the number of crosswalks for a run, the better is the performance of the simulation. In the experiments presented below, the temperature range posted by walkers was set to a window of size 400K and 200K around its target temperature, i.e., [temp - 200K, temp + 200 K] and [temp - 100K, temp + 100 K].

The first set of experiments were conducted on a Grid consisting of heterogeneous Linux-based computers on the Rutgers campus network. Each computer runs a single

CometG instance and supports 4 walkers. Table 4.2 shows the number of temperature cross-walks measured for decentralized replica exchange simulations with 8, 16, 32, 64, and 128 walkers, compared with the corresponding number of cross-walks obtained using a traditional centralized approach where exchanges are all conducted at a central (master) node. The latter case was achieved by mapping the CometG shared space to a single peer node. As shown in the table, the number of observed temperature cross-walks increases with increasing numbers of walkers and the posted temperature range. The decentralized implementation achieves more cross-walks than the centralized approach when the number of walkers is greater than 64, although the centralized approach achieves more crosswalks for a small number of walkers. This is because a centralized node quickly becomes a bottleneck as the number of walkers increases. The average wall-clock execution time of the simulation for different numbers of walkers are plotted in Figure 4.13. As seen in the figure, the decentralized implementation scales well, while as expected, the centralized implementation does not scale. The impact of centralization is even more pronounced for larger systems in wide-area Grid environments.

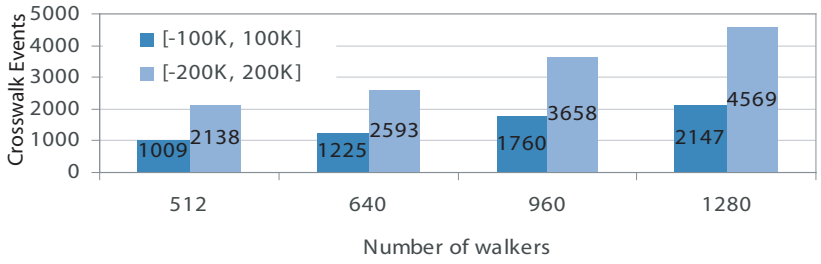


Figure 4.14: Effect of the posted temperature ranges on the number of temperature crosswalk events for large numbers of walkers.

The second set of experiments measured the number of temperature cross-walks for larger numbers of walkers in the decentralized implementation. The GARE/CometG-based replica exchange implementation supports non-nearest neighbor temperature exchanges, which is essential for ensuring proper mixing of temperatures across the walkers, especially when the number of walkers is large. This experiment used a fixed system size of 32 nodes and evenly distributed the walkers across these nodes. Figure 4.14 plots

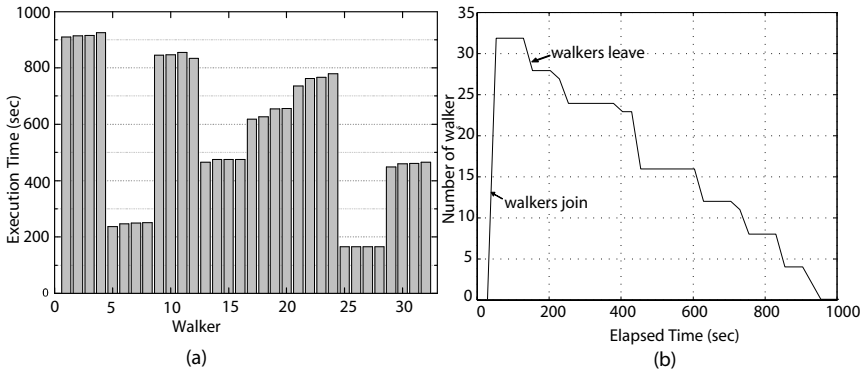


Figure 4.15: Evaluation of GARE/CometG-based replica exchange on the PlanetLab wide-area test bed. (a) Execution time of the different walkers. (b) Change in the number of walkers due to the dynamism of the environment.

the number of cross-walks for temperature range [-200K, 200 K] and [-100 K, 100 K] using 512, 640, 960, and 1280 walkers. These results illustrate the effects of increasing the temperature range on the number of cross-walks. The results also demonstrate the ability of GARE/CometG to effectively support the increased communication due to larger numbers of walkers and larger temperature ranges.

The third experiment was conducted on the wide-area PlanetLab [7] test bed. PlanetLab is a large scale heterogeneous distributed environment composed of interconnected sites on a global scale. The goal of this experiment is to demonstrate the ability of GARE/CometG to support replica exchange applications in unreliable and highly dynamic environments such as PlanetLab, which essentially represents an extreme case for a Desktop Grid environment. GARE/CometG was deployed more than 200 machines on PlanetLab, however, only a fraction of these nodes could be effectively used at anytime. This experiment was conducted on January 03, 2007. In the experiment, we used temperature range at [-200K, 200K] and 32 walkers, which were dynamically mapped to nodes that joined the replica exchange space. Walkers were dynamically initialized on the nodes (up to 4 walkers per node). The timeout threshhold of *getp* and *query* operation was set to 5 seconds. These walkers dynamically joined the application, started their computation and performed exchanges, and left the system when their computation terminated or the node failed or lost connectivity. The joining or leaving of walkers did not impact the execution of other walkers. The number of

walkers was monitored during the experiment. The application terminated after all the walker finished their sampling cycles, in which 86 cross-walk events were observed. The results of the experiment are plotted in Figure 4.15. Figure 4.15 (a) plots the execution time for each walker. This plot illustrates the heterogeneity of the PlanetLab nodes. Figure 4.15 (b) demonstrates the fluctuations in the number of walkers during the lifetime of the application due to the dynamism of the environment.

This chapter presents the design and implementation of CometG decentralized computational infrastructure that extends Desktop Grid environments to support robust parallel asynchronous applications. CometG builds on top of Comet and provides scalable communication/coordination abstractions and programming abstractions for supporting parallel asynchronous iterative computations and replica exchange simulations. Two prototype systems have been implemented and evaluated on top of CometG to demonstrate the effectiveness of this infrastructure. The experimental results demonstrate both the performance and scalability of the CometG system. The results also illustrate the effectiveness of using CometG to support the parallel asynchronous applications in Desktop Gird environments.

Chapter 5

Rudder, An Agent-based Coordination Framework for Autonomic Composition of Grid Applications

This chapter describes Rudder [55, 53, 54, 58], a decentralized agent-based coordination framework for supporting autonomic composition of Grid applications. Rudder provides software agents and coordination protocols for the dynamic discovery and selection of software service elements, enactment and configurations of workflows, and the management and adaptations of these workflows to respond to changing Grid environments. Rudder also implements the agent interaction and negotiation protocols and enables appropriate application behaviors to be dynamically negotiated and enacted. The defined protocols and agent activities are supported by Comet, which provides a scalable decentralized shared-space based coordination substrate. The implementation, operation, experimental evaluation, and an illustrative example of the system are presented.

5.1 Autonomic Composition of Grid Applications

The goal of the Grid infrastructure is to enable a new generation of applications that combine intellectual and physical resources spanning multiple organizations and disciplines, and provide vastly more effective solutions to scientific, engineering, business and government problems [70]. As Grid computing has evolved, the collaborative problem solving enabled by the Grid has also evolved from primarily file exchange to direct access to hardware, software and information components. The resulting Grid applications, which are based on seamless discovery, access to, and interactions among resources and services, have complex and highly dynamic computational and interaction behaviors, and when combined with the uncertainty of the underlying infrastructure,

result in significant development and management challenges. Autonomic development and management strategies, which are inspired by biological systems and the human autonomic nervous system, have recently been proposed [39, 47, 71] to address these challenges.

A key issue in the development and management of these autonomic applications is the autonomic composition of the distributed autonomous elements. A single available element might not address specific application requirements, and composing several elements to form a unit with integrated functionalities is necessary. Autonomic composition enables applications or parts of an application to be dynamically composed from discrete elements to meet the changing requirements and system behaviors, deal with element failures, optimize performance, address QoS constraints, etc. However, enabling autonomic composition is difficult in Grid environments because the available elements are typically numerous, heterogeneous, and available in a dynamic on-demand manner. Challenges include element descriptions, their discovery, and their dynamic and adaptive composition, interaction and coordination. Recent initiatives, such as the "Semantic Grid" [75], complement the Grid service-oriented architecture [37] to enhance the scientific process with seamless interaction on a global scale. Effectively, the composition of loosely coupled Grid services is emerging as the desired paradigm for constructing Grid applications. In this context, solutions being developed as part of the "Semantic Web" [21] can be leveraged to support accurate Grid service description, discovery, and composition. However, an effective autonomic composition framework is still absent in Grid environments.

Dynamic composition, then, can be viewed as the runtime specification, configuration and enactment of these workflows. Workflows and workflow composition has been an area of active research in the Grid community in recent years. Current research efforts in this area can be broadly classified into two categories: (1) automatic workflow generation and (2) dynamic workflow enactment. Issues addressed by research efforts in the first category include ontology reasoning, deductive theorem proving, and AI planning, while research efforts in the second category focus on infrastructure support for service discovery, workflow configuration and execution, and workflow adaptation. This

research belongs to the second category. Specifically, this thesis presents Rudder based dynamic workflow compositions, which employs semantic web, multi-agent systems, and workflow enactment techniques to discover, select, and compose heterogeneous and distributed Grid services.

5.1.1 Autonomic Composition of Grid Applications: Requirements and Current Approaches

As outlined above, the inherent scale, complexity, heterogeneity, and dynamism of emerging Grid environments and applications result in significant programming and runtime management challenges. As a result, developing Grid applications requires redefining Grid programming frameworks and middleware services. Specifically, it requires that static (defined at the time of instantiation) application requirements, their structures and system and application behaviors be relaxed, and that the behaviors and structures of elements and applications be sensitive to the dynamic state of the system and the changing requirements of the application and be able to adapt to these changes at runtime. Clearly, enabling autonomic composition of elements and applications is a key issue in effectively addressing these requirements. Enabling autonomic composition requires conceptual frameworks and an implementation infrastructure. Conceptual frameworks consist of models, languages, standards, methods and constraints that govern the composition of elements. Implementation infrastructures provide the mechanisms, including programming and run time systems, to enforce the compositions specified using the conceptual framework.

Conceptual Frameworks

Conceptual frameworks address the following issues: (1) *Element specification:* The conceptual framework should unambiguously identify an element, and should be sufficiently rich to capture the capabilities of an element, including its functional attributes (e.g., input, output, precondition, effects, etc.) and non-functional attributes (e.g., cost, service quality, security, etc.). Further, the specification should be formally defined and capable of being processed, interpreted and reasoned using agents/machines,

e.g., to check if two descriptions are equivalent, partially match, or are inconsistent. (2) *Application process specification:* The conceptual framework should provide information about the elements involved in the application process, their roles, and their interactions. This specification is similar to a workflow, and includes a set of activities and their execution dependencies. (3) *Composition policy specification:* The conceptual framework should define composition methods and constraints and enable users to specify requirements such as cost, performance, QoS, etc.

Related work in conceptual frameworks for autonomic composition includes efforts within the Semantic Web community addressing the description, discovery and composition of services. Projects such as myGRID [91] represent recent efforts aimed at uniting the Semantic Web and Grid computing communities. In particular, the Web Ontology Language (OWL) [69] is emerging as a standard in industry as well as in the scientific and engineering research communities, for Web service discovery, composition, and invocation. The OWL-S *Profile* specifies a service using three information components: service capability specified in terms of its inputs, outputs, preconditions, effects, and component sub-processes; service attributes such as QoS, cost, and classification in the taxonomy; and description of service providers. Note that in addition to describing advertised services, profiles can also be used to describe requested services. The OWL-S *Process Model* allows the requesters to decide whether and how to interact with a service. It defines the basic functions performed by service providers as atomic processes, which can be composed into more complex processes using control structures such as sequence, if-then-else, or split. Finally, OWL-S *Grounding* specifies the implementation details of a service such as messaging protocols and message formats.

In the Grid computing community, workflow models are popular approaches for describing and composing complex scientific applications. A Grid workflow is a set of tasks that are processed on distributed resources in a defined order to accomplish a specific goal. Workflow management techniques can be applied to generate Grid workflow and dynamically assemble applications using services and resources distributed across the Grid. In general, workflow-based systems enact abstract workflow descriptions as composition plans to discover and compose elements. The workflow description can be

user-defined or autonomically generated. Workflow descriptions may use markup languages such as XML, WSFL [10], XLANG [11], BPEL4WS [12],and GSFL [50], or use a graphic representation such as Petri Nets [74] and UML (Unified Modeling Language). However, these languages do not provide well-defined semantics, which in turn limits their ability to support seamless service interoperability. On the other hand, while the OWL-S profiles allow descriptions to be more precise, its process model lacks flexibility. For example, OWL-S does not describe the relationships between the elements, their synchronization, or the termination of a process. A possible solution is to extend workflow descriptions to use the OWL-S profile ontology for element and composition specification.

Finally, composition polices and constraints allow users to express specific requirements and expectations such as performance and availability. Grid environments provide a large number of similar or equivalent services and resources. These services may provide the same functionality but may optimize different non-functional aspects such as performance, cost, reliability, security, etc. Further, different users or applications may have different expectations and requirements. Therefore, only considering functional characteristics during the composition may be insufficient.

Implementation Systems

Critical components of an implementation system for autonomic composition include an efficient, scalable and flexible discovery mechanism, and a high-level integration mechanism. The discovery mechanism enables the selection of appropriate elements while the integration mechanism enables selected elements to be composed coherently, without conflicts in element dependencies and interactions. Most of existing approaches compare the syntactic and semantic components [61] of element descriptions during the generation of the composition plan. However, this approach may not ensure runtime compatibility during application execution due to the dynamic availability and state of elements and resources on the Grid. As a result, runtime composability and compatibility checking is important for autonomic composition, specially since interactions can ad hoc, ephemeral and opportunistic.

Realizing an autonomic composition framework for Grid applications presents several challenges. Such a system has to implement services and protocols to address element representation, discovery, and cooperation, while addressing the scale, heterogeneity and dynamism of Grid environments and applications. Key design issues include the overall system architecture as well as discovery, communication, and coordination subsystems. In a centralized architecture, every element publishes its existence, capabilities and functionalities in a globally known and possibly centralized registry, and every agent queries this central registry to discover elements and compose applications. However, such architecture suffers from performance and scalability bottlenecks, single point failures, and may be more vulnerable to denial of service attacks. On the other hand, decentralized architectures are more scalable, resilient and have higher availability, but require mechanisms for maintaining information consistency and tend to be more complex. High-level communication and coordination subsystems that are based on semantics rather than names/identifiers and addresses and provide abstractions for process cooperations, communication and synchronization, can help reduce this complexity.

The software agent paradigm provides decentralization, dynamic and coordinated decision-making and autonomous behaviors, and supports representation translation, dynamic discovery and negotiated coordination, making it an effective approach for realizing autonomic composition systems. The motivations for employing agents to address autonomic composition for self-managing Grid applications include two aspects. First, agents with knowledge capabilities provide a natural abstraction for bridging external and internal data structures in the system. Typically, discovery systems provide external representations that enhance element accessibility and allow users to relatively easily express what they can offer or what they want, e.g., using the OWL-S profile ontology. Internally, discovery substrates use specific representation, such as keywords, for data indexing and query resolution to achieve efficient and scalable data lookup. Agents provide an effective mechanism for translating between and gluing these representations. Second, the adaptive behaviors of agents enable the composition plans and policies to be enacted through a dynamic negotiation process. Agent negotiation

mechanisms can be used to the selection of the most appropriate elements from those that are currently available. This includes evaluating non-functional attributes of the elements, which can be difficult to estimate or predict in dynamic Grid environments and may result in sub-optimal selections.

5.2 Rudder, an Agent-based Coordination Framework for Grid Applications

Rudder agent framework [55, 53, 54, 58] provides agent abstraction and coordination protocols for supporting dynamic composition, coordination, interactions and application self-managing behaviors. The Rudder framework builds on top of Comet substrate. The Comet substrate provides a shared-space abstraction and supports the implementation of the coordination protocols. Rudder employs the Comet substrate to provide a two-level composition spaces. A conceptual overview of Rudder is shown in Figure 5.1, and consists of 4 key components:

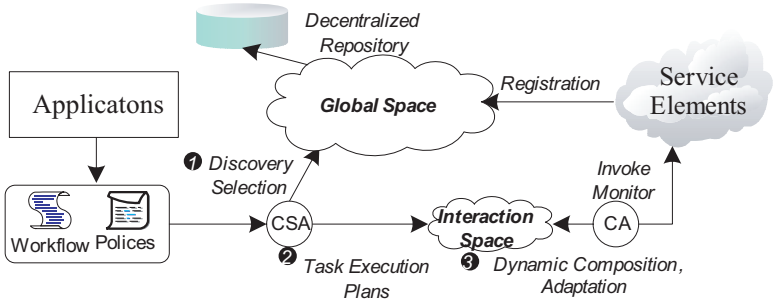


Figure 5.1: A conceptual overview of the Rudder.

- *GlobalSpace*, which is a persistent space that supports the registration, publishing and semantic discovery of service elements. A service provider can access the *GlobalSpace* to register and publish its service in Rudder.

- *InteractionSpace*, which is a dynamically constructed contextually localized space that is dedicated to a particular workflow. This space provides the interaction/coordination medium for configuring and enacting a workflow, and only includes the providers of services that are a part of the workflow.

- *Composition agent (CSA)*, which manages one or more workflows. This agent is responsible for discovering service elements to perform workflow tasks, instantiating a CA to manage each discovered service element, dynamically negotiating with the CAs to select elements based on system context and user preferences, and generate task execution plans to enact the workflows.

- *Component Agent (CA)*, which manages a software/service element and the execution of the workflow task assigned to that element. This agent is responsible for monitoring the service element and enforcing the workflow adaptation to respond to application/system dynamics based on specified adaptation policies.

5.2.1 Classification of Rudder Agents

The Rudder agent framework defines two types of agents: Component Agent (**CA**) and Composition Agent (**CSA**). CAs represent discrete service elements and use OWL-S profile to identify and control the elements. A service element may be an application, service or resource unit (e.g., computer, instrument, and data store). Such an element along with its CA represents a managed element in Rudder. The responsibilities of a CA include advertising the capabilities of the element, providing uniform access to the element, configuring the element based on its execution context, managing its execution, and adapt the composition. The CA supports workflow adaptation using element switching, which allows a failed or an active element to be replaced with a single element at run time to obtain good performance. In Rudder, the candidate elements form a redundancy group for a workflow task. The service discovery assumes these elements have equivalent functionalities and input/output parameters. Thus, any element in such a group can be replaced with another in the same group, while maintaining the syntactic and logic correctness of the workflow. A CA has 3 main components, shown in Figure 5.2, which are described in the following:

- *Task manager* accesses the *InteractionSpace* and manages the task execution. It extracts the plan tuple from the space, generates the task templates, dispatches the retrieved task, generates the resulting task tuples, and inserts them into the

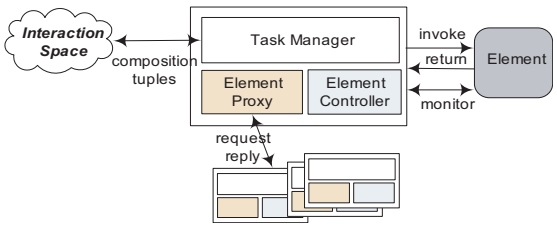


Figure 5.2: The structure of an element service agent.

space. It has two dispatching modes: default mode, in which the task is dispatched to *element controller*, and forwarding mode, in which the task is dispatched to *element proxy*. The *task manager* switches to forwarding mode if the element switching is triggered.

- *Element controller* monitors the element and invokes the task execution. It periodically queries to see if the element is still available. If the element does not response for a pre-defined time interval, the element is considered to be failed. Once getting a task, it invokes the element to execute the task and waits for the results. During task execution, it also queries the performance metric and the transferred states. If execution is successful, it returns the results to the *task manager*. Otherwise, a failure is returned. the *element controller* triggers the switching when the element fails or the switching rule fires.

- *Element proxy* selects and forwards the task request from the *task manager* to a replacement candidate element. After getting the task, it broadcasts a query message to all candidate CAs and accepts the first replying SA as the replacement. It then sends task execution request to this CA and waits for the results. If state transfer is required, it retrieves the current states and sends a continue execution request to the CA. The replacing CA gets the execution request and invokes a task execution. Finally, it forwards the results to *task manager*. If no candidate CA is selected for a defined time interval, it raises a selection failure. The *task manager* catches this failure and inserts a "Stopped" execution plan into the space. Once this happens, the CSA notifies the administrator to re-initiate a discovery.

CSAs are transiently generated agents. The CSAs dynamically discover and compose managed elements to realize applications. CSAs employ predefined composition plans to discover relevant elements and to negotiate with the CAs to select, configure, and compose the elements. In Rudder, composition plans are generated from the application process and are available to the CSA [1]. Further, the semantics of terms and concepts used in the composition plans as well as the application specific ontology are common knowledge among agents. A composition plan has three components: (1) a set of atomic tasks, each of which has a semantic description, using the application ontology, that can be used to discover and select elements to fulfill the task; (2) a process description describing the dependencies and interactions between tasks; (3) constraints, which reflect user requirements and may be defined at the task level (e.g., minimize the execution time of a task) as well as the plan level (e.g., minimize total cost). A composition plan is enacted by a CSA by using the task descriptions to semantically discover elements, selecting and configuring appropriate elements, composing these elements using the process description and coordinating with other agents to satisfy constraints and application requirements.

5.2.2 Coordination Protocols In Rudder

Coordination protocols provided by Rudder include discovery protocols and interaction/negotiation protocols. Rudders uses Comet to realize *GlobalSpace*, which supports and implements discovery protocols for semantic service storage and discovery.

Discovery Protocol: enables agents to register, unregister, and discover elements. In Rudder, element profiles are categorized based on an application defined taxonomy, and mapped onto a corresponding semantic space. The *GlobalSpace* uses a profile to map the element to the corresponding logical semantic space, where the dimensions of the space are the keywords that can be used to describe the service element. The coordinates of the service in this space are then used to store the service into *GlobalSpace* and to discover it. The discovery process consists of navigating this semantic space

[1] Plans may be automatically generated through AI planning and deductive theorem proving. However, this is not currently addressed in Rudder.

to narrow an element query to a small set of potential matching profiles and then performing semantic matching on these profiles.

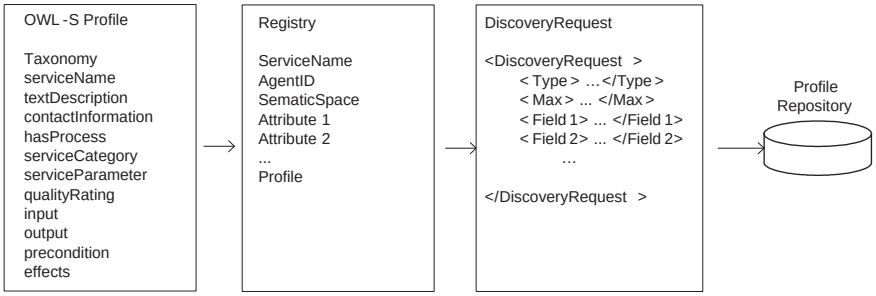


Figure 5.3: Discovery message for registering/unregistering an element.

When an element is added to the system, its associated CA parses the element's OWL-S profile description, and creates a registry entry that uniquely identifies the element in Rudder. The attributes form the coordinates of a semantic space. For example, a computational storage resource may belong to the 3D storage space with coordinates "space", "band width", and "cost". The process is shown in Figure 5.3. The registry itself is decentralized and is implemented using the Comet substrate described below. The CA is also responsible for maintaining the consistency of this information in the registry and updates the registry when one or more of the element's attributes change. A periodic heart-beat message is used to ensure the liveliness of elements. When the element permanently leaves the system, the agent unregisters the service and deletes the corresponding registry entry.

The discovery protocols allow agents to search for elements. Searching consists of two steps. First, the agent generates the request description identifying the semantic space and consisting of relevant keywords, partial keywords, and/or wildcards. It then searches for candidate elements within the decentralized repository in a distributed manner. The matching process consists of an initial lexical matching of the keywords in the query followed by a semantic matching [87] to evaluate the similarity between the request and matched element profiles. The matching elements are returned to the requesting agent.

Interaction Protocols: allow distributed agents to interact, coordinate and negotiate during composition in order to reach a mutually acceptable agreement. Implemented protocols are based on existing agent interaction protocols [28], such as those for consensus, mutual exclusion, bargaining, auctions, distributed constraint satisfaction, coalition formation, distributed planning, etc.

The appropriate protocols are selected based on the composition context. During element selection, the CSA can make a decision based on a fixed criteria (e.g., minimum execution time), and consequently the simple and efficient Contract-Net Protocol (CNP) [82] (see Section 5.3.4) is employed. Similarly, a Marketplace like service-oriented negotiation protocol [81] (see Section 5.3.4) is employed when the agents need to achieve a mutually acceptable agreement within a dynamic context, for example, the negotiation of non-functional element properties.

5.2.3 System Implementation

The current prototype of Rudder has been implemented using Comet. A conceptual overview of the Rudder implementation architecture is presented in Figure 5.4. As

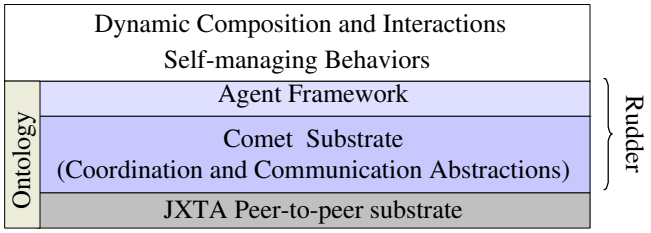


Figure 5.4: An overview of the Rudder implementation architecture.

described in Chapter 3, the Comet coordination layer provides primitives to support shared-space based coordination model, including **out**, **in**, **rd**, and **rdall** operators. Each peer node in Comet provides an agent environment responsible for generating, configuring, and destroying agents. Agents are implemented in JAVA as single thread processing units. The agents communicate with each other by associatively reading, writing, and extracting tuples, and interact using the Rudder interaction protocols. These protocols are implemented using the abstractions and services provided by Comet.

The Rudder discovery protocol is implemented as follows. When an element is added to the system, its CA registers the element by writing its profile into the global tuple space using the Comet **out (ts, t)** operation, where *ts* identifies the semantic space and tuple *t* encapsulates the registration request. The registration request is routed by Comet to the appropriate peer node in the overlay and the profile is stored in a local repository at that node. Similarly, an element can be unregistered using the **in** operator. Agents can query a single matching element using the **rd** operator or all matching elements using the **rdall** operator.

As in the registration case, the query is routed by Comet to the appropriate peer node(s) in the overlay, where semantic matching is used to check the similarity between the request and available profiles. The semantic matching process compares the syntactic and semantic composability of the elements. It is similar to semantic web service matching [87] and ensures that the interacting elements are compatible in aspects of operation modes (request-response), messages, number of parameters, data types, binding protocols, etc. This matching can be implemented using OWL-S matching tools such as OWL-S Matcher [68].

Interaction protocols are implemented using the communication abstractions provided by Comet as follows. For each negotiation, the first step involves session setup where the initiating agent creates a session identifier. This agent then sends the setup message to the selected agents and waits for their acceptance. Once the negotiation has been setup, the initiator informs the participants of the interaction protocol and related information, such as negotiation item, bargaining strategies, roles, etc. After the setup is complete, the agents engaged in the negotiation can directly interact in a peer-to-peer manner.

5.3 Autonomic Composition of Grid Workflows Using Rudder

Rudder uses the workflow model as its basis for autonomic composing applications. A Grid workflow can be viewed as a set of tasks organized as a well-defined flow of executions, and can be thought of as a composition of Grid services with interaction dependencies. Workflows are effective integration strategies for developing dynamic Grid applications. Further, recent advances in workflow-based techniques allow a workflow to be decomposed as sub-flows and enacted in a decentralized manner, enhancing both performance and scalability [26].

Unlike traditional workflow management system, the Rudder framework presented in this thesis also addresses dynamic service selection and negotiation. In business workflow management, services are selected during the plan generation phase based on user defined constraints or parameters. This approach is effective for business workflows as these applications generally consist of shorter transaction processing tasks with small amounts of data. However, Grid applications are generally more dynamic and involve more long running tasks, larger data flows, and utilize heterogeneous and dynamic resources. This requires dynamic workflows involving dynamic selection, configurations and composition of services. Further, negotiations are required to resolve conflicts and competition between candidate services at runtime to meet user objectives.

In autonomic workflow composition, the first step is to generate composition plans. A composition plan includes a predefined workflow process and user specified constraints. Composition plans are generated as follows. First, the application process is represented using a standard workflow language. This representation is then structurally decomposed into a set of component workflows. A composition plan is created by syntactic processing each component workflow. Specifically, the process description is directly extracted from the workflow description. Each task description is defined as an OWL-S profile. Users may additionally specify non-functional properties for tasks, such as QoS, cost, etc.

Once the composition plans have been generated, the system instantiates CSAs to enact the plans in a distributed manner.The CSAs employ discovery protocols to search

for candidate elements for each task in the plan, and ensure that the selected elements have compatible syntactic and semantic attributes. As several existing elements may provide "similar" functionalities, the agent may use the non-functional properties of the element (e.g., cost, security, privacy, time, availability, etc.) to select the most appropriate one. Further, dynamic runtime selection between these elements may require negotiation. Key steps in the autonomic composition process are illustrated below.

5.3.1 Autonomic Workflow Composition

Rudder views autonomic composition, as the dynamic configuration, enactment, and adaptation of workflows. In Rudder, workflows are specified using XML, along with adaptation policies that define how a task or the workflow may be adapted during execution to satisfy non-functional requirements (e.g., performance, response time, resource usage, availability, etc.). The operation of Rudder is illustrated using the sample workflow shown in Figure 5.5. This workflow starts its execution at task A. When it completes, task A initiates task B, which, after its completion, initiates either task C or D, based on its result. Task C initiates task F, after which, the workflow terminates. Alternately task D is executed, and the workflow terminates.

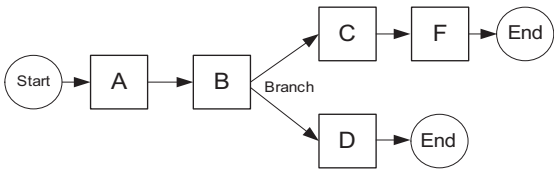


Figure 5.5: An illustrative workflow.

The workflow composition in Rudder consists of 3 phases. (1) *Discovery and selection*: a dynamically instantiated CSA discovers a group of element services for each workflow task, and instantiates a CA for each service. It then negotiates with each CA group to dynamically select an element service. After that, the CSA generates a task execution plan for each task. (2) *Composite service setup*: the CSA dynamically establishes an *InteractionSpace* across itself and the providers of the selected element services. Then, it distributes the execution plans to CAs through the space. (3) *Workflow execution*: the agents coordinate to execute the workflow by taking and writing

composition tuples. During each task execution, the CA may switch the composed element to adapt the workflow. The composition tuples, the three composition phases, and workflow adaptations are described in more detail in the following.

5.3.2 Composition Tuples

Rudder defines two type composition tuples for executing the composition process, which are *plan tuple* and *task tuple*. A plan tuple describes a task execution plan and is generated by the CSA. A *task tuple* triggers a task execution and is generated by the CA from the task execution plan.

```
<Plan>
 <Name>B</Name>
 <Status> Activated </Status>
 <Dependency>
   <Pre Type="Sequence"> A</Pre>
   <Post Type="Branch"> If (Size>1000) Then C
              Else D  </Post>
 </Dependency>
 <CAlist>
   <Selected>
     CA-id-51BBA99F4F1B4084BE62963D772A7BE305
   <Selected>
   <Candidate>
     CA-id-BE3E85BF0F1671F4F2B80D9E13BFD4D5BD
     CA-id-1441E705D9D2E4CC6B369D350C87FCCC50
     ... .
  </Candidate>
 </CAlist>
 </AdaptationPolicy>
    <Switching StateTransfer="True">
    If (performance< thresh_hold) Then switching
    </Switching>
    <TranStates>
      <Name>z</Name>
     ... ..
    </TranSates>
 </AdaptationPolicy>
</Plan>
```

(1)

```
<Plan>
 <Name>*</Name>
 <Status> Activated </Status>
 <Dependency>
   *
 </Dependency>
 <CAlist>
   <Selected>
     CA-id-51BBA99F4F1B4084BE62963D772A7BE305
   <Selected>
   <Candidate>
     *
  </Candidate>
 </CAlist>
 </AdaptationPolicy>
    *
 </AdaptationPolicy>
</Plan>
```

(2)

Figure 5.6: An example of a plan tuple and template.

Figure 5.6 (1) presents a sample plan tuple for task B of the workflow shown in Figure 5.5. It consists of the following 5 fields: (1) *Name*, which identifies the task in workflow. (2) *Status*, which can be "Activated" or "Stopped". (3) *Pre and post dependencies*, which specify the task's relationships to its immediate predecessor(s) and successor(s). These relationships can be one of "Sequence, Branch, And-join, Or-join, or And-split", which are shown in Figure 5.7. Predecessor and successor tasks are specified using task names. Dependency specifications may also include logical operators such as "and", "or", and "if...then...else...". Note that the start task is the predecessor of

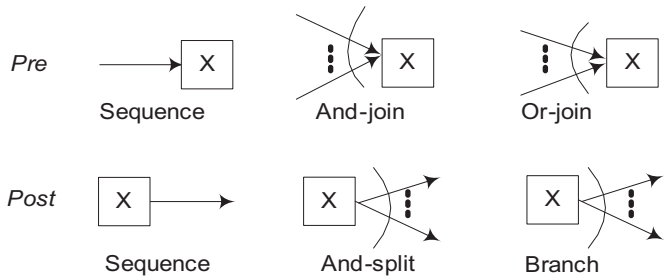


Figure 5.7: *Pre* and *Post* dependency relationships.

the first task in the workflow, and the end task is the successor of the last task in the workflow. (4) *CAlist*, which includes the selected CA identifier and all candidate CA identifiers. (5) *Adaptation policy*, which specifies how to adapt the workflow in response to specific events. For example, if the performance (e.g., throughput, bandwidth, or latency) of a service drops below the required thresh-hold, the element may be replaced. In the element switching, the state transferring can also be supported. If the state transferring is required, the user must explicitly set the transfer attribute as true and identified the names of the transferred states in the following fields.

A task execution plan tuple can be retrieved using a plan template, which matches the plan tuple and may have wildcard "*". Figure 5.6 (2) shows the plan template for retrieving the "Activated" task execution plan B. A task tuple has fields of *name*, *predecessor*, and *input data* including (host and port). The task tuple for task B is presented in Figure 5.8(1). The tuple has name as "B", task "A" as its predecessor, and its input data is located at "foo.cs.bar.edu" at port "9900". A task tuple can be retrieved using a matching task template. Figure 5.8(2) shows the template for retrieving the sample task tuple.

```
<Task>
    <name>B</name>
    <predecessor>A</predecessor>
    <inputdata>
      <host> foo.cs.bar.edu </host>
      <port> 9900 </port>
    </inputdata>
 </Task>
```

(1)

```
<Task>
    <name>B</name>
    <predecessor>A</predecessor>
    <inputdata>
      *
    </inputdata>
 </Task>
```

(2)

Figure 5.8: Examples of task tuple and template.

The task tuples and templates are generated by the CAs based on task execution

plans. Specifically, the CAs map the *Pre* dependency to task templates, and the *Post* dependency to task tuples. The task tuples are generated after the task execution. Each part in the *Post* dependency is mapped to a task tuple. The CA set its plan name as the predecessor, the succeeding task name as the tuple name, and the corresponding data location as the input data. For example, the CA for task A generates the task tuple in Figure 5.8 (1).

The CAs generate the task templates from the *Pre* dependency as below: (i) *Sequence* relationship maps to a task template, which has plan name as the name, the wildcard "*" as input data, and the preceding task name as the predecessor. For example, the task template B in Figure 5.8 (2) is generated from plan B. (ii) Each part participating the *And-join* relationship maps to a task template, which has the similar setting as above. (iii) All parts participating the *Or-join* relationship maps to a task template, which has the similar setting except using wildcard "*" as the predecessor.

5.3.3 Workflow Composition Phases

This section presents the three workflow composition phases in Rudder. Figure 5.9 illustrates the *discovery and selection* phase, which has 4 steps. The user submits the workflow and adaptation polices of the composite service. A CSA is instantiated to process the user inputs. The CSA parses the workflow specification and generates discovery request for each workflow task. Then, it uses the *GlobalSpace* to discover a group of candidate services for each task, and instantiates a CA for each discovered service. The CSA negotiates with each SA group to dynamically select an appropriate service and marks others as backups. Finally, the CSA generates a task execution plan for each workflow task.

Figure 5.10 illustrates the 4 steps of the *composite service setup* phase. The CSA initiates a transient Comet space as the *InteractionSpace*, and invites all the selected CAs to join this space. Each CA accepts the invitation and executes a Comet join protocol [57]. After joining the space, the CA locally accesses the *InteractionSpace* to *In* its task execution plan tuple. Once all CAs have joined, the CSA inserts the "Activated" plan tuples into the space, which will be extracted by the corresponding

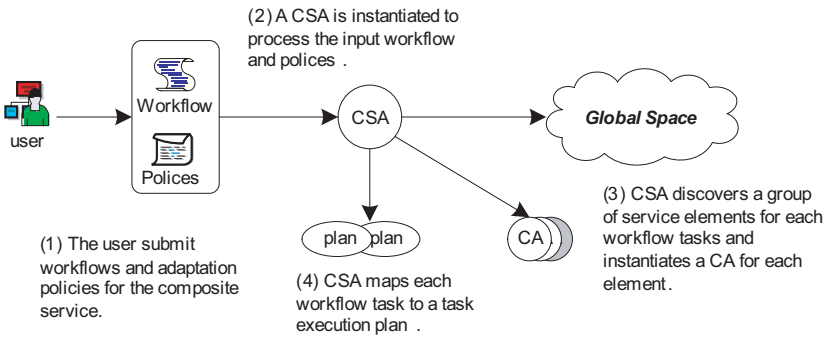


Figure 5.9: Phase 1: Discovery and selection.

CAs. The CAs generate the task templates and initiate *In* operations to wait for their task tuples.

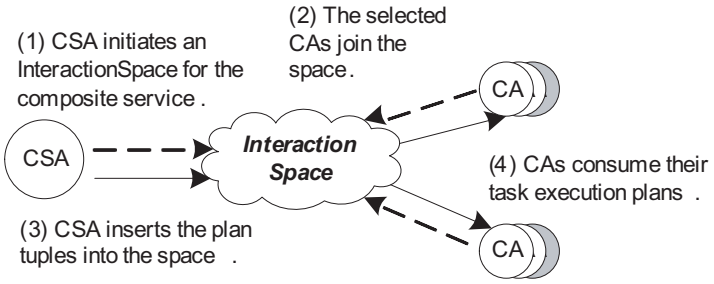


Figure 5.10: Phase 2: Composite service setup.

Figure 5.11 illustrates the *workflow execution* phase. The CSA inserts the first task tuple to start the execution, and waits for an end task tuple or a "Stopped" plan tuple. Then, the CAs coordinate to execute the workflow by taking and writing task tuples. For example, the CA for task B in Figure 5.5, starts executing the task after taking task tuple B. After execution, the CA inserts either task tuple C or task tuple D into the space, depending on the result of task B. The last task's CA inserts an end task tuple, which will be consumed by the CSA. Finally, the CSA collects the output data and returns to user. If the composite service is permanently terminated, the CSA informs the infrastructure to destroy the *InteractionSpace* and all the CAs.

5.3.4 Autonomic Composition Mechanisms

Rudder employs agent-based negotiation mechanisms to enable the autonomic workflow composition. The major negotiation mechanisms include: Contract Net Protocol based

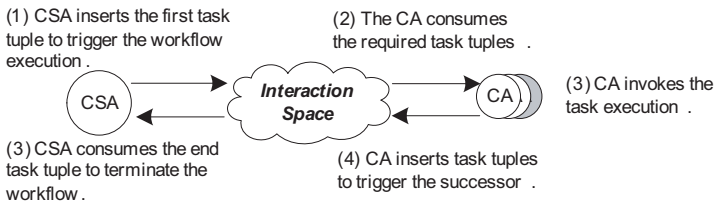


Figure 5.11: Phase 3: Workflow execution.

element selection and marketplace model non-functional properties negotiation.

Dynamic Element Selection

Dynamic element selection is based on negotiations among CSAs and CAs and is illustrated using the Contract Net Protocol (CNP) [82], assuming that the CSA has a composition plan to enact and there are several elements and corresponding CAs capable of executing tasks in this plan. The CNP based negotiation process is illustrated in Figure 5.12. During negotiation, the CSA acts as the manager and the CAs act as contractors. The process consists of the following steps: (1) The CSA searches for candidate CAs using the discovery protocol and advertises the specified task to all candidate CAs. (2) CAs analyze the received task information and respond with a bid; (3) The CSA evaluates received bids, assigns the task to the CA with the best bid, and refuses the other CAs; (4) The CA delegates the task to its associated element and returns the result(s) from task execution to the CSA within a bounded time. If result(s) are not received by the CSA within this time, the CSA explicitly terminates the process. The protocol is implemented using the communication and coordination abstractions provided by Comet. For example, the discovery phase is implemented using the *rdall(ts, $\bar{t}$)* operation, where ts is the name of semantic space used in the task description and $\bar{t}$ consists of keywords from this space. Subsequent agent interactions use peer-to-peer communication abstractions provided by Comet .

The primary reasons for using the CNP-based negotiation protocol are its efficiency and flexibility. The cost of the Contract-Net Protocol is O(N), where N is the number of participating agents. The CNP negotiation process can be customized to specific application requirements. For example, the criteria used by the CSA for choosing CAs

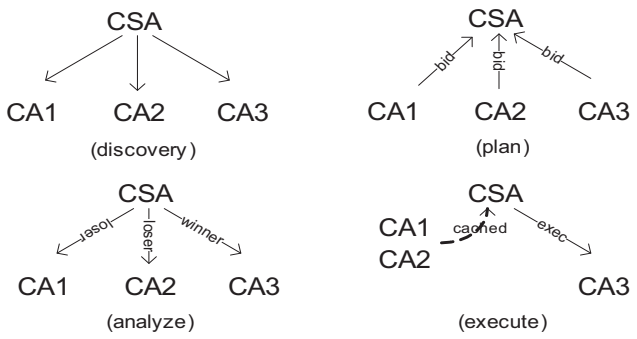


Figure 5.12: Dynamic element selection using CNP-based negotiation.

can be dynamically specified. Further, the CSA may cache information about discovered elements, and can (re)-negotiate with these cached elements if the selected element can no longer perform the task for some unexpected reason. The overall process can be further optimized by having the CSA evaluate bids after receiving a percentage of the bids instead of waiting for all responses.

Multi-Stage Property Negotiation

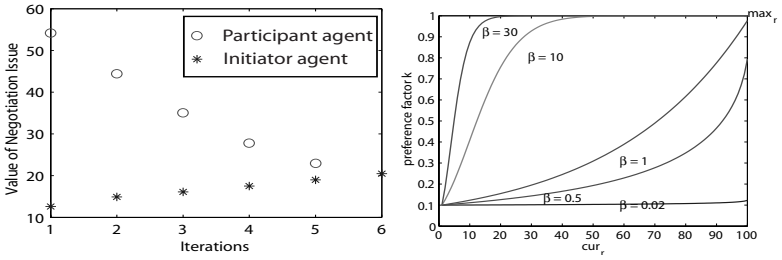
The marketplace model can be used to negotiate non-functional properties of a composition plan. In this negotiation model, instead of a one time determination, values are decided through multiple stage adjustment. This is achieved by iteratively exchanging a finite set of issues between the buyer and seller agents. A buyer agent receives a "plan" from a seller agent, evaluates it and decides either to accept it and stop the negotiation with an agreement, or reject it and propose a counter plan. In case of unsuccessful negotiations, the participating agents can choose to be further coordinated by a mediator, which can be an agent or a system administrator. This mechanism enables agents to resolve locally decided strategies and select a mutually acceptable strategy.

In the implementation of the marketplace model, each negotiation session is setup by an initiator agent. The initiator agent may use the discovery protocol to discovery other participants, which is similar to that used in the CNP protocol described above. Once setup is complete, the agents engaged in the negotiation directly communicate using Comet peer-to-peer communication abstractions. Each negotiating agent uses the remaining amount of a local resource [81] (e.g., remaining number of iterations) to

determine its plan in successive negotiation iterations as follows:

$$f(cur_r) = k^{(\frac{max_r - cur_r}{max_r - min_r})^{\beta}} \tag{5.1}$$

where the possible values of the allocated resource is between $[min_r, max_r]$, the current value is cur_r, the preference factor k $(0 < k < 1)$ determines the initial value of the issue under negotiation, and the conceding rate $\beta(\beta > 0)$ determines the agent behaviors. If $\beta > 1$ the function concedes faster and results in a greedy agent. If $\beta <= 1$, the agent is selfless.



(a) (b)

Figure 5.13: Example of two agent negotiation using the marketplace model.

The mechanism described above enables applications to dynamically tune their configurations to ensure that they continue to meet the composition constraints despite system dynamism and uncertainty. For example, a self-managed distributed Video-On-Demand application must select the appropriate level of network service that can best meet the user requirement while minimizing cost. The desired value of service level must thus be negotiated between the video file server element and the end-user client. For instance, the server has an acceptable range [10, 25] and the client can accept the value in the range of [15, 60]. The appropriate value can decided in Rudder using negotiation between the two component agents. Let the initiator agent have $\beta = 10$ and the other agent have $\beta = 5$. The resource-driven function with $k = 0.1$ used by the both agents are plotted in Figure 5.13(b). As shown in Figure 5.13(a), an agreement is reached after 6 iterations and the negotiated value of the issue is 21. The effectiveness and efficiency of this negotiation process can be tailored using different agent configurations.

5.3.5 Operation and Experimental Evaluation

Rudder is implemented as a peer-to-peer infrastructure where the peers are nodes representing organizations on the Grid that provide and/or consume Grid services. The architecture of the Rudder stack is shown in Figure 5.4. The overall operation of Rudder includes two phases: *bootstrap* and *running*. During the *bootstrap* phase, the infrastructure peer nodes join the *GlobalSpace* group. The *running* phase consists of stabilization and user modes. In the stabilization mode, a peer node responds to messages/queries from other peers in the system, which maintains the overlay routing table and handles node failures or leaving. In the user mode, the user can start composition process on any peer node. The dynamic instantiation time for an *InteractionSpace* includes the time for creating the shared-space and initializing it (in the order of seconds), and time required for a peer node to join this new space (about 10 seconds). Note that this setup time for an *InteractionSpace* is a one-time cost for a workflow and can significantly improve performance by reducing operation latencies.

The experimental evaluation presented as below focuses on element discovery and element selection for dynamic composition. The experiments were setup both on a Rutgers campus network and PlanetLab testbed. The execution time is measured for systems with different number of peer nodes, and for different numbers of elements and agents. In case of two agent negotiation, the overall communication cost is based on the number of negotiation iterations and the latency of peer-to-peer messaging, which is independent of the system size.

Element discovery: The first experiment measures the time required to semantically discover registered elements, which is the interval between when a CSA issues a discovery request and when results containing all element profiles matching the query are returned. This time includes the time for routing the templates at the peers, locally matching the template profile with registries in the local repository at the node, and returning matching results. Note that this measurement does not include the cost of semantic matching. The template used in this experiment is specified using element attributes, including service type, location, and performance/QoS guarantees, etc., and

has a size of at least 440 bytes. The experiments were conducted on the Rutgers campus network. The average execution time shown in Figure 5.14 illustrates that the discovery time increase (from 0.1s to 65s) is much slower than the increase in the number of elements (from 3 to 2700), and is independent of the system size. This demonstrates the scalability of the system and its suitability to distributed decentralized systems.

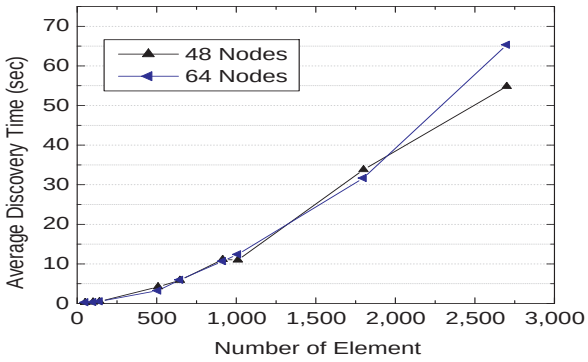


Figure 5.14: Scalability of element discovery.

Element selection: The second set of experiments evaluate the Contract-Net Protocol based element selection the Rutgers campus networks, in which CAs represent elements randomly distributed at the peer nodes and a CSA attempts to find the best CA to execute a task. The task length is fixed and independent of the element selection time. The tasks are generated using a Poisson process with inter-arrival mean time of 1s and 5s to simulate different application behaviors. The CSA begins the bid evaluation process when (1) it receives all the bids or (2) it receives a certain percentage of bids. The measured execution time is from the time when the CSA announces a task to the time when it gets the results from the selected CA to which the task is assigned, and does not include the task execution time. The time thus includes task announcement, element selection, and result return communication time.

The first experiment was setup on Rutgers network. Figure 5.16(b) plots the average execution time for the two cases, i.e., when the CSA begins to evaluate the bids after receiving all the bids and only 50% of the bids (i.e., Eva_r=0.5). In Figure 5.16(b) , the execution time increases linearly with the number of CAs, and the performance of

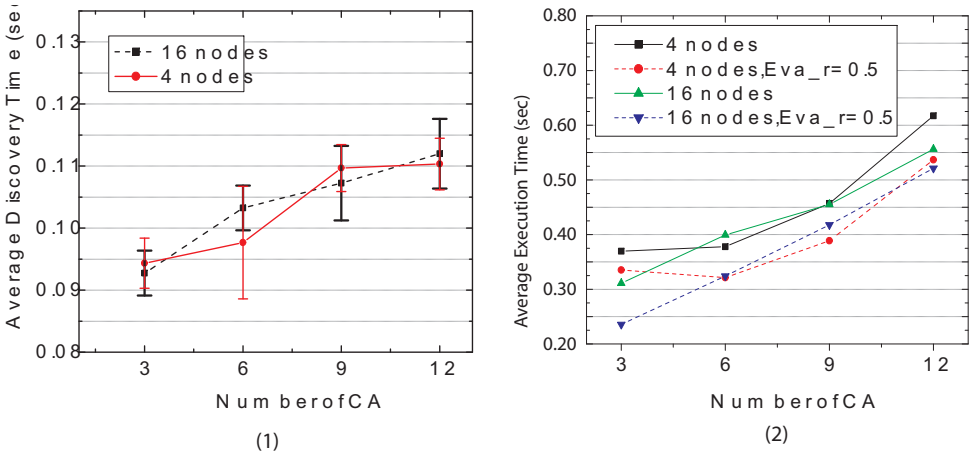


Figure 5.15: Average CNP-based element selection and execution time.

this process is improved in case (2). Element discovery time is also measured in this experiment, and is separately plotted in Figure 5.16(b) (1). In the plot, the discovery time increases only about 20% when the number of matched profiles increases 400%.

The second experiment was conducted on PlanetLab testbed using the same measurements. Figure 5.16 plots the average execution time for the two cases, i.e., the CSA begins evaluating bids after it receives all the bids; and the CSA begins evaluating bids after it receives only 50% of the bids (i.e., Eva_r=0.5). Figure 5.16(a) plots the element discovery time, which increases by only about 28% when the number of matched profiles increases by 600%.

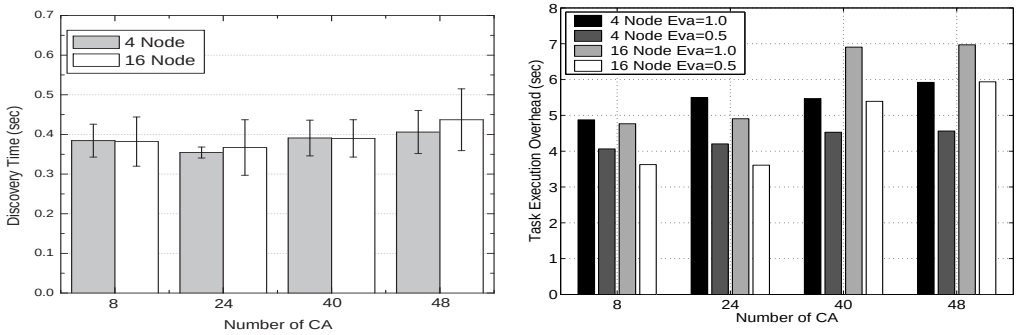


Figure 5.16: Average CNP-based element selection and execution time.

The results of above two experiments show that element discovery scales well and

is fairly independent of the system size and the execution overhead increases linearly with the number of CAs, which is expected.

5.4 An Illustrative Example: Autonomic Oil Reservoir Optimization Using Rudder

One of the fundamental problems in oil reservoir production is determining the optimal locations of the oil production and injection wells. However, the selection of appropriate optimization algorithms, the runtime configuration and invocation of these algorithms and the dynamic optimization of the reservoir remain challenging problems. In this example we use Rudder to support the autonomic compositions, interactions and adaptations to enable an autonomic self-optimizing reservoir application. The application consists of key elements: (1) sophisticated reservoir simulation components (e.g. IPARS [89] factory) that encapsulate complex mathematical models of the physical interactions in the subsurface; (2) distributed data archives that store historical, experimental, and observed data; (3) sensors embedded in the instrumented oilfield providing real-time data about the current state of the oil field; (4) optimization services based on the Very Fast Simulated Annealing (VFSA) [24] and Simultaneous Perturbation Stochastic Approximation (SPSA) [89]; (5) the economic modeling service.

These elements need to dynamically discover one another and interact as peers to achieve the overall application objectives. First, the simulation components should dynamically obtain necessary resources, detect current resource state, and negotiate required qualities of service. Next, the simulation components must interact with one another, and with archived history and real-time sensor data, to enable a better characterization of the reservoir. Further, the reservoir simulation components interact with optimization services and with the data to optimize well placement, with weather services to control production, and with economic modeling service to detect current and predicted future oil prices so as to maximize the revenue from the production.

The operation of this application using Rudder, is illustrated in Figure 5.17. The overall process is achieved by (1) generating composition agents based on application

workflows, (2) agents discovering and composing the involved components to enable the oil reservoir management process, which includes monitoring oil production behaviors and detecting needs for optimization, and (3) agents using high-level policies to orchestrate interactions to optimize well placement and oil production.

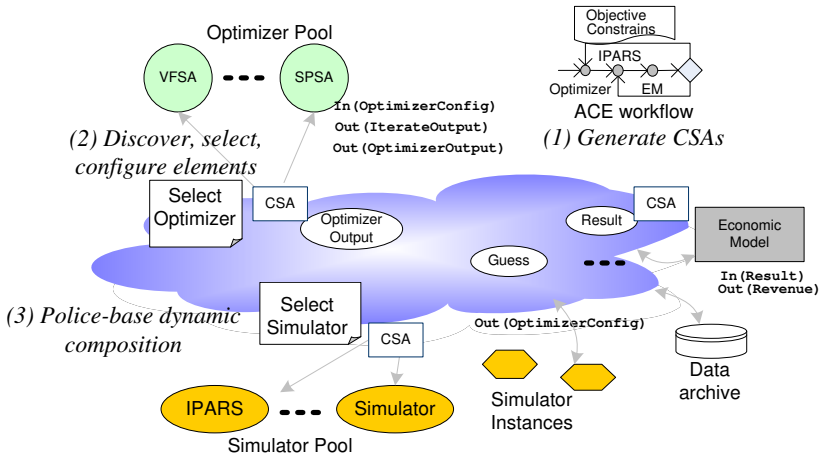


Figure 5.17: Autonomic Oil Reservoir Optimization

First, the composition engine (ACE) [14] generates the following workflows to satisfy the application objectives: (i) the optimization service provides the IPARS reservoir simulator with an initial guess of well parameters based on the configuration of the oil field; (ii) IPARS uses the well parameters along with current market parameters to periodically compute the current revenue using an Economic Model (EM) service; and (iii) IPARS iteratively interacts with the optimization service to optimize well parameters for maximum profit. Based on above workflows, three CSAs are instantiated for the EM, Optimizer, and IPARS respectively. The CSAs dynamically discover the appropriate autonomic elements with desired functionality and cost/performance characteristics using the discovery protocol, and configure the workflows using interaction rules. The CAs use the interaction rules to dynamically establish interaction relationships among the elements and using appropriate communication mechanisms. The CSAs then coordinate with the CAs to enable the application.

Application self-management and self-optimization behaviors are achieved via the autonomic behaviors of the agents. Each CA monitors and manages the execution of its

element, while the CSAs discover and compose elements and resources to satisfy current application objectives. For example, the choice of optimization algorithm depends on the size and nature of the reservoir. In case of reservoirs with many randomly distributed maxima and minima, the VFSA algorithm can be employed during the initial optimization phase. Once convergence slows down, VFSA can be replaced by SPSA, which is suited for larger reservoirs with relatively smooth characteristics. Using these policies, the Optimizer CSA selects the appropriate optimization service, and configures it to optimize the application according to the current objectives of the application.

This chapter describes Rudder agent-based coordination framework for autonomic composition of Grid applications. Rudder provides software agents to enable the dynamic discovery and composition of application workflows. In Rudder, the agents dynamically select appropriate service elements, enactment, and configure the elements as parts of application workflows. Rudder implements the agent negotiation protocols and enables appropriate application behaviors to be dynamically negotiated and enacted. The agent protocols and activities are implemented using Comet coordination substrate. The Rudder system has been implemented, deployed, and evaluated. The experimental results on PlaneLab wide-area testbed and Rutgers campus networks demonstrate both the performance and scalability of Rudder. The autonomic oil reservoir optimization using Rudder is presented as an illustrative example.

Chapter 6

Summary, Conclusion, and Future Work

6.1 Summary

The primary objective of the research presented in this thesis was to investigate coordination infrastructures that address the complex coordination/computation requirements of Grid applications. The key contribution of this thesis is the Comet conceptual architecture model and implementation infrastructure, which provide coordination abstractions for supporting dynamic, scalable, and asynchronous applications in wide-area Grid environments. The Comet employs fully decentralized architecture and provides a global virtual shared-space abstraction that can be associatively accessed by all peer nodes in the system. The Comet space is constructed from a semantic information space that is deterministically mapped using a locality preserving mapping onto the dynamic set of peer nodes in the Grid system. The resulting Comet space maintains content locality and guarantees that content-based tuple queries are delivered with bounded costs. Comet is composed of layered abstractions prompted by a fundamental separation of communication and coordination concerns.

The Comet decentralized coordination infrastructure was developed to demonstrate the conceptual architecture model. Each peer node in Comet runs a stack with a communication layer, a coordination layer, and an application layer. The communication layer provides associative messaging services and manages system dynamism using a self-organizing overlay. The coordination layer implements Linda-like coordination primitives, by which all peers can associatively access the space without knowing the physical location or identifiers of the hosts. The application layer provides programming abstractions to enable application formulation and execution. The Comet infrastructure has been deployed and evaluated using Planetlab wide-area testbed as well as a

campus network at Rutgers University. The experimental results demonstrated the scalability and efficiency of Comet and the feasibility of wide-area Grid deployment.

Two prototype systems were developed using Comet. The first prototype was the CometG decentralized (peer-to-peer) computational infrastructure, which extended Desktop Grid environments to support parallel asynchronous applications. CometG provides coordination spaces and programming abstractions for parallel asynchronous iterative computations as well as an asynchronous formulation of the replica exchange algorithm for molecular dynamics applications. The second prototype was the Rudder coordination framework. Rudder provides software agents and coordination protocols for the dynamic discovery and selection of software service elements, enactment and configurations of workflows, and adaptations of these workflows to respond to changing Grid environments. The two prototype systems were deployed and evaluated on the PlanetLab testbed and a Rutgers campus network. The experimental results demonstrated the flexibility, scalability and effectiveness of the Comet infrastructure, as well as its ability to address complex coordination requirements of Grid applications.

6.2 Conclusion and Contributions

The emergence of wide-area Grid computing and Desktop Grid computing, based on the aggregation of large numbers of resources, is rapidly emerging as the dominant paradigm for distributed problem solving for a wide range of application domains. However, the heterogeneity, dynamism, and uncertainty of the underlying computing environment result in significant application development and management challenges, enabling flexible robust application coordination becomes a key issue. The shared-space based coordination model that provides temporal and spatial decoupling associative data access mechanisms can address most requirements of the Grid applications. However, realizing a scalable robust shared-space based coordination infrastructure for distributed wide-area environments presents several challenges.

This thesis presented the Comet decentralized coordination infrastructure for Grid environments. Comet implements a scalable, decentralized tuple space, which provides

Linda-Like primitives to enable communication and synchronization of distributed processes and software elements. The Comet employs peer-to-peer architecture and layered abstractions for prompting the separation of communication, coordination and programming concerns. The combination of these layers provides a flexible, efficient, scalable, and application-friendly support for Grid applications. The effectiveness of the Comet is demonstrated by two prototype application systems. The deployment and experimental evaluations of these systems illustrate the scalability and flexibility of Comet, and the effectiveness of using Comet to support applications in wide-area Grid environments.

Key contributions of this research are summarized below.

Comet Conceptual Architecture Model

Comet provides a global virtual shared-space constructed from the semantic information space used by entities for coordination and communication. This information space is deterministically mapped, using a locality preserving mapping, onto the dynamic set of peer nodes in the Grid system. The space builds on an associative DHT, which uses locality preserving mapping Hilbert Space Filling Curve (SFC) to map tuples/templates from the multi-dimensional information space to the one-dimensional peer indices. As a result, the Comet space maintains content locality and provides efficient content-based data queries.

Comet employs a layered architecture which promotes the separation of coordination, communication, and application programming concerns. The communication layer provides content-based routing, direct communication channels, and system dynamism management. The coordination layer implements Linda-like coordination primitives, by which all peers can associatively access the space without knowing the physical locations or identifiers of the tuples or hosts. The application layer provides programming abstractions and mechanisms for enabling application formulation and execution.

Comet Coordination Infrastructure

The Comet coordination infrastructure implements and demonstrates the architecture model. The Comet infrastructure provides tuples and templates in format of XML strings, which are lightweight, flexible, and suitable for efficient information exchange in distributed heterogeneous environments. It adapts the Squid information discovery scheme to construct the distribute hash table for tuple distribution and lookup. A peer node in Comet infrastructure runs a stack with three layers. The communication includes a content-based routing engine and a structured self-organizing overlay. The coordination layer include a data repository for storing, pending requests, and retrieving tuples, a flexible matching engine, and a message dispatcher that interfaces with the communication layer to convert the coordination primitives to messaging operations and vice versa. The application layer provides the CometG computational mechanisms for parallel asynchronous Grid computations, and Rudder coordination framework for composing component-based applications.

CometG Computational Infrastructure

CometG is a peer-to-peer computational infrastructure that extends Desktop Grid environments to support parallel asynchronous formulations of Grid computations. CometG builds on top of Comet tuple space and provides coordination space abstractions and programming modules to support master-worker/BOT parallel formulations of asynchronous computations. The coordination spaces support dynamic task distribution and management as well as inter-task communications. A programming module support an application-specific computational component that can locally compute a retrieved task, and contains task retrieval/submission mechanisms as well as interaction/negotiation protocols. Prototypes of parallel asynchronous iterative applications and parallel asynchronous replica exchange simulations have been developed to demonstrate the CometG system.

Rudder Coordination Framework

Rudder is a decentralized agent-based coordination framework for supporting the dynamic composition of Grid applications. Rudder provides software agents and coordination protocols for the dynamic discovery and selection of software element services, enactment and configurations of workflows, and the management and adaptations of these workflows to respond to changing Grid environments. The Rudder uses the workflow model as its basis for autonomic composing applications. Its agent interaction and negotiation protocols enable appropriate application behaviors to be dynamically negotiated and enacted. The major negotiation mechanisms include: Contract Net Protocol based element selection and marketplace model non-functional properties negotiation. The defined protocols and agent activities are supported by Comet coordination infrastructure.

6.3 Future Work

The directions for future extensions of this research are envisioned as below:

- Failure resilient tuple space infrastructure. A resilient tuple space coordination system must address both, the failures of coordinating application processes as well as the failures of the tuple space system. The current Comet provides system-level and application-level approaches for addressing crash failures and arbitrary execution delays of application processes. One major enhancement can be implementing resilient coordination algorithms using wait-free concurrent data objects [43]. The Comet architecture naturally supports the scalable implementation of wait-free data objects using a conditional atomic swap (**cas**) operator [80]. Using the **cas** operator, application developers can simply realize robust and efficient coordination behaviors using wait-free algorithms [43].

- Large-scale Desktop Grid based asynchronous parallel computations. Chapter 4 described the CometG computational infrastructure, which currently can support coordination groups consisting of tens to hundreds of peers. Further, each peer can run multiple instances of masters and/or workers. The scalability of CometG

can be potentially extended to thousands or even millions of nodes with the following enhancements: (1) separating the space nodes from end nodes, where the space nodes provide coordination services and the end nodes host the application program modules; (2) employing relatively powerful peers, i.e., super-peers, with larger memory capacity and network bandwidth, as space nodes and master nodes; and (3) using high-throughput task dispatch implementations such as the task servers popularly used by current Desktop Grid projects to support millions of users.

- Advanced workflow composition and control system for Grid applications. Chapter 5 presented Rudder coordination framework for dynamic composing pre-generated application workflows. It is possible that the application workflow needs be changed at runtime to address dynamically changing environments or user requirements. The possible changes can be adding, deleting, modifying a task or sub-flows so that workflow instances can be created on-the-fly and coordinated at runtime. The composition and coordination model based on asynchronous decoupled shared-space abstraction naturally supports the realization of these adaptations. Investigating and developing an autonomic workflow composition and control system on top of Rudder is a promising research direction.

References

[1] Climatprediction.net. http://climateapps2.oucs.ox.ac.uk/cpdnboinc/download_main.php.

[2] Document object model (dom) level 2 core specification. http://www.w3.org/TR/DOM-Level-2-Core.

[3] Folding@home. http://folding.stanford.edu/.

[4] The message passing interface (mpi) standard. http://www-unix.mcs.anl.gov/mpi/.

[5] mpijava. http://www.hpjava.org/mpiJava.html.

[6] Predictor@home. http://predictor.scripps.edu/.

[7] Project PlanetLab. http://www.planet-lab.org.

[8] Seti@home. http://setiathome.ssl.berkeley.edu/.

[9] Xtremweb. http://xw.lri.fr:4330/XtremWeb/.

[10] Web services flow language (wsfl) 1.0. http://www306.ibm.com/software /solutions/webservices/pdf/WSFL.pdf, 2001.

[11] Web services for business process design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.

[12] Business process execution language for web services (version 1.1). ftp://www6.software.ibm.com/ software/developer/library/ws-bpel.pdf, 2003.

[13] E. Gallicchio A. K. Felts, Y. Harano and R. M. Levy. Free energy surfaces of β -hairpin and α -helical peptides generated by replica exchange molecuar dynamics with agbnp implicit solvent model. 56:310–321, 2004.

[14] M. Agarwal and M. Parashar. Enabling autonomic compositions in grid environments. In *Proceedings of 4th International Workshop on Grid Computing (Grid 2003)*. IEEE Computer Society Press, 2003.

[15] G. Antoniu, P. Hatcher, M. Jan, and D. Performance Noblet. Evaluation of jxta communication layers. In *Proceedings of the Fifth International Workshop on Global and Peer-to-Peer Computing*, 2005.

[16] J. Bahi, S. Contassot-Vivier, and R. Couturier. Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(4):289–299, 2005.

[17] J. Bahi, S. Contassot-Vivier, R. Couturier, and F. Vernier. A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(1):4–13, 2005.

[18] Jacques M. Bahi, Stéphane Domas, and Kamel Mazouzi. Combination of java and asynchronism for the grid: A comparative study based on a parallel power method. In *Proceedings of 18th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2004.

[19] David Edward Bakken and Richard D. Schlichting. Supporting fault-tolerant parallel programming in linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, 1995.

[20] Gerard M. Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the ACM*, 25(2):226–244, 1978.

[21] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. Scientific American, May 2001.

[22] Dimitri P. Bertsekas and John N. Tsitsiklis. Convergence rate and termination of asynchronous iterative algorithms. In *Proceedings of the 3rd international conference on Supercomputing*, pages 461–470. ACM Press, 1989.

[23] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation:Numerical Methods*. Athena Scientific, 1997.

[24] V. Bhat, V. Matossian, M. Parashar, M. Peszynska, M. Sen, P.Stoffa, and M. F. Wheeler. Autonomic oil reservoir optimization on the grid. In *Concurrency and Computation: Practice and Experience, John Wiley and Sons (to appear)*.

[25] James C. Browne, Madulika Yalamanchi, Kevin Kane, and Karthikeyan Sankaralingam. General parallel computations on desktop grid and p2p systems. In *Proceedings of the 7th workshop on Workshop on languages, compilers, and runtime support for scalable systems*, pages 1–8. ACM Press, 2004.

[26] P. Buhler and J. Vidal. Towards adaptive workflow enactment using multiagent systems. *Information Technology and Management Journal: Special Issue on Universal Enterprise Integration*, 2004.

[27] Nadia Busi, Cristian Manfredini, Alberto Montresor, and Gianluigi Zavattaro. Peerspaces: data-driven coordination in peer-to-peer networks. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 380–386. ACM Press, 2003.

[28] Stefan Bussmann, Nicholas R. Jennings, and Michael Wooldridge. Re-use of interaction protocols for decision-oriented applications. In *Proceedings of 3rd Int Workshop on Agent-Oriented Software Engineering*, 2002.

[29] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Reactive tuple spaces for mobile agent coordination. *Lecture Notes in Computer Science*, 1477, 1998.

[30] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Xml dataspaces for mobile agent coordination. In *Proceedings of the 2000 ACM symposium on Applied computing*, pages 181–188. ACM Press, 2000.

[31] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–459, April 1989.

[32] Paolo Ciancarini, Robert Tolksdorf, Fabio Vitali, Davide Rossi, and Andreas Knoche. Coordinating multiagent applications on the WWW: A reference architecture. *IEEE Transactions on Software Engineering*, 24(5):362–375, May 1998.

[33] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[34] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[35] G. Cugola and G. Picco. Peerware: Core middleware support for peer-to-peer and mobile systems. Technical report, Politecnico di Milano, 2001.

[36] J.Waldo et al. Javaspace specification - 1.0. Techinical report, Sun Microsystems, 1998.

[37] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, June 2002.

[38] A. Frommer and D. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123(1):201–216, 2000.

[39] A.G. Ganek and T.A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal Special Issue on Autonomic Computing*, pages 5–18.

[40] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[41] C. Geyer and E. Thompson. J. am. stat. assoc.

[42] S. Gnanakaran H. Nymeyer and A. E. Garcia. Atomic simulations of protein folding using the replica exchange algorithm. 383:119–149, 2004.

[43] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, 1991.

[44] Fabrice Huet, Denis Caromel, and Henri E. Bal. A High Performance Java Middleware with a Real Application. In *Proceedings of the Supercomputing conference*, 2004.

[45] K. Hukushima and K. Nemoto. J. phys. soc. jpn. 65, 1996.

[46] Y. Cao A.E. Cho W. Damm R. Farid A.K. Felts T.A. Halgren D.T. Mainz J.R. Maple R. Murphy D.M. Philipp M.P. Repasky L.Y. Zhang B.J. Berne R.A. Friesner E. Gallicchio J.L. Banks, H.S. Beard and R.M. Levy. Integrated modeling program, applied chemical theory (impact). 26:1752–1780, 2005.

[47] J. Kephart, M. Parashar, V. Sunderam, and R. eds Das. Proceedings of the first international conference on autonomic computing. 2004.

[48] D. Kondo, M. Taufer, C. Brooks, H. Casanova, and A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2004.

[49] Erwin Kreyszig. *Advanced Engineering Mathematics*. Wiley, 1998.

[50] Sriram Krishnan, Patrick Wagstrom, and Gregor von Laszewski. GSFL: A workflow framework for grid services. http://www.globus.org/cog/papers/gsfl-paper.pdf, July 2002.

[51] R. Levy L. Zhang, M. Parashar and E. Gallichio. Salsa: Scalable asynchronous replica exchange for parallel molecular dynamics applications. In *Proceedings of The 2006 International Conference on Parallel Processing*, pages 127–134, 2006.

[52] Tobin J. Lehman, Stephen W. McLaughry, and Peter Wycko. T spaces: The next wave. In *HICSS*, 1999.

[53] Zhen Li and Manish Parashar. Rudder: An agent-based infrastructure for autonomic composition of grid applications. *International Journal Multiagent and Grid Systems*, 1(3).

[54] Zhen Li and Manish Parashar. Enabling dynamic composition and coordination for autonomic grid applications using the rudder agent framework. *The Knowledge Engineering Review*, 2006.

[55] Zhen Li and Manish Parashar. A decentralized agent framework for dynamic composition and coordination for autonomic applications. In *Proceedings of the 3rd International Workshop on Self-Adaptive and Autonomic Computing Systems*, August 2005.

[56] Zhen Li and Manish Parashar. A decentralized computational infrastructure for grid based parallel asynchronous iterative applications. *Journal of Grid Computing*, 4(4), December 2006.

[57] Zhen Li and Manish Parashar. Comet: A scalable coordination space for decentralized distributed environments. In *Proceedings of the 2nd International Workshop on Hot Topics in Peer-to-Peer Systems*, July 2005.

[58] Zhen Li and Manish Parashar. An infrastructure for dynamic composition of grid service. In *Proceedings of the 7th IEEE International Conference on Grid Computing (Grid 2006)*, page 315 316. IEEE Computer Society Press, September 2006.

[59] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Tuples on the air: A middleware for context-aware computing in dynamic networks. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*. IEEE Computer Society, 2003.

[60] E. Marinari and G. Parisi. Europhys. lett. 19, 1992.

[61] Brahim Medjahed, Athman Bouguettaya, and Ahmed K. Elmagarmid. Composing web services on the semantic web. *The VLDB Journal*, 12(4):333–351, 2003.

[62] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.

[63] A. Murphy, G. Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the 21 st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 524–536, 2001.

[64] Rob V. Nieuwpoort, Jason Maassen, Thilo Kielmann, and Henri E. Bal. Satin: Simple and efficient java-based grid programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, 2005.

[65] Philipp Obreiter. Extending tuple spaces towards a middleware for ecommerce. *Thesis ,University of Karlsruhe*, 2000.

[66] L. Oliveira, L. Lopes, and F. Silva. p^3: Parallel peer to peer an internet parallel programming environment. In *Proceedings of the Workshop on Web Engineering and Peer-to-Peer Computing*, 2002.

[67] Andrea Omicini and Franco Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999.

[68] OWL Matcher. http://ivs.tu-berlin.de/Projekte/owlsmatcher/.

[69] http://www.daml.org/services/owl-s/1.1, 2004.

[70] M. Parashar and J.C. Browne. Conceptual and Implementation Models for the Grid. In *Proceedings of the IEEE, Special Issue on Grid Computing*, volume 93, pages 653–668, 2005.

[71] M. Parashar and S. Hariri. Autonomic computing: An overview. 2005.

[72] A. Plaat. Optimizing parallel applications for wide-area clusters. In *Proceedings of the 12th International Parallel Processing Symposium*. IEEE Computer Society, 1998.

[73] Project JXTA. Internet: http://www.jxta.org.

[74] Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[75] D. Roure, D. Jennings, and N. Shadbolt. The semantic grid: Past, present, and future. In *Proceedings of the IEEE*, volume 93, 2005.

[76] K. Y. Sanbonmatsu and A. E. Garcia. Proteins. 46, 2002.

[77] Karthikeyan Sankaralingam, Madhulika Yalamanchi, Simha Sethumadhavan, and James C. Browne. Pagerank computation and keyword search on distributed systems and p2p networks. *Journal of Grid Computing*, 1(3):291–307, 2003.

[78] Cristina Schmidt. *Flexible Information Discovery With Guaratees In Decentralized Distributed Systems*. PhD thesis, Rutgers University, 2005.

[79] Cristina Schmidt and Manish Parashar. Enabling flexible queries with guarantees in p2p systems. *IEEE Network Computing, Special issue on Information Dissemination on the Web*, (3):19–26, June 2004.

[80] Edward J. Segall. Resilient distributed objects: Basic results and application to shared tuple spaces. In *SPDP 1995*.

[81] C. Sierra, P. Faratin, and N. Jennings. A service-oriented negotiation model between autonomous agents. In *Proceedings of 8th European Workshop on Modeling Autonomous Agents in a Multi-Agent World*, 1997.

[82] R. G. Smith. The contract net protocol: high-level communication and control in a distributed problem solver. *Distributed Artificial Intelligence*, pages 357–366, 1988.

[83] H. De Sterck, R. S. Markel, T. Phol, and U. Rüde. A lightweight java taskspaces framework for scientific computing on computational grids. In *Proceedings of the ACM symposium on Applied computing*, pages 1024–1030. ACM Press, 2003.

[84] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[85] Y. Sugita and Y. Okamoto. Replica-exchange molecular dynamics method for protein folding. 314:141–151, 1999.

[86] R. Swendsen and J. Wang. Phys. rev. lett. 57, 1986.

[87] Stefan Tang. Matching of web service specifications using daml-s descriptions. Thesis, March, 2004.

[88] Robert Tolksdorf and Dirk Glaubitz. Coordinating web-based systems with documents in xmlspaces. In *Proceedings of the Sixth IFCIS International Conference on Cooperative Information Systems*, pages 356 – 370. Springer Verlag, 2001.

[89] W. Bangerth H. Klie V. Matossian, M. Parashar and M. F. Wheeler. An autonomic reservoir framework for the stochastic optimization of well placement.

[90] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2004.

[91] Chris Wroe, Robert Stevens, Carole Goble, Angus Roberts, and Mark Greenwood. A suite of daml+oil ontologies to describe bioinformatics web services and data. *International Journal of Cooperative Information Systems*, 2003.

[92] Narendar Yalamanchilli and William W. Cohen. Communication performance of java-based parallel virtual machines. *Concurrency - Practice and Experience*, 10(11-13):1189–1196, 1998.

[93] Y. Zhang, Xiaoye S. Li, and O. Marques. Towards an automatic and application-based eigensolver selection. In *LACSI Symposium*, 2005.

Vita

Zhen Li

05/2007 Ph. D. in Electrical and Computer Engineering, Rutgers University, New Brunswick, NJ, USA

04/2002 M. E. in Computer Engineering, Beijing University of Posts And Telecommunications, Beijing, China

07/2000 B. S. in Computer Science, Zhengzhou University, Zhengzhou, China

09/2005-04/2007 Graduate Assistant, CAIP, Rutgers University, NJ, USA

09/2002-07/2005 Teaching Assistant, Department of Electrical and Computer Engineering, Rutgers University, NJ, USA

09/1999-04/2002 Research Assistant, National Networks Lab, Beijing University of Posts and Telecommunications, Beijing, China

Journal Publications

"Rudder: An Agent-based Infrastructure for Autonomic Composition of Grid Applications," **Z. Li** and M. Parashar, *International Journal Multiagent and Grid Systems*, 1(3), 2005.

"Enabling Dynamic Composition and Coordination for Autonomic Grid Applications using the Rudder Agent Framework," **Z. Li** and M. Parashar, *The Knowledge Engineering Review*, 2006.

"A Decentralized Computational Infrastructure for Grid based Parallel Asynchronous Iterative Applications," **Z. Li** and M. Parashar, *Journal of Grid Computing*, 4(4), Springer-Verlag, 2006.

Conference Publications

"An Infrastructure for Dynamic Composition of Grid Services," **Z. Li** and M. Parashar, *Proc. of the 7th IEEE/ACM International Conference on Grid Computing*, 2006.

"Comet: A Scalable Coordination Space in Decentralized Distributed Environments," **Z. Li** and M. Parashar, *Proc. of the 2nd HotP2P International Workshop*, July 2005.

"A Decentralized Agent Framework for Dynamic Composition and Coordination for Autonomic Applications," **Z. Li** and M. Parashar, *Proc. of the 16th International Workshop on Database and Expert System Applications*, Copenhagen, Denmark, August 2005.

"Rudder: A Rule-based Multi-agent Infrastructure for Supporting Autonomic Grid Applications," **Z. Li** and M. Parashar, *Proc. of 1st International Conference on Autonomic Computing*, May 2004.

"Enabling Autonomic Grid Applications: Dynamic Composition, Coordination and Interaction," **Z. Li** and M. Parashar, *UPP 2004*, Springer Verlag, Vol. 3566, 2005.

"Enabling Autonomic Self-Managing Grid Applications," **Z. Li**, H. Liu and M. Parashar, *Proc. of International Workshop on Self-* in Complex Information Systems*, 2004.

"AutoMate: Enabling Autonomic Grid Applications," M. Parashar, H. Liu, **Z.** Li, V. Matossian, C. Schmidt, G. Zhang and S. Hariri, *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, Special Issue on Autonomic Computing, Kluwer Academic Publishers, Vol. 9, No. 1, 2006.

"Enabling Autonomic Grid Applications: Requirements, Models and Infrastructures," M. Parashar,**Z.** Li, H. Liu, V. Matossian and C. Schmidt, *Self-Star Properties in Complex Information Systems*, Lecture Notes in Computer Science, Springer Verlag, Vol. 3460, 2005.

"AutoMate: Enabling Autonomic Applications on the Grid," M. Agarwal, V. Bhat, **Z.** Li, H. Liu, B. Khargharia, V. Matossian, V. Putty, C. Schmidt, G. Zhang, S. Hariri and M. Parashar, *Proc. of the Autonomic Computing Workshop*, 5th Annual International Active Middleware Services Workshop, 2003.