Coordinated Collaboration for E-commerce Based on the Multiagent Paradigm

LEE Ting-on

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

September 2000


香港中文大學
圖書館系統
28 JUN 2001
UNIVERSITY
LIBRARY SYSTEM
THE CHINESE UNIVERSITY OF HONG KONG

Acknowledgments

First and foremost, I would like to give my grateful thanks to my supervisor, Prof. Kam-Wing Ng for his guidance throughout the two years of my postgraduate study. If it had not been for Prof. Ng's insightful advice and his patient guidance, my research would not have conducted reasonably. Prof. Ng's precise and thorough review on our published papers and this thesis is most appreciated.

Next, my gratitude goes to Prof. Moon-Chuen Lee and Prof. Ho-Fung Leung who have given me precious comments and suggestions as they marked my term papers; and Prof. Boon-Toh Low who inspired me on thinking and research as he supervised my final year project during my undergraduate study.

Many thanks to Ka Ho Mak, my final year project partner who kept giving me helpful advice on various aspects evey time we met; my fellow postgraduate students, Hing-Wing Chan, Kwong-Wai Chen, Ka-Lung Chong, Yin-Hung Kuo, Tsui-Ying Low, Wing-Kai Lam, Ka-Po Ma, Kam-Po Shan, Xiao-Qing Wang, Siu-Ham Wong, Wai-Ching Wong and Wai-Chiu Wong, who shared my joy and sadness in the postgraduate program, and dear Katso members who filled my emptiness and gave me encouragement.

Finally, I take this chance to give my gratitude to my family for giving me support ever since I was born. Thanks for giving me their love and care that lead to all the achievements I have attained thus far.

Coordinated Collaboration for E-commerce Based on the Multiagent Paradigm

submitted by

LEE Ting-on

for the degree of Master of Philosophy
at the Chinese University of Hong Kong

Abstract

The Internet has been expanding rapidly over the recent decades as are the activities conducting over the World Wide Web. The complexity of online services grows along with the increasing population online. The robustness of network applications and distributed systems can no longer be sustained by the traditional distributed programming approaches in an effective manner. For this reason, the software agent paradigm has emerged as a promising methodology to resolve complex distributed computation problems at high scalability.

As more research attention is being drawn on the software agents, the multiagent paradigm stems from employing multiple agents to add further capabilities and performances to distributed systems. Although research over multiagent systems has emerged in recent years to formulate open, flexible and scalable solutions to large-scale distributed problems such as WWW information retrieval, data mining and

electronic marketplace, the full potential of the multiagent paradigm has yet to be revealed, as most of the major mobile agent frameworks only provide primitive support for inter-agent communication. The implementation of any collaboration architecture is up to the system developers' responsibility.

In this thesis, we present a **Co**mponentware for **D**istributed **A**gent **C**ollaboration (CoDAC) as a solution to general agent coordination problems. CoDAC utilizes the component model to offer flexible and reliable coordination support to mobile agents distributed over the network. The major contribution of CoDAC is to embed atomic commitment capabilities into the collaboration among distributed agents with enhanced fault tolerance.

多代理爲本電子商務之協調協作

作者: 李定安

論文摘要

隨著網上的活動日益繁忙, 互聯網絡在近數十年正經歷激烈的膨脹。同時各類網上服務的內容亦隨網絡用戶的增長而日趨複雜。傳統的離散計算範型已難以高效率地應付當前的網上動態離散環境, 爲此代理範型急速冒升成爲理想的離散計算技術。

藉著代理範型方面多年的研究成果, 近年多代理系統的學術研究發展迅速,其應用範圍主要針對計算機網絡上離散以及大規模的難題,例如互聯網上分佈信息的搜索, 數据發掘以及電子商業等各方面提供靈活的解決方案 但是目前大部份主導的移動代理結構模型對多代理協作方面只提供有限的輔助,令多代理計算範型的潛能未能盡顯。

在這篇論文中,我們提出一個輔助多代理在分散環境下協作的部件 CoDAC 作爲多代理系統的骨幹並爲該類系統提供具彈性的支援, 其主要貢獻在於將完整及容錯的特性整合在多代理協作系統之上。

Contents

List of Figures

List of APIs

List of Tables

Chapter 1

Introduction

The Internet has been expanding rapidly over the recent decades driven by a wide range of activities conducted over the World Wide Web. For instance, business organizations perceive the Web as a potential market that could boost sales at comparatively low cost. As a result the electronic marketplace has emerged as the key growing entity over the Internet. However, the complexity of online services grows along with the increasing population online. The robustness of network applications and distributed systems can no longer be sustained by the traditional distributed programming approaches in an effective manner. In particular, poor network qualities and information overload impose indispensable burden on system performance. For this reason, the software agent paradigm has emerged as a promising methodology to resolve complex distributed computation problems with high scalability.

Thanks to the mobile agent paradigm, we experience a breakthrough to move the process to the data source. This mobility of agent (the software process) brings benefits in many ways. An agent continues to operate even if it is temporarily disconnected from the network as it essentially performs its operation locally at the data source. In fact, an agent can be kept offline and immune to any harm caused by

network latency for most of the time of its execution. In addition, it utilizes the limited bandwidth by sending only the relevant results over the network. All these benefits justify the deployment of agents in the distributed computation environment.

As more research attention has drawn on the software agents, the multiagent paradigm stems from employing multiple agents to add further capabilities and performances to distributed systems. The multiagent paradigm further unravels the potential of software agents in realizing various attractive goals. For example, more elaborated services can be provided from a group of cooperating agents, each implementing different logic to address different needs and to simulate different behaviors. These agents represent different interests and negotiate with each other to find out the optimal solution for the best interest of all involved parties. Further, multiagent systems utilize the autonomy of software agents to facilitate parallel processing where a comprehensive work can be divided into several component tasks, each performed by an individual agent concurrently so as to increase system throughput. Further, replicated service agents can be employed to offer high flexibility and fault tolerance.

Recent research over multiagent application focuses on formulating open, flexible and scalable solutions to large-scale, distributed problems such as WWW information retrieval, data mining and electronic marketplace. In particular, the multiagent paradigm is drawing increasing interest over the areas of e-commerce [MGM99, GTB99], virtual enterprises and intelligent manufacturing [JAS99], scientific computing [DHRR99], home automation [Run99], and network communities [HOY+99].

As long as multiple distributed objects (e.g. software agents) are engaged in some kind of global behavior, the concept of knowledge reasoning [HM90] is concerned. Reasoning about knowledge plays a fundamental role in distributed systems, where communication within the system can be viewed as the act of transforming the system's state of knowledge. For instance, agents can only base their actions on their local information. This knowledge, in turn, depends on the messages they receive and the events they observe. Thus, there is a close relationship between knowledge

and action. When we consider the task of performing coordinated actions among multiple agents in a distributed environment, it does not suffice to consider only individual agent's knowledge. Rather, we need to look at the states of knowledge of the groups of agents. Attaining particular states of group knowledge is a prerequisite for performing coordinated actions of various kinds.

Common knowledge [HM90] corresponds to the facts that are universally known. Therefore, reaching a common knowledge (i.e. the strongest stage of group knowledge) is essential for execution of simultaneous or consistent actions within the group.

Mole [SBH96], as a fore-runner in embedding the exactly-once semantics in the mobility of agents, presents a protocol suite to guarantee an agent to be executed exactly once with enhanced fault tolerance in the reduction of risk on an agent to be blocked. This model enforces the common knowledge among the set of agent execution environments, denoted as nodes, in order to solve the blocking problem.

In this model [RS97], the task of each agent performs in a sequence of steps. A step corresponds to the action performed on the local resources as an agent visits an individual network node. As an agent often has to visit several network nodes to accomplish its task, which step the agent has to perform on which node and the order in which the steps have to be performed is described by an itinerary, which may be adapted during the execution of the agent [SRM98]. The itinerary is constituted of stages, where each stage consists of a nonempty set of nodes that can alternatively serve the agent. Each node in a stage assumes either one of two roles, worker or observer. Only one worker exists in each stage at a time and the execution of an agent associates to a stage the set of operations performed by the agent while it visits the worker of this stage. This set of operations is treated as a transaction (i.e. a step transaction). Observers simply serve as replacement in case the worker crashes.

When an agent completes a step, the agent object with the code and all private data belonging to the object are captured and transferred to the nodes associated with the next stage (i.e. both worker and observers). There, it is re-instantiated at the worker and the step to be performed on this node is executed.

To provide reliable agent execution, the agent is executed using the protocols for providing the exactly-once property of mobile agents presented in [RS98], namely the monitoring protocol, the selection protocol and the voting protocol.

Although the selection protocol selects a new worker among the available observers when the current worker is suspected to have failed in the monitoring protocol, this protocol does not enforce a strong state of knowledge and may turn out with multiple selected workers and hence duplicated step transactions. Therefore, the voting protocol is designed to preserve the exactly-once semantic of the step transaction. This protocol attains common knowledge by requesting all stage nodes to vote for or against the commitment of a step transaction associated to a worker. If a majority of the stage nodes agree with the worker to commit, then that step transaction can commit mutual exclusively whereas other outstanding workers must abort. In this sense, this protocol has enforced the common knowledge on exactly which node commits the step within a stage.

Inspired by this model, we intended to design a tool for solving distributed coordination problems. Clearly, the Mole model only deals with one specific coordination problem, that is, the exactly-once commitment with added fault tolerance to a step and, thereby, the entire task of an agent as a whole. Rather, we develop our tool for enforcing common knowledge atop of fundamental collaboration practices in multiagent environments. Since, the coordination effort in the Mole model is imposed on the execution environments (i.e. the stage nodes), this causes certain platform dependency on the mobile agents. In order to eliminate platform dependency, we decided to integrate the coordination capability into the software agent itself on top of standard Java facilities.

In this thesis, we will present a **Co**mponentware for **D**istributed **A**gent **C**ollaboration (CoDAC) as a solution to general agent coordination problems. CoDAC utilizes the component model to offer flexible and reliable coordination service to mobile agents distributed over the network. It takes advantages of the Jini infrastructure [Sun99a] in order to be deployable with plug-and-play capability at runtime. CoDAC encapsulates its constituent features with respect to the

enforcement of common knowledge and interacts with agents through well-defined interfaces. It features modularized and interchangeable building blocks for multiagent systems. On top of that, it exercises the self-managing property to manage its own resources and adds no management burden on the associated agents.

Beyond the attainment of common knowledge, CoDAC adds flexibility and reliability into the coordination framework. For instance, CoDAC boosts flexibility to a new extent, as it breaks the gap between different agent platforms. With its strong compatibility, CoDAC can bring heterogeneous agents implemented and operating in different agent platforms together to engage in collaborations. Above all, CoDAC offers the core functionality to manage the groups of agents regardless of their heterogeneity. These groups are managed with enhanced reliability in a way that failures within a group will be self-recovered in a timely fashion. In particular, the coordination center can shift from one agent to another in a controlled manner when failure occurs in certain members. Furthermore, CoDAC presents a hierarchical group infrastructure which adds scalability to multiagent systems as the coordination effort decentralizes throughout the hierarchy where dynamic changes in the group membership can be handled effectively at the local domains.

CoDAC is a comprehensive tool for the multiagent paradigm, as it has not only addressed the coordination issues in multiagent collaboration, but has also enhanced such crucial factors as flexibility, reliability and scalability in support for large-scale open multiagent systems.

1.1 Roadmap to the Thesis

In this thesis we will present the design issues and the coordination mechanisms implemented in CoDAC. To begin with, we first have a brief introduction to the software agent paradigm in Chapter 2. We will introduce the key players in this

paradigm and discuss their implications on the distributed environment. In Chapter 3, we give a survey on existing coordination models in various agent frameworks.

In the following chapters, we go into the design issues of the CoDAC framework. Chapter 4 presents the standard facilities that serve as the foundation for the design and implementation of CoDAC. In Chapter 5, we identify the key requirements in the multiagent paradigm and explain the system infrastructure of CoDAC. Chapter 6 describes the collaboration model and the protocol suite that entails the coordination mechanisms in CoDAC will be explained in Chapter 7.

Chapter 8 details the implementation of CoDAC follows with an example for illustration in Chapter 9. Chapter 10 summarizes the characteristics of the collaboration framework implemented in CoDAC as the key contributions it delivers. We complete the thesis with a summary of the contributions and a discussion of future works in Chapter 11.

Chapter 2

Software Agents and Agent Frameworks

The software agent has emerged in the last decade as a promising solution to distributed computation problems like poor network quality, limited bandwidth and network legacy, etc. In this chapter, we introduce the key entities in the software agent paradigm. First, we give a definition to software agent and discuss its implication to the distributed environment in section 2.1. Next, we define an agent framework and identify the common communication facilities available in such frameworks in section 2.2 and 2.3 respectively. In section 2.4, we define the component concept which plays a key part in agent frameworks.

2.1 Software Agents

A software agent, in nature, is a computer program. However, the boundary between the two is not precisely defined. Current research offers a variety of definitions on

the concept of software agent, yet there is still no systematic way to distinguish between an agent and a program. In summarizing the many ways of describing a software agent, we come up with a set of properties shared among typical agent applications. A program may be usefully qualified as an agent according to this set of properties that it may possess. These properties include:

Autonomy: One of the features of an agent that draws most attention is the autonomy [FG96] it possesses. Agents are self-contained independent software entities that execute continuously and autonomously in attaining their goals on behalf of the end-users or other program entities without direct intervention by human. Agents act pro-actively to take the initiative roles to accomplish their tasks with authority granted by the user.

Reactive and goal-oriented: Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks [Maes95] for which they are designed. An agent can be viewed as perceiving its environment through sensors and acting upon that environment through effectors [RN95].

Temporally continuous: Every agent acts continually over some period of time [FG96]. A software agent, once invoked, typically runs until it decides not to. In some cases, human can kill an agent mandatory, but in most cases, human intervention is undesirable. For example, mobile agents on the Internet may be beyond calling back by the user.

Flexible and adaptive: Actions taken by agents are not scripted, instead, they are driven by some knowledge or representation of the users' goals or desires in harmony with the dynamic conditions in the environment [OW94]. Further, these actions taken will affect conditions in the environment, changing what agents will sense in the future and thereby effecting how the agents act subsequently.

Communicative: Social ability [WJ95] is another key feature an agent possesses to facilitate task accomplishment. Agents equipped with this ability are capable to interact with one another via some kind of agent-communication language, wherein

participating in collaborative operations. Software agents typically exercise their social ability to engage in dialogs and negotiations, and to coordinate transfer of information.

Mobility: Mobile agents, in particular, possess the ability to migrate from one host to another. As an agent migrates, it is not only the code but also the state [BHRS97] of the agent that has to be transferred to the destination. An agent may possess a predefined itinerary at compile-time or decide on its next destination at runtime [RS97] so as to accomplish their tasks on various data sources.

Property	Meaning
Autonomous	Exercises control over its own actions
Reactive	Responds in a timely fashion to changes in the environment
Goal-oriented	Does not simply act in response to the environment
Temporally Continuous	Is a continuously running process
Social Ability	Communicates with other agents
Adaptive	Changes its behavior based on its previous experience
Flexible	Actions are not scripted
Mobile	Able to transport itself from one machine to another

Table 2.1: Properties of software agent

The above properties are summarized in Table 2.1. Satisfying the first four properties, namely, autonomous, reactive, goal-oriented and temporally continuous, qualifies a computer program as an agent in general. Fulfillment of other additional properties produces potentially more useful classes of agents. For example, mobile

agents inherit the mobile property, whereas learning agents inherit the adaptive property. A program justified as an agent utilizes these properties to pursue its goals.

2.1.1 Advantages of Agents

In the following, we examine the advantages of the software agent paradigm in terms of the properties described in the previous section. In particular, we focus on the advantages delivered from the mobility and autonomy of an agent.

1. To facilitate high quality, high economical mobile applications: Applications employing mobile agents transparently utilize the network to accomplish their tasks, while taking full advantage of resources local to their host machines in the network. Instead of fetching data remotely, agents perform their operations at the data source, wherein enhancing higher performance with reduction on communication cost in terms of the number of remote interactions and the amount of data transmitted over the network. An overall improvement is justifiable if the performance gains exceed the extra overhead for transferring the agents.
2. To facilitate software-distribution on demand: In traditional client-server systems, new code has to be installed manually by users or system operators. The installation is sometimes rather challenging and often requires detailed knowledge about the current state of the computer system. The software-distribution on demand [BHRS97] paradigm emerges as an easier installation alternative, which not only able to transport code, but also to install packages automatically. For instance, a mobile agent system offers similar services as it utilizes platform-independent languages like Java to deliver programs in forms of mobile agents to the clients, which embrace an environment to install and execute these modules.

3. To utilize low bandwidth, high latency, error prone communications channels efficiently and economically: The agent network employs a store and forward mechanism to transfer agents between nodes. This is well suited to the problematic nature of many communications channels, especially in the mobile arena. Queuing and persistent checkpoints enhance this further, to the point that agents can use such channels with no degradation in reliability or response. For example, the client part of the application can be transferred, as an agent, from a mobile device to stationary servers in the network. Not only the individual requests are sent to the network, but also the entire task is moved to the data source where it is performed asynchronously. Once the task transfer is complete, the mobile device can be disconnected from the network. Some time later, the device can reconnect to receive the results of the task. As the data processing takes place locally at the source, the network has no effect on the agent as it executes.

2.1.2 Roles of Agents

As seen in section 2.1, different agents can inherit different sets of properties, resulting in a hierarchical classification based on set inclusion. On the other hand, in most common agent applications, where heterogeneous components can inter-operate, the participating agents assume a variety of roles. These agents are differentiated from one another by the roles they take and can be classified into the following categories [HS98a]:

User agent: a user agent acts as an intermediary between the user and the system, providing access to such resources as data analysis tools, workflows and concept-learning tools. It supports a variety of interchangeable user interfaces (eg. query forms, graphical query tools, etc), result browsers and visualization tools.

Broker agent: Broker agents implement directory services for locating appropriate agents with appropriate capabilities. They manage a namespace service and may store and forward messages and locate message recipients. Brokers might also function as communication aids by managing communications among the various agents, databases, and application programs in an environment.

Resource agent: Resource agents provide access to information stored in legacy systems, among which three common types are classified by the resource they present. *Wrapper agents* implement common communication protocols and translate commands and results into and from local access languages. For example, a wrapper agent may use a local data-manipulation language such as SQL to communicate with a relational database. *Database agents* manage specific information resources, and *data analysis agents* apply machine learning techniques to form logical concepts from data or use statistical techniques to perform data mining.

Execution agent: Execution agents are implemented as rule-based knowledge systems. They supervise query execution, operate as script-based agents to support scenario-based analysis, or monitor and execute workflows. A *mediator agent* is a specialized execution agent that works with brokers to determine which resources might have relevant information. It also decomposes queries to be handled by multiple agents and combines the partial responses obtained from multiple resources.

Security agent: Security agents provide system-wide authentication and authorization, and can be used to enforce appropriate usage policies for system resources.

Such variety of agents embodies diverse knowledge, reasoning approaches and perspectives. They represent people or business interests that have different goals and motivations and collaborate as a whole that constitute the system backbone.

2.2 Agent Frameworks

A framework [Lewa98] is a tool for managing a system of interacting objects and for developing objects that will integrate seamlessly into the framework. The common goal of every framework is to enhance well maintainable and consistent software systems. This goal is attained through standardization on the patterns of collaboration between the objects that constitute the framework, such that every component inside a framework shares consistent design attributes, and may even share common implementations.

For instance, object-oriented frameworks allow the highest common abstraction level between a number of similar systems to be captured in terms of general concepts and structures. Hence, a framework is essentially a large design pattern that captures the essence of one specific kind of object system along with the elements common to a family of the relevant systems. The bulk of the system functionality is captured in the framework, which is maintained as a single entity. Each software system that builds atop a framework is an instantiation of that framework.

It follows that an agent framework can be viewed as a tool which entails an abstract design for agent-based systems. It standardizes the abstract interfaces for which the agents and other entities within the system must conform in order to be integrated seamlessly into the system and to utilize the basic common services provided by the underlying facilities and middlewares.

In particular, a mobile agent framework is an infrastructure that supports the mobile agent paradigm. Examples of mobile agent frameworks include Aglet [OK97], Ajanta [KT98], Concordia [WPW97], Grasshopper [IKV98], etc. Although the architectures in different mobile agent frameworks are different in their implementation, the core functionality delivered by each framework is more or less the same. For instance, each mobile agent framework must provide a hosting environment for the agents, a space for which an agent resides, executes and interacts with other entities within the system. For examples, the so-called aglet context

[OK97] in Aglet serves as warehouse or workplace where aglets can communicate with each other, locations are offered in Mole [SBH96] for agents to execute upon. Similarly, the agent server [KT98] and the agent manager [WPW97], in Ajanta and Concordia respectively, serve the same purposes. These execution environments (commonly known as places) typically implement the transfer protocols to offer basic mobility support needed for agents. They perform the serialization and deserialization of the agent codes, and recover their internal states.

On top of each place, there is a variety of components known as services, that provides a set of common basic services including naming and trading, messaging, security and access to various resources. For example, the directory manager [WPW97] in Concordia maintains a registry of application services and enables mobile agents to locate the application servers they wish to interact with on each host, the security managers in Ajanta [KT98], AMETAS [ZMG98] and Concordia are responsible for authenticating and authorizing the received agents, monitoring agents' behavior and granting the privileges to access system resources whereas the communication service [IKV98] in Grasshopper supports location-transparent interactions between agents, places and non-agent-based entities.

Services are typically employed as proxies for the systems resources, which shield the underlying resources against direct access from agents. These proxies serve the requests from agents, verify the requests based on the security policy, direct any justified requests to the actual resources and finally return the results to the agents. All mobile agents must rely on interfacing with these proxies to gain access to system resources indirectly. This serves as a primitive solution to protect the host against malicious agents.

2.3 Communication Services and Concepts

In section 2.1, we have identified social ability as one of the qualifying properties to be an agent. This property enables an agent to communicate with one another and even to engage in collaborations. In this section, we will see how agents exercise this capability. In particular, we address the various types of communication services implemented in well-known agent platforms, namely Ara, Aglet, Ajanta, Concordia, Grasshopper and Mole.

For instance, each type of communication identified among these seven platforms can be classified as either one of the three categories: message channel, remote procedure call and event channel.

2.3.1 Message Channel

Message channels implement the basic form of communication, message exchange, between different agents. Messages are implemented in the form of objects and typically have an arbitrary object as its argument that stores the actual content of the message. As an agent wants to talk to another agent, it has to create a message object first, and then send it to the peer. The incoming messages are stored in a queue before they are being processed one by one. The receiver agent can determine what to do by checking the type of the received message.

Message channels are advantageous in terms of the simplicity to trace as well as the flexibility to extend. In particular, various agent communication languages such as KQML and KIF can be implemented readily on top of message channels.

In Aglet [OK97], messages can be transmitted on both local and remote scales. In particular, the content of the messages passed by remote messaging must implement the java.io.Serializable interface, such that it could be marshaled and unmarshaled by the Java object serialization facilities. Sending a remote message is different from dispatching an aglet in the sense that a remote message does not cause any transfer of

bytecode, and therefore the classes used in the message have to be installed in both hosts.

The messaging mechanism in Mole [MJF96] is developed for indirect data-oriented inter-agent communication, which can either be synchronous or asynchronous. A message is sent from an agent to the location (the hosting place of the agent) specifying the addresses of both sender and receiver. The destination location will thereby direct the message to the receiving agent if that receiver exists. Otherwise the message will be queued and sent back to the sender after a timeout.

An asynchronous remote messaging facility is available in Ara [HT97] for simple status reports, error messages and acknowledgments. Each message is addressed to one or more agents by their names that consist of a unique id, an identification of their principal and an optional symbolic name from a hierarchical name space. The message will be delivered to all agents at the indicated place whose names are subordinates of the indicated recipient name in terms of the hierarchical agent name space. This address scheme is applicable for place-wide message multicast or application-level transparent message forwarding. However, in order to avoid remote coupling, this messaging facility does not guarantee against any message losses.

2.3.2 Remote Procedure Call

Remote procedure call facilitates direct action-oriented synchronous communication, in which the flow of control will be transferred from the caller agent to the callee until the request is served and the results are returned. Only the public method of the callee can be remotely invoked and any such method would be executed concurrently to the callee's normal control flow. Obviously, the callee must never migrate during which the RPC is executing.

Most frameworks implement the RPC mechanism based on the Java RMI facility, for example, Mole [SBH96], Ajanta [KT98] and Grasshopper [IKV98]. An agent

wishing to make itself available for remote invocation specifies the interface that it intends to support, and install an RMI proxy in the local RMI registry.

When a remote entity wishes to communicate with such an agent, it searches the RMI registry for the RMI stub for the agent. The stub passes RMI calls through to the agent object and relays the results back to the caller. The RMI calls are not necessarily applied for mere remote communication purpose, they are also utilized for local communication among agents on the same host. As the communication is location-transparent, there is no difference between remote method invocations and local method invocations within the agent code.

Above all, the RPC communication is not limited to the Java RMI facility in particular. The communication service in Grasshopper supports, as well, the Internet Inter-ORB Protocol (IIOP) and provides its OMG MASIF-compliant CORBA interfaces for remote interactions.

2.3.3 Event Channel

The distributed event model provides non-session-oriented communication channels that enable anonymous communication among agents without the need to specify the identities of the communication partners in advance. The service of an event channel is essentially operated by an event manager, which is responsible for accepting event registrations, listening for and receiving events, and notifying the interested parties of each event it receives. Each agent must register with the event manager such that it can forward the appropriate events to the subscriber.

In Concordia [WPW97], a customizable communication channel is provided for individual agent as Selected Events. Each agent registers with the event manager and specifies a set of event types it intends to receive such that the event manager will deliver the subscribed events only.

On the other hand, Concordia implements group-oriented events to provide a channel for agents within an application to communicate and collaborate with each

other in which all the involved events are delivered to this group of agents without filtering. All agents intended to receive group-oriented events need to register with the event manager to join a group beforehand. Whenever the event manager receives an event from any member, it forwards the event to all other agents in the group.

2.4 Components

Components [Lewa98] are the smallest self-managing, independent, and useful parts of a system that can be replicated, customized, and inserted into application programs. Components promise rapid application development and a high degree of customizability for end users, leading to fine-tuned applications that are relatively inexpensive to develop and easy to learn. Components come in a variety of different implementations to support a wide range of functions designed for use in a variety of systems and to provide reliable services regardless of context. Numerous individual components can be created and tailored for different applications.

Components are most often distributed objects incorporating advanced self-management features. Such components rely on robust distributed-object models so as to maintain transparency of location and implementation. Components may contain multiple distributed or local objects, and they are often used to centralize and secure an operation. As the implementation of a component is transparent to the application developers, one needs only to identify the function of this component and the means of invoking this behavior via the interface before reusing it. Interaction with components typically occurs through event handling and method invocation.

Components revolutionize the development of scalable systems by featuring as modularized and interchangeable building blocks. Advanced architectures offer the end user the ability to add components, allowing simple customization of applications.

Self-managing components take responsibility for their own resources, work across networks and interact with other objects. These capabilities are frequently given to components through a distributed object framework that acts as a middleware to regulate the necessary inter-object communications and provides a resource pool for each component.

Components are used easily by other objects since no management burdens are imposed on the client object. Component objects rely on a solid event model that allows objects to broadcast specific messages and generate certain events. These events signal those listening objects to take appropriate actions accordingly. Each listening object responds to a given event in its own manner. By using object-oriented techniques such as polymorphism, closely related objects react differently to the same event. These capabilities simplify the programming of complex client/server systems and also help provide an accurate representation of the real-world system modeled.

Chapter 3

Related Work

Research over multiagent systems has emerged in recent years to formulate open, flexible and scalable solutions to large-scale, distributed problems such as WWW information retrieval, data mining and electronic marketplace. Although coordination models [Adl95] have been studied extensively in the past, mobility and the openness of the mobile agent paradigm introduce new problems and needs. In this chapter, we first look into a simple taxonomy of the coordination models in practice in section 3.1. Next, we will discuss the pros and cons of each model identified in this taxonomy in section 3.2.

3.1 Collaboration Behaviors

To begin with, two main characteristics can be identified to distinguish the collaboration behavior in different coordination models, namely spatial and temporal coupling:

- spatially coupled coordination models require the involved entities to share a common name space; conversely, spatially uncoupled models enforce anonymous interactions.
- temporally coupled coordination models imply synchronization of the involved entities; conversely, temporally uncoupled coordination models achieve asynchronous interactions.

As a result, four categories of coordination models can be derived:

1. Direct: both spatially and temporally coupled
2. Meeting-oriented: spatially uncoupled and temporally coupled
3. Blackboard-based: spatially coupled and temporally uncoupled
4. Linda-like and Reactive Tuple Spaces: both spatially and temporally uncoupled.

Figure 2.1 summarizes the four categories and associates each with the appropriate agent frameworks.

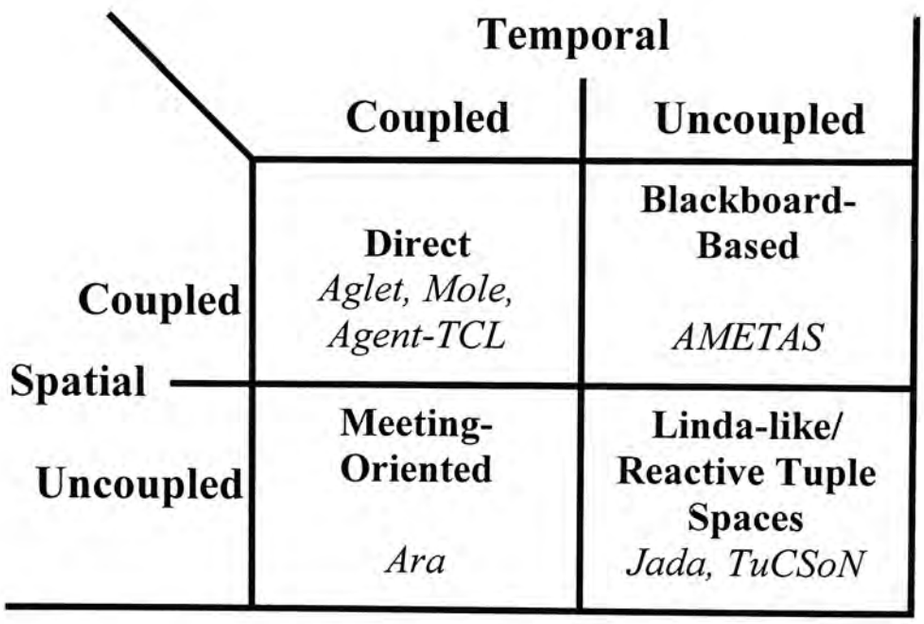

Figure 3.1: Coordination models for mobile agents

3.2 Direct Coordination

In direct coordination models, agents initiate a communication by explicitly naming the involved partners (spatial coupling) and this usually implies synchronization (temporal coupling) among the communicating agents as well. For inter-agent coordination, two agents must agree on a peer-to-peer communication protocol, whereas the coordination between agents and the resources at the hosting environment usually occurs in a client-server manner [Adl95].

Direct coordination is session oriented. The advantage of session is to serve as an explicit communication relationship for building stateful entities. Session-oriented communication can essentially support stateful inter-agent collaboration.

However, direct coordination is generally not suitable for large-scale mobile agent applications as subsequent remote interactions require stable network connections which induce high dependence on network reliability. After all, wide-area communications between mobile entities, whose location may change unpredictably, require complex and highly informed routing protocols instead of rigid session-oriented communication.

Further, as mobile agent applications are intrinsically dynamic, it may be difficult to adopt a spatially coupled model in which the identities of the communication partners must be identified. In some applications, agents cannot know how many other agents compose the application, as agents are created dynamically depending on various environment factors. In addition, when establishing a communication session, agents must be forced to synchronize their activities that, instead, are intrinsically asynchronous and autonomous.

Among the variety of agent applications, direct coordination models can only be exploited effectively for gaining access to local resources where a local server is provided as a manager to interact with agents in a client-server way. Most of the Java-based agent systems like Aglet [OK97], Agent Tcl [KGN+97] and Mole [SBH96] adopt the client-server style communication that is based on message

exchange. In particular, Agent Tcl provides message passing and byte streams at its lowest level whereas higher-level communication mechanisms are implemented at the agent level using message passing or streams.

3.3 Meeting-oriented Coordination

In meeting-oriented coordination, agents can interact with no need of explicitly naming the involved partners. Interactions occur in the context of known meeting points that agents join, either explicitly or implicitly, to communicate and synchronize with each other. An active entity must assume the role of initiator to open a meeting point. Meetings are essentially local and immune to network problems like unpredictable delay and unreliability. A meeting takes place at a given execution environment and only local agents can participate in it.

As agents must share the common knowledge of either the meeting venue or the events that force them in joining a meeting, full spatial uncoupling is not achieved. Although the meeting model partially solves the problem of exactly identifying the involved partners, it has the drawback of enforcing a strict synchronisation between agents. Because in many applications, the schedule and the position of agents cannot be predicted, the risk of missing a meeting is very high.

Meeting-oriented coordination is implemented in Ara [PS97]: one agent can assume the role of meeting server announcing a meeting point at one hosting environment; incoming agents can enter the meeting to coordinate each other. The Ara core provides the so-called service point, which are meeting points with well-known name for agents located at a specific place to interact as clients and servers through exchange of synchronous request and reply messages. Each request is stamped with the name of the client agent and the servers may use that in deciding on the reply.

3.4 Blackboard-based Coordination

In blackboard-based coordination, interactions occur via shared data spaces, local to each hosting environment, used by agents as common repositories to store and retrieve messages. As long as agents must agree on a common message identifier to communicate and exchange data via a blackboard, they are obviously spatially coupled. The most significant advantage of this coordination model derives from the full temporal uncoupling in which messages can be left on blackboards without needing to know, neither where the corresponding receivers are nor when they will read the messages. This clearly suits a mobile scenario in which the position and the schedules of the agents can be neither monitored nor granted easily. Further, in forcing all inter-agent communications to perform via a blackboard, the hosting environments can easily control all interactions, thus leading to a more secure execution environment than that those models mentioned above. With regard to agent-to-host interactions, a blackboard can be exploited to let agents retrieve the needed information without requiring the presence of a specialized resource manager and to let the local environment provide in the blackboard all the data it wants to publish.

AMETAS [ZMG98] implements the blackboard-based coordination models where there is no direct communication between agents and services. Each registered agent and service is assigned a local mailbox and a reference to a driver object. Any request that an agent issues to a service or other agent is sent to its associated driver object first. The driver object, in turn, deposits the request into the mailbox of the intended recipient. Up to this point, the communication procedure is over for the agent and it may even leave the place.

On the other side, the driver object of the requested entity will retrieve the message from its associated mailbox and forward it to the service or agent. In response, the recipient might send back any reply following the same procedure.

Furthermore, group communication can be achieved through address marks that model the creation of group mailboxes.

3.5 Linda-like Coordination

In Linda-like coordination, the accesses to a local blackboard are based on associative mechanisms [CG89] where information is organized in tuples and to be retrieved in an associative way via a pattern-matching mechanism. Associative blackboards, denoted as tuple spaces, enforce full uncoupling in terms of both temporal agreement and mutual knowledge during collaboration.

Associative coordination suits well mobile agent applications. As it is impossible for an agent to learn a complete and up-to-dated knowledge of the hosting environment on the Internet, agents would somehow require pattern-matching mechanisms to adaptively deal with uncertainty, dynamicity and heterogeneity. This associative matching provides a simple means of finding the interested objects in according to their content, without having to know where these objects are stored. This coordination model significantly simplifies agent programming and reduces application complexity

The concept of associative blackboard has been implemented, atop of Java, in the Jada system [CR97] where the so-called ObjectSpace abstraction can be used by mobile agents to store and associatively retrieve object references. Furthermore, agents can create private ObjectSpaces to privately interact without affecting hosting execution environments.

3.6 Reactive Tuple Spaces

In the tuple space coordination model, reactivity stems from embodying computational capacity (i.e. operations or methods) within the tuple space itself, to let it issue specific programmable reactions that can influence the access behavior. The tuple space is no longer a mere tuple repository with a built-in and stateless associative mechanism, as in Linda. Instead, it can also have its own state and react with specific actions to the accesses made by mobile agents. Reactions of a tuple space can be triggered in order to access or modify the content in that tuple space, and even to influence the semantics of the agents' accesses.

Reactivity of the tuple space can provide several advantages. Reactions can be used to implement specific local policies for the interactions between the agents and the hosting execution environment, to achieve better control and to defend the integrity of the environment from malicious agents. In addition, reactions can adapt the semantics of the interactions to the specific characteristics of the hosting environment, thus simplifying the agent programming task much more than the rigid pattern-matching mechanism of Linda.

While several proposals in the coordination area identify the necessity of adding reactivity to the raw Linda model [CG89], a few proposals apply this concept to mobile agents. The TuCSoN model [OZ98] defines programmable logic tuple centres for the coordination of knowledge-oriented mobile agents in which the tuple space defines a Linda-like interface while reactions are programmed as first-order logic tuples. The PageSpace project [CTV+98] defines an enriched Linda-like coordination model for distributed Web applications. The presence of special purpose agents accessing the space and changing its content can provide the capability of influencing the coordination activities of application agents. Reactivity can be integrated also in different coordination models. For example, in the OMG event-based communication model, synchronization objects can embody specific policies to influence the interactions between the agents involved in a meeting.

Chapter 4

Background and Foundations

In this chapter, we introduce the standard facilities that serve as the foundation for the design and implementation of the CoDAC framework. We first describe why we opted for Jini and JavaSpaces as the enabling technologies for implementing CoDAC in section 4.1. Next we give a brief description on the key concepts of both technologies in section 4.2 and 4.3 respectively. At the end of this chapter we introduce the mobile agent platform that serves as the test bed for CoDAC in section 4.3.

4.1 Choice of Technologies

The implementation of CoDAC is greatly facilitated by both Jini and JavaSpaces technologies. For instances, the Jini technology delivers the core functionality to enable network plug-and-play capability which particularly suits our collaboration model. As agents may roam randomly over the network, it is hard to tell where an agent is going in advance. In particular, the hosting environment may not necessary

have installed the needed components (i.e. proxies or stubs) for an agent to access a remote service. Here, Jini provides effective search and downloading of codes in the so-called lookup service. With this lookup service, the agent can obtain and plug in the appropriate proxies anytime to engage in various services regardless of the platform heterogeneity.

The Jini architecture grants high flexibility to the CoDAC framework where the platforms involved can vary from desktop computers to handheld PDAs and even some simple devices like pagers and cellular phones as long as a Java virtual machine is available. Further, Jini breaks the incompatibility between different agent frameworks and enables agents in heterogeneous frameworks to interact.

On the other hand, JavaSpaces technology provides reliable services for storing a group of related objects persistently and retrieving them based on an associative value-matching lookup for specified fields. These mechanisms for storage and retrieval of objects are accessible both locally and remotely. In either case, JavaSpaces implements a transaction model that ensures an operation on a space to be atomic. Transactions are supported for single operation on a single place, as well as multiple operations over one or more spaces which are performed using the two-phase commit model under the default transaction semantics of the Jini transaction.

4.2 Jini Technology

The Jini architecture [Sun99a] provides an infrastructure for defining, advertising and finding services in a network. Services are defined by one or more Java language interfaces or classes. The Jini framework is designed to allow a service on a network to be available to anyone who can reach it, and to do so in a type-safe and robust way. The components of the Jini framework can be segmented into three categories; infrastructure, programming model and services. The infrastructure is the set of components that enables building of a federated Jini system, while the services

are entities within the federation. The programming model comprises interfaces that enable the construction of reliable services.

The Jini framework is built on top of the Java technology and utilizes the homogeneity enforced by the Java virtual machine that standardizes a common execution environment to enable downloaded code to behave the same everywhere. In such a homogeneous platform, the same typing system can be used for local and remote objects as well as the objects passed between them. These objects can be serialized into a transportable form that can later be deserialized. In the serialization, an object can be associated with a codebase that indicates the place or places from which the object's external codes (i.e. some classes that are referenced by the downloaded object, but are not stored in the lookup services) can be downloaded. Hence, such external codes can be downloaded when needed during deserialization. After all, the Java virtual machine protects the host from viruses that could otherwise come with downloaded code. Downloaded code is restricted to operations allowed by the virtual machine's security policy.

In Jini everything is a service. It brings to the network facilities for distributed computing, network based services, seamless expansion, reliable smart devices, and easy administration. It provides lookup services and a network bulletin board (or blackboard) for all services on the network.

4.2.1 The Lookup Service

The Jini lookup services [Sun99b] facilitate a search of services connected by the communication infrastructure and store not only pointers to the service on the network, but also service proxy code and interfaces that enable a user to acquire and execute these services. The lookup service is analogous to the naming or directory service in traditional distributed systems, a place where the clients go to find services. Services are stored in a lookup service by a serialized proxy object.

When a service boots up or initially connects to a network, it typically will find a lookup service using a Jini Discovery protocol [Sun99c] that sends messages to the local networks asking for available lookup services. The service will then register to each discovered lookup service with a serialized instance of the services to be advertised.

When a client needs a service, it first contacts a lookup service. It either discovers the lookup service using a discovery protocol (just like a service do), or talks to one directly using a URL-style identifier. Once the client has a proxy for the lookup service, it asks the lookup service to find one or more services that match a template. Templates define the client's requirement on the service including the types the client wants to use.

The Lookup service uses object oriented type rules to match a search request against all the services currently registered. The client may ask for a single matching proxy object, an array of matching proxy, or an array of service description information for interactive browsing of the lookup service's contents. Finding a usable service results initially in the downloading of the proxy code which can then be used to configure and deploy actual services. The matching service is returned to the client in the form of a serialized proxy object. When the client deserializes the proxy, any necessary code will be downloaded to the client. The location of such code is stored in the serialized proxy object as the service publishes its own code for the client to download.

Then the client invokes methods on the proxy in order to send requests to the associated server. The client is typically unaware of the details of the implementation of the particular proxy. It will invoke the methods on whatever object it gets back. The specific proxy's code will implement the relevant methods as appropriate for the given service.

4.2.2 Proxy

Downloadable service proxies are the key feature that gives Jini the ability to use services and devices without doing any explicit driver or software installation. Jini proxies provide zero-administration way to acquire and use the "glue logic" for communicating with any arbitrary back-end service or device.

In traditional distributed computing systems, an abstract interface definition commonly expressed in an interface definition language such as IDL describes the methods that a remote service understands. This description defines a wire protocol. Once this interface is defined, all servers must be able to receive and execute the method calls. Network protocols are very rigid in the sense that they define exactly and only what they were originally designed to define, and they place strong requirements at the receiving end of the messages.

In the Jini framework, with the introduction of downloadable proxies, defining network services at the API level is made much more flexible. The proxy that implements the abstract interface can be small or large, simple or complex. For instance, there are a number of common practices [Edw99] about how the proxy objects are implemented:

1. The downloaded proxy object performs the service. That is, the object that is sent to the consumers of the service does everything that the service claims to do, by itself. This strategy would be used when the service is implemented purely in software, and there are no external resources that need to be used.
2. The downloaded object is an RMI stub for talking to some remote service. This case is commonly used when there is some centralized RMI-based process somewhere on the network that implements the service. Here, the proxy is simply the automatically generated stub object for the RMI service, which only possesses the necessary ability to speak RMI.
3. The downloaded object is a "smart" proxy [Edw99] that can speak any private communication protocol for talking to the service. This strategy is most

commonly used in two cases. The first is where there is some legacy software system involved. The proxy serves as a wrapper object that interfaces to the legacy service using the system's expected protocols (e.g. sockets, proprietary database languages, etc) and yet provides a pure Java interface that is accessible remotely. The second use for this strategy is when the service is actually provided by some hardware device. In this case, the proxy acts essentially like a downloadable device driver and is implemented to speak whatever proprietary back-end protocols.

This additional layer of client-side code allows the designers of remote services to concentrate on what makes a good programming API for clients rather than what makes a good wire protocol. In a Jini system the wire protocol designs are left to the implementors of each service, and need not be agreed upon among vendors. Only the API must be standardized, and only to the point of common functionality.

4.3 JavaSpaces

The JavaSpaces provides a shared, network-accessible repository for objects utilized for persistent object storage and exchange. The system design of JavaSpaces resembles Linda-like systems described in Chapter 3. JavaSpaces as a Java realization of tuple spaces is similar to Linda systems in that they store collections of information for future computation and are driven by value-based lookup.

Within the space, information is stored in entries as the common currency for all applications. By exchanging entries, objects can communicate, synchronize and coordinate their activities. Entries are objects, in nature, so they may have methods associated with them to implement its behavior and operate as reactive tuple spaces.

An entry can be written into a JavaSpaces service, which creates a copy of that entry in the space that can be used in future lookup operations. Entries that have been

written to a JavaSpaces service can be retrieved using lookup operations with templates. Templates are entry objects that have some or all of its fields set to specified values that must be matched exactly. The remaining fields are left as wildcards (null references) where these fields are not used in the lookup. Given a template *T* as a potential match against an entry *E*, fields with values in *T* must be matched exactly by the values in the same fields of *E*, whereas the wildcards in *T* match any value in the same field of *E*.

The type of *E* must be either of the same type or as a subtype of the type of *T*. In the latter case, all fields added by the subtype are considered to be wildcards. This enables a template to match entries of any of its subtypes.

There are two kinds of lookup operations: read and take. A read request to a space returns either an entry that matches the template on which the read is done, or an indication that no match was found. A take request operates like a read, but if a match is found, the matching entry is removed from the space. Obviously, an entry written to the space can be retrieved at most once using the take operation.

4.4 Grasshopper Architecture

The Grasshopper framework [IKV98] is chosen as the test bed for the CoDAC collaboration model due to its high reliability with full support of the Java 1.2 platform. In this section, we give a brief introduction to the system architecture of Grasshopper. The core components in the systems include Agency, Region and Region Registry as shown in Figure 4.1:

Agency: An agency is the actual runtime environment for mobile and stationary agents. At least one agency must run on each host that shall be able to support the execution of agents. A Grasshopper agency consists of two parts: the core agency and one or more places.

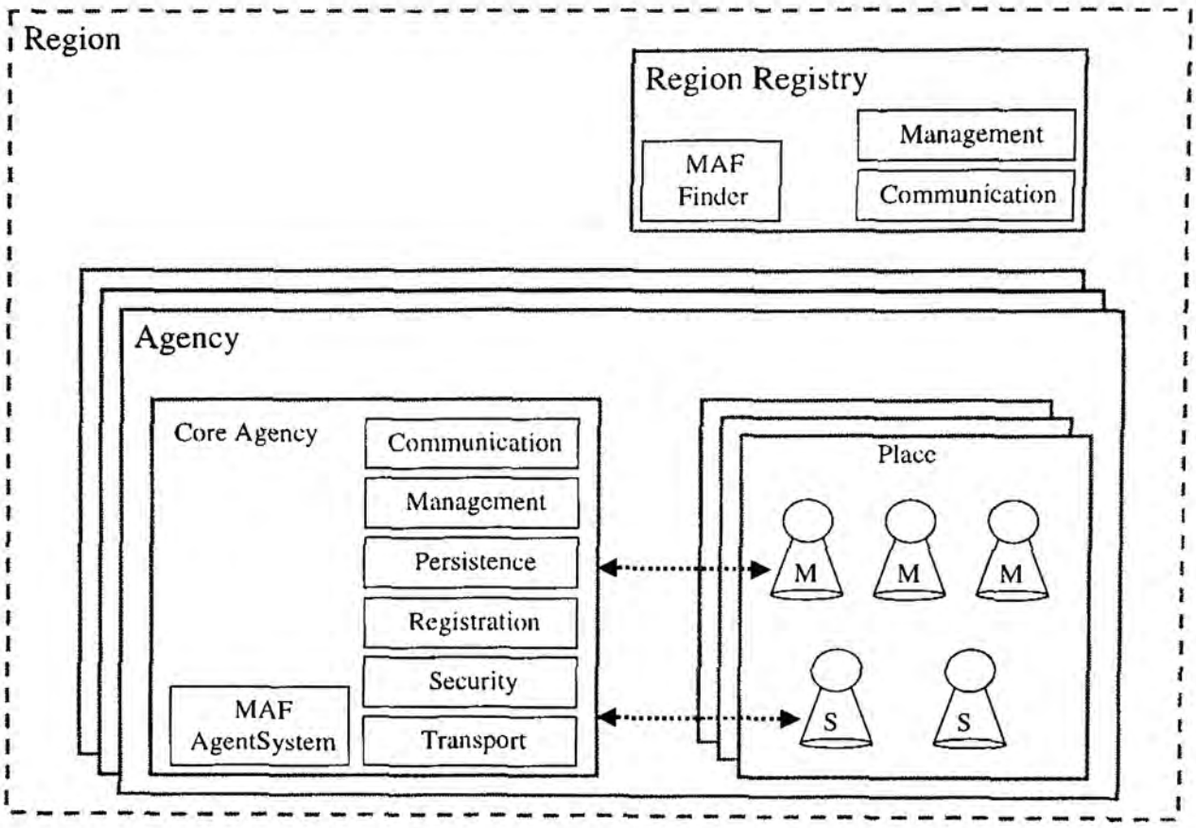

Figure 4.1: Architecture of the Grasshopper framework

Core Agencies represent the minimal functionality required by an agency in order to support the execution of agents. This functionality includes communication, registration, management, security and persistence services.

Places provide logical grouping of functionality inside of an agency. There may exist a communication place offering sophisticated communication features, or there may be a trading place where agents offer or buy information or service access. The name of the place should reflect its purpose.

Region: The region concept facilitates the management of the distributed components, agencies, places, and agents in the Grasshopper environment. Agencies as well as their places can be associated with a specific region by registering them within the accompanying region registry. All agents that are currently hosted by these agencies will also be automatically registered by the region registry. If an agent

moves to another location, the corresponding registry information is automatically updated. A region may comprise all agencies belonging to a specific company or organization.

Region Registry: The region registry maintains information about all components that are associated with a specific region. When a new component is created, it is automatically registered within the corresponding region registry. While agencies and their places are associated with a single region for their entire lifetime, mobile agents are able to move between the agencies of different regions. The current location of mobile agents is updated in the corresponding region registry after each migration. By contacting the region registry, other entities are able to locate agents, places, and agencies residing in a region. Besides, a region registry facilitates the connection establishment between agencies or agents.

Chapter 5

The CoDAC Framework

Throughout an agent's lifecycle, coordination is essential to associate its activities with other entities like various resources or other agents on the execution environments. For instance, an application may be composed of several mobile agents that perform a task collaboratively as they roam across remote sites to access resources and services allocated there.

Although the multiagent paradigm is now on the move, most of the well-known mobile agent frameworks like Aglet [OK97], Grasshopper [IKV98], Ajanta [KT98] and Agent TCL [KGN+97] provide primitive support for inter-agent communication only, while the implementation of any group-based coordination architecture is up to the system developers' responsibility.

For this reason, the general goal of CoDAC is to provide building blocks for collaborative multiagent systems that significantly shorten the development cycle for relevant systems and applications. This componentware delivers the core software components for a multiagent collaboration environment. Above all, CoDAC is adaptive to various well-known and standard agent development frameworks and customizable to meet specific system requirements of an individual application.

Above all, an agent collaboration framework inherits the key requirements from general distributed models in terms of availability and consistency issues. We will describe the key requirements to meet the above properties in section 5.1. Next, we introduce the key components in the CoDAC framework in section 5.2 followed with a description on the system architecture in section 5.3. The communication and the collaboration model will be explained in sections 5.4 and 5.5 respectively.

5.1 Requirements for Enabling Collaboration

The requirements for enabling automated collaboration are primarily addressed in the "process groups" paradigm where the availability and consistency issues focus on consistent group membership, atomic commitment, uniform reliable multicast and fault tolerance. These requirements are described as follows.

5.1.1 Consistent Group Membership

Group membership [SC98] is an agreement among a group of objects that acknowledges a member's involvement and being operational. A group membership protocol establishes an agreement on a valid group membership and serves as the fundamental element for maintaining availability and consistency in distributed applications. The key objective of a group membership protocol is to provide support for dynamic group membership for a wide range of Internet applications and service scenarios. With the provision of dynamic group membership, an individual object is free to join or leave a group dynamically without affecting others in the group.

The membership protocols play important roles for many distributed applications. If consistency is not enforced on the group membership, the availability and integrity of distributed systems cannot be guaranteed. For example, a server being visible to one member but invisible to another in a server group may cause improper denial of

service to the clients even through the requested service is available. To prevent such error, the group membership should be agreed and maintained consistently among the set of operational members regardless of any network failure.

To maintain a consistent group membership, a group membership protocol should enforce both the uniqueness and the validity properties [Rei94]:

Uniqueness: If members p_i and p_j are correct and $V_x(p_i)$ and $V_x(p_j)$ are defined as their x-th version of views respectively, then $V_x(p_i) = V_x(p_j)$

Validity: If p_i is correct and $V_x(p_i)$ is defined, then $p_i \in V_x(p_i)$ and for all correct $p_j \in V_x(p_i)$, $V_x(p_j)$ is eventually defined

Note, a member is said to be correct as long as it behaves rationally and does not intrude the system by manipulating the group membership. A view is a set of data attributes that generally enlist the identities of the members associated with the same group, it represents the owner's perception on the actual group membership. Due to the dynamic property of the group membership, the content of a view must be updated from time to time to reflect any change in the group membership. The constant changes in the group membership generate a sequence of views ordered with the version number.

The uniqueness property implies that all the views sharing the same version identifier are the same at each correct member. The validity property states that each correct member is a member of its own view and the correct members of this view are eventually aware of their membership in the group. Validity and uniqueness imply altogether that those correct p_i at any $V_x(p_i)$ are exactly the set of correct members that intuitively form a group and mutually believe one another to be members.

5.1.2 Atomic Commitment

In distributed systems, partial failures can occur in a way that some system entities may be working while others have failed. For this reason, transactional behaviors are essentially important in distributed computing, as they provide a means for enforcing consistency over a set of operations on one or more remote participants. Transactions [Sun99e] are a fundamental tool for many kinds of computing. A transaction allows a set of operations to be grouped as a whole such that they either all succeed or all fail. The operations in the set appear from outside the transaction to occur simultaneously.

In transaction processing, the algorithm that ensures consistent termination is called an atomic commitment protocol ACP [BH87]. The ACPs are designed to ensure a single logical action (either Commit or Abort) is consistently designed and carried out by all parties involved in a distributed transaction as the following conditions are enforced:

1. All participants that reach a decision reach the same one
2. A participant cannot reverse its decision after it has reached one
3. The commit decision can only be reached if all participants voted for it
4. If there are no failures and all participants voted for commitment, then the decision will be to commit
5. If all existing failures are repaired and no new failures occur for a sufficiently long time, then all participants will eventually reach a decision

Condition 1 ensures that the transaction terminates consistently. Condition 2 states that the termination of a transaction at a participant is an irrevocable decision. Condition 3 implies that a transaction cannot commit unless all participants agree to do so, whereas condition 4 is a weak version of the converse of condition 3. It is not required in condition 1 that all participants have to reach a decision as one may fail and never recover, but condition 5 does require that all participants be able to reach a decision once failures are repaired.

5.1.3 Uniform Reliable Multicast

A distributed application is usually composed of different parties communicating through message passing. Point to point appears to be the most simple communication pattern, but in most cases, group communication or multicasts are often more desirable. Group communication raises two issues, namely the reliability and the ordering issue. For instance, uniform reliable multicast [SS93] is concerned with atomicity as well as the total ordering of the multicasts. A multicast *m* to a group *g* is uniform reliable iff the following condition holds:

- If a member in group *g* has received *m*, then all non-faulty members of *g* eventually receive *m*.

In general, uniform reliable multicast has the property that if *m* has been received by any member of a group, then *m* is received by all members that reach a decision. On top of that all *m*'s are received in total order.

5.1.4 Fault Tolerance

Distributed systems are vulnerable to network failures, these failures can be generalized as either site or communication failures. The former occurs when a site experiences a system failure where processing stops abruptly and the contents of volatile storage are destroyed. In particular, as each site is either functioning or has failed, different sites may be in different states as a result of partial failure. On the other hand, the latter may occur for various reasons. First, a message may be corrupted due to noise in a link. Second, a link may malfunction temporarily, causing a message to be completely lost; or third, a link may be broken for a while, causing all messages sent through it to be lost. Further, a combination of site and communication failures may cause network partition, when the operational sites are

divided into two or more components, where every two sites within a component can communicate with each other, but those in different components cannot.

Fault tolerance concerns with the above issues in enhancing the availability of distributed systems. Common practice employs replicated components for as replacement for the failed parts but this often arise various coordination problems.

5.2 System Components

CoDAC can be viewed as a Jini technology-enabled service delivered by a set of distributed objects, which make use of the Jini technology infrastructure [Sun99a] to discover and interact with each other. This collection of service objects serve as a flexible and reliable backbone that supports both local and global collaboration atop a hierarchical structure. Such hierarchical group structure decentralizes the coordination effort with backup support for fault recovery while enforcing consistency and atomicity.

All participating agents are organized into collaboration groups in which they exchange information and collaborate to take consistent actions. Agents may migrate from place to place and are free to join or leave a group at will. Each agent is associated with a priority, which defines the total ordering among the agents and reflects the sequence they register with the group. Without loss of generality, the agent associated with the highest priority is assigned as the coordinator to manage the group.

This collaboration backbone is composed of two key components, namely the DistributedAgentAdapter and the CollaborationCore.

5.2.1 Distributed Agent Adapter

Distributed Agent Adapter (DA adapter) is a software component [Szy97] implemented as a Jini service object that performs the foundational functions required by a collaboration framework, namely the enforcement of consistent group membership, atomic commitment protocol, uniform reliable multicast and fault recovery. As a software component, the DA adapter encapsulates the above functionality and is deployable at runtime. For instance, DA adapters rely on the Jini framework to maintain the transparency of locations. They are registered with and stored in the Jini lookup services such that each agent can download an instance of them from a lookup service at runtime in order to join a collaboration group. After deserialization, the DA adapter will perform the necessary collaboration routine on behalf of the associated agent as long as the latter stays in the group and remains functional. The agent will need to suspend the DA adapter upon migration to the next spot and resume the adapter when it has settled again to continue its collaboration work.

The DA adapter serves as a smart proxy [Edw99] to interface its associated agent with the other collaborating parties as shown in Figure 5.1. It can be utilized as a gateway for reliable communication with agents in the same local group or in adjacent groups. The communication channels involved are connected on top of the JavaSpaces technology [Sun99d]. The underlying mechanism is transparent to the collaborating agents, as the DA adapters read and write messages into the space, interpret the received messages autonomously and only notify the associated agents to take corresponding actions when necessary. This will be explained in more details in Chapter 8.

Each individual DA adapter exercises the self-managing property of software components to take responsibility for their own resources (the associated agents), work across networks and interact with one another to constitute a reliable communication backbone for the collaborating agents as a whole. The infrastructure of the backbone is in the form of a hierarchy that spans down from the global

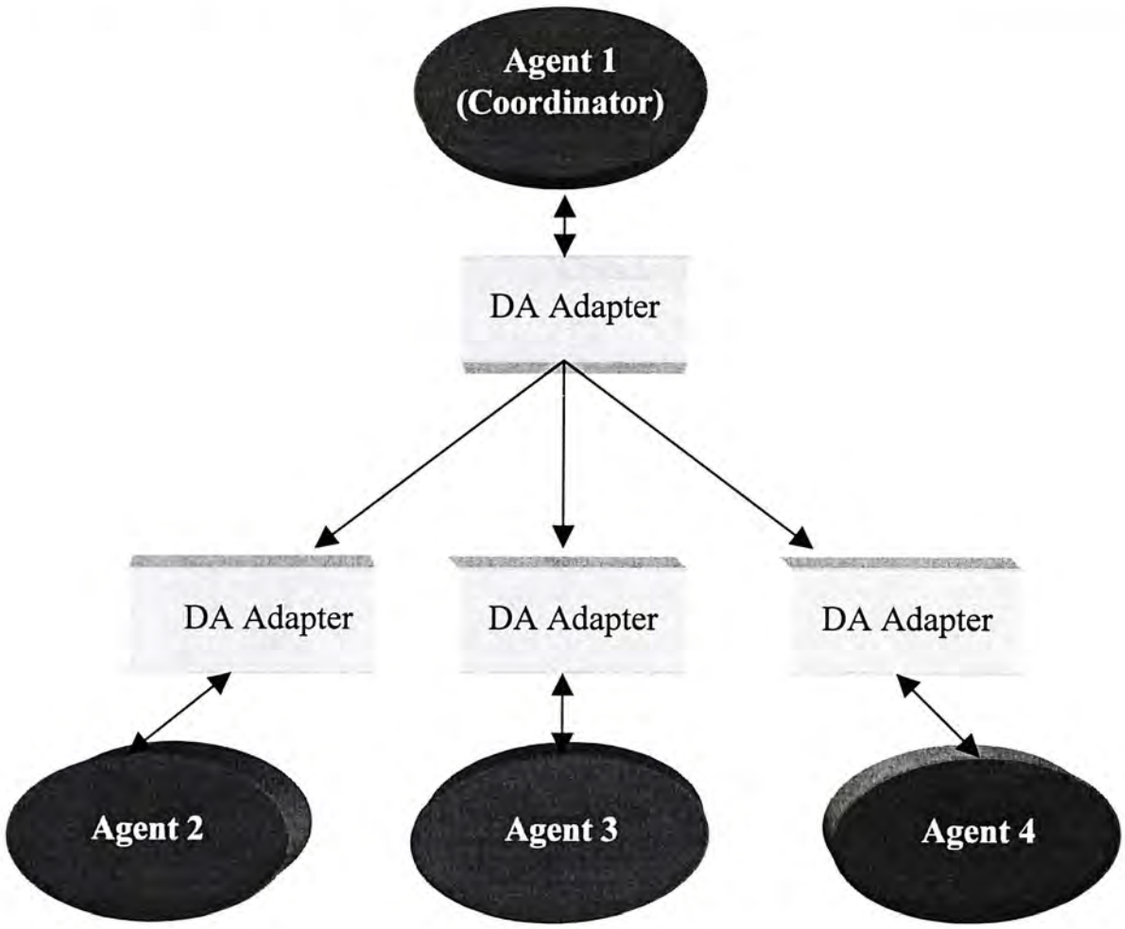

Figure 5.1: Agent collaboration group

coordinator to subsequent local group coordinators and terminates at the leaf level where the collaboration members reside as shown in Figure 5.2.

The root and the consecutive levels of each subtree in the hierarchy correspond to a local collaboration group headed by a local coordinator at the local root. In this sense, each intermediate node in the hierarchy corresponds to a local coordinator with dual identities as both a coordinator at its local domain and a collaboration member with respect to the ancestral collaboration group. On one hand, the local coordinator coordinates each individual subordinate's work into local collaboration. On the other hand, it collaborates with the peers in the ancestral group to pursue the global goals. A DA Adapter generally contains a Distributed Agent Manager and conditionally embeds a Collaboration Manager, whose functionality will be described in the next section.

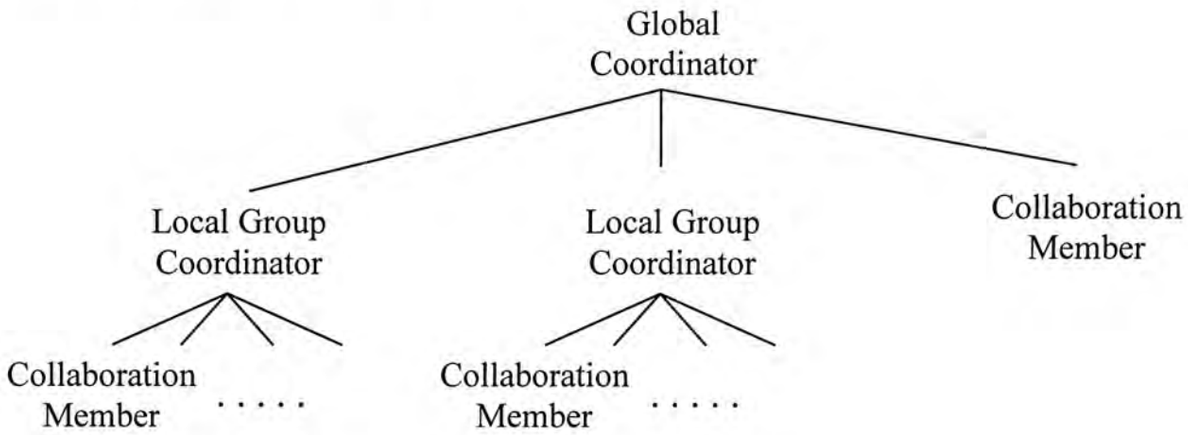

Figure 5.2: The agent group hierarchy

5.2.2 CollaborationCore

The CollaborationCore serves as the super-class for the implementation of the knowledge-reasoning logic of the group coordinator. As it is impossible to generalize all possible analytical mechanisms into a single class to handle collaboration of all kinds, the CollaborationCore is simply an abstract interface that entails the underlying interaction with the DA adapter such that any of its subclasses could integrate seamlessly into the collaboration framework. It is up to the responsibility of the system developers to implement the desired analytical approaches to meet their specific requirements.

Object instances of the CollaborationCore subclasses (denoted as kernels for simplicity) are user-defined Jini service objects that performs the specific analytical works required to compute the collaboration results. Therefore, every group coordinator must obtain a kernel from the lookup service and plug it into the associated DA adapter before it possesses the capability to lead a group.

5.3 System Infrastructure

The CoDAC collaboration framework enforces the consistency of the decisions reached among the participating agents atop the distributed transaction semantic. For instance, the CoDAC implementation conforms to the standard X/Open Distributed Transaction Processing DTP model [Xop95] to facilitate resource sharing among multiple distributed agents and coordinate their work into global collaboration. The system infrastructure is illustrated in Figure 5.3 (For simplicity, only the DA Adapter of the coordinator is shown in detail). There are altogether four entities participating in the framework, namely the agent, the distributed agent manager, the collaboration manager and the kernel.

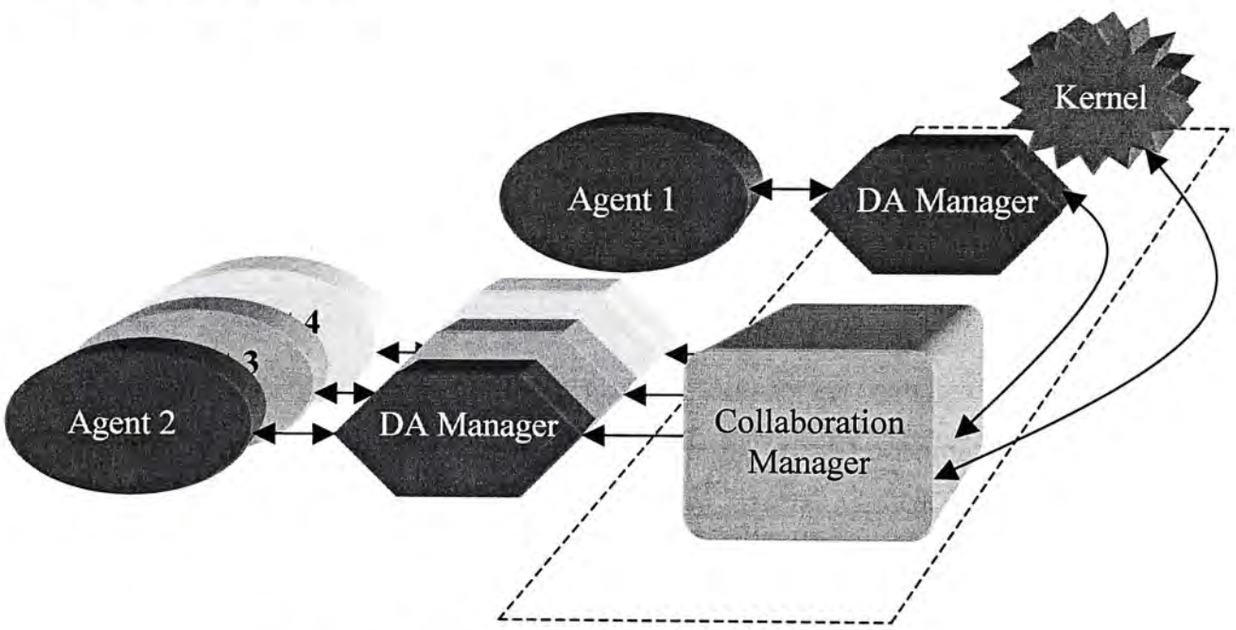

Figure 5.3: The distributed transaction infrastructure

5.3.1 Agent

Agents are analogous to the transactional resources [Xop95] in the DTP model. They are recoverable objects [LLK+97] containing the actual state to be changed by a

transaction. The state to be updated in a transaction can be the internal state of the agent or some shared network resources (e.g. databases, file systems, service agents, etc) referenced by the agent. In any case, the agent should possess the ability to recover to a consistent state in the presence of failures.

5.3.2 Distributed Agent Manager

The Distributed Agent Manager (DA manager) is analogous to the resource manager [Xop95] in the DTP model. Every DA adapter has an instance of DA manager. DA managers structure the changes to the state and the resources of the agents they manage as recoverable and atomic transactions. They constitute the collaboration context and let the collaboration manager to coordinate completion of the transactions entailed in the collaboration atomically. The DA manager, once opened is kept open until the associated agent migrates or terminates.

5.3.3 Collaboration Manager

The Collaboration Manager is analogous to the transaction manager [Xop95] in the DTP model. Only the DA adapter associated with the group coordinator instantiates a collaboration manager. The collaboration manager manages collaboration and coordinates the decision to start, commit or rollback. This ensures the collaboration to terminate consistently. Further, the collaboration manager also coordinates the recovery activities of the collaboration group when necessary, such as graceful replacement of any failed coordinator.

5.3.4 Kernel

The Kernel is analogous to the application program [Xop95] in the DTP model. It is a self-contained object associated with the group coordinator. It implements the

desired logic to analyze for structured collaboration results. The kernel defines the start and end of collaboration and specifies a sequence of consistent actions based on the resources within the collaboration context.

5.4 Collaboration

Collaboration in CoDAC is a complete unit of work that may comprise many computational tasks performed by individual agents such as user interaction, data retrieval and communication. Typical agent collaboration modifies the state or the associated resources of the collaborating agents.

Collaboration implements the transaction semantics and is able to be rolled back. An agent may roll back the collaboration in response to an event such as the failure of system components. Every collaborating agent subjected to transaction control must be able to undo its work in a collaboration at any time that it is rolled back.

Each agent is associated with a DA manager that serves as a proxy to interface with the collaboration manager. The DA manager allows the collaboration manager to start and end the collaboration associated with the participating agents and to coordinate the collaboration completion process. At collaboration termination, the DA managers are informed by the collaboration manager to prepare to commit or rollback the collaboration atop an atomic commitment protocol. When the coordinator determines that the collaboration can complete without failure of any kind, it commits the collaboration. This means that all collaborating agents deliver the same collaboration result and that changes to internal state and external resources take permanent effect. Either commitment or rollback results in a consistent state as completion means either commitment or rollback.

5.4.1 Global Collaboration

Every DA manager in the collaboration context must implement the transactional semantics. Many DA managers may operate in collaboration for the same unit of work. For example, the root coordinator might request update to several different databases referenced by agents in separate local groups. This unit of work is a global collaboration that occurs inside a transaction (i.e. the collaboration transaction) where work occurring anywhere in the group must be committed atomically. Each DA manager must let the collaboration manager coordinate its recoverable units of work that are part of the global collaboration.

Commitment of an agent's private work depends not only on whether its own operations can succeed, but also on operations occurring at other agents remotely. If any operation fails anywhere, every participating DA manager and the associated agent must roll back all operations they did on behalf of the collaboration manager. A given DA manager is typically unaware of the work that other DA managers are doing. A collaboration manager informs each DA manager of the existence, and directs the completion, of global collaborations. A DA manager is responsible for mapping the underlying recoverable units of work to the global collaboration.

5.4.2 Local Collaboration

A global collaboration may involve one or more local collaborations. A local collaboration, refers to the collaboration among the peer members in a collaboration group, is a part of the work in support of a global collaboration for which the collaboration manager and the DA managers engage in an interleaved but coordinated transaction commitment protocol. Each of the DA manager's internal units of work in support of a global collaboration is part of exactly one work.

A global transaction might involve inter-group collaboration. For example, the root coordinator requests its subordinates to prepare commitment to some collaboration results. Among which, any local coordinator within the root

collaboration group may, in turn, initiate a local collaboration nested within this global collaboration. Each local group engages into a local transaction wherein the local coordinator coordinates to delivery of the global collaboration results on behalf of the root coordinator. Every local collaborator gives its vote to the root coordinator as long as the local work group has reached a mutual agreement to prepare commit or abort the transaction. The root coordinator gathers all votes from its subordinates and coordinates their work to the final decision, whether to commit or abort globally. Each local coordinator, thereby, terminates the local coordination in accordance with the global decision to enforce global consistency.

Chapter 6

Collaboration Life Cycle

The life cycle of a collaboration consists of three phases, namely initialization, results gathering and results delivery. A collaboration process, once initialized, begins as the coordinator requests computation results from each individual agent and terminates after each participating agent installs or aborts the finalized collaboration results, as shown in Figure 6.1. We will describe each phase in that order. A single collaboration may not necessary get the job done, in this case, subsequent collaborations can take place before the ultimate goal is attained.

6.1 Initialization

At the very beginning, the coordinator agent *c* starts a collaboration group by instantiating a DA adapter with a unique group ID. This instance of DA adapter, in turn, discovers all available lookup services on the network for advertising the group. The DA adapter makes the collaboration group public through registering a serializable instance of its clone as a service proxy on each lookup service it has

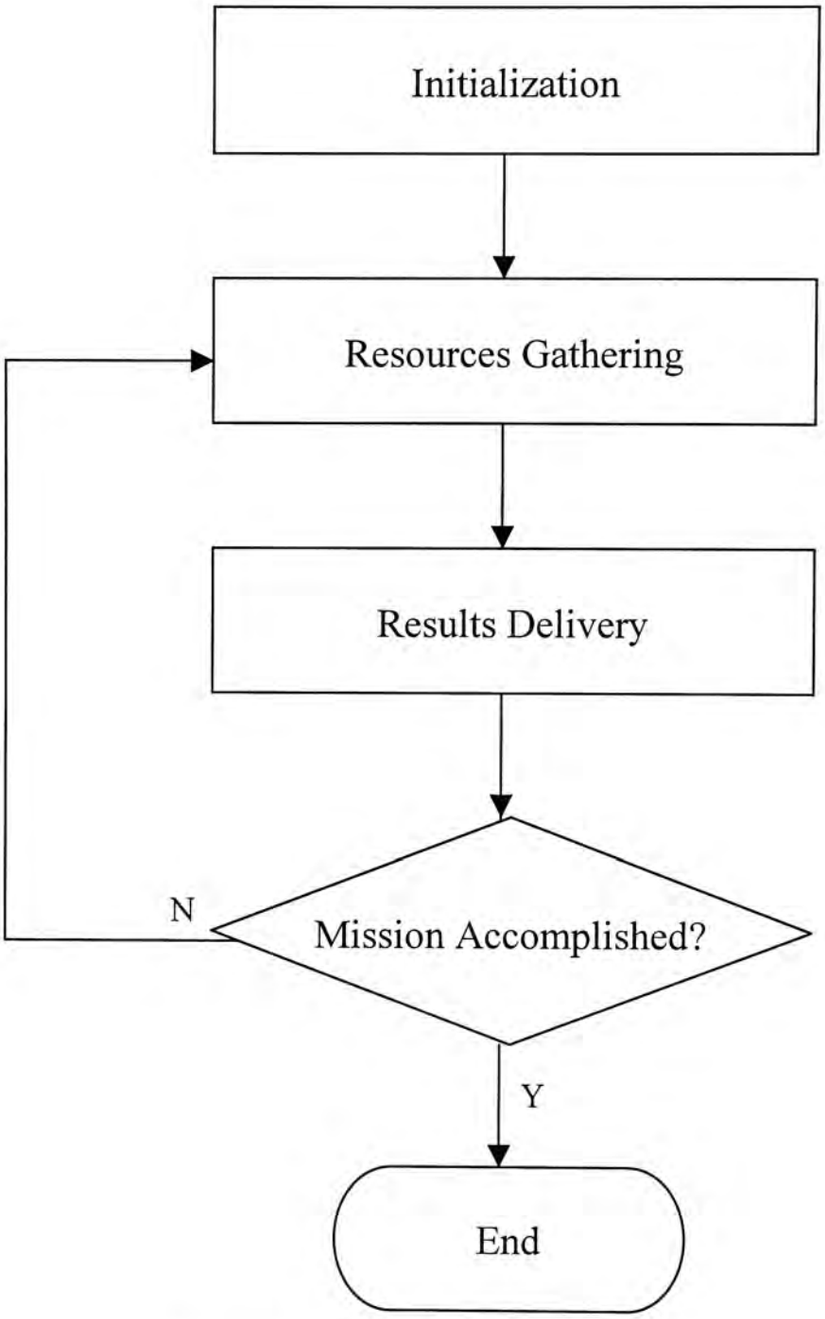

Figure 6.1: The collaboration life cycle

discovered. Each registered proxy shares the same service ID [Sun99a]. For each agent *p* that intends to participate in a collaboration group, it gains access to one or more lookup services around as ordinary Jini service clients do. Next, *p* searches for the desired service proxy, a serialized instance of DA adapter in this case, through

the lookup service. The search criteria can be based on the group ID, the Jini service ID [Sun99b] or even the ID of the coordinator. As long as the desired collaboration service is located, the relevant DA adapter will be downloaded to *p*. After being deserialized, the DA adapter contacts the original DA adapter (the one associated with *c*) and issues a request to join the collaboration group on behalf of *p*. In response, the DA adapter of *c* verifies the request, checks for data consistency and grants the membership for *p* under mutual agreement with all available members within the group. Such mutual agreement is enforced by the group membership protocol to be detailed in Chapter 7. If the request is granted after all, p becomes part of this group and is ready to collaborate. The procedure described above is summarized in Figure 6.2.

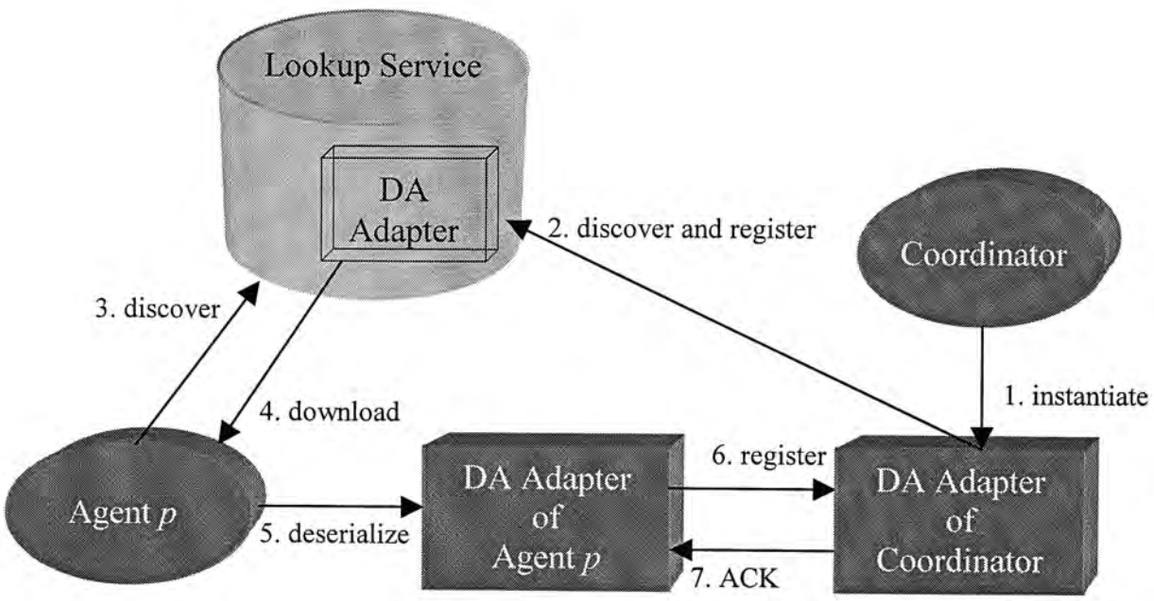

Figure 6.2: Collaboration initialization phase

6.2 Resouces Gathering

The collaboration begins with the assembling of available resources within the collaboration context. To begin collaboration, the DA adapter of *c* instantiates a collaboration manager which maps the collaboration into a transaction. Upon initiation, the collaboration manager issues a collaboration request to each DA adapter within the collaboration context. This request signals each participating agent to deliver its individual computation results to *c*. As a DA manager receives the collaboration request, it notifies its associated agent immediately by firing a GatherResourcesEvent. In response, the agent presents the relevant data to the DA manager as soon as the data is available and the DA manager simply forwards the data to the collaboration manager. The resources gathering phase terminates after all the participating agents have contributed their computation results or when the collaboration manager times out. Either case, all the gathered information will be delivered to the kernel for analysis. If *c* is not equipped with a kernel yet, it must download one from the lookup service before it proceeds. The above procedure is illustrated in Figure 6.3:

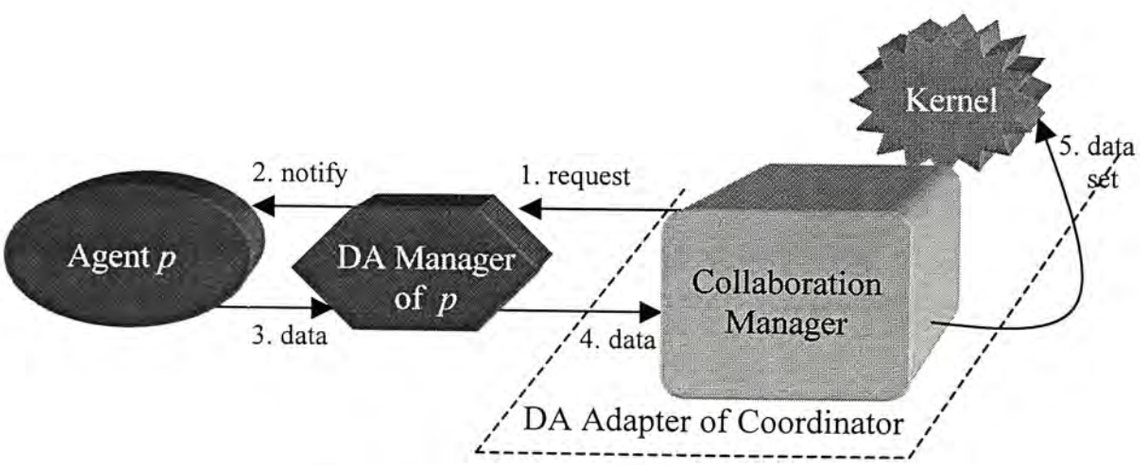

Figure 6.3: Resources gathering phase

6.3 Results Delivery

After the kernel has completed evaluating the collection of data assembled, it deduces some kind of collaboration results (e.g. identifying the optimal offer from a bunch of merchants), and the collaboration may end. At the end of collaboration, the kernel returns the collaboration results to the collaboration manager which, in turn, forwards the collaboration results to each DA Manager within the collaboration context inside a transaction. The underlying atomic commitment protocol will be described in Chapter 7. Anyway, all collaborating agents will install the collaboration results consistently as long as the transaction commits. The collaboration manager terminates whereas the collaboration group persists. Each participating agent either completes the missions stated in the collaboration results (e.g. to commit/abort its operation on a database) or adapts its behavior accordingly (e.g. to revise its goals and objectives) while the coordinator may initiate subsequent collaboration as needed. The above procedure is summarized in Figure 6.4:

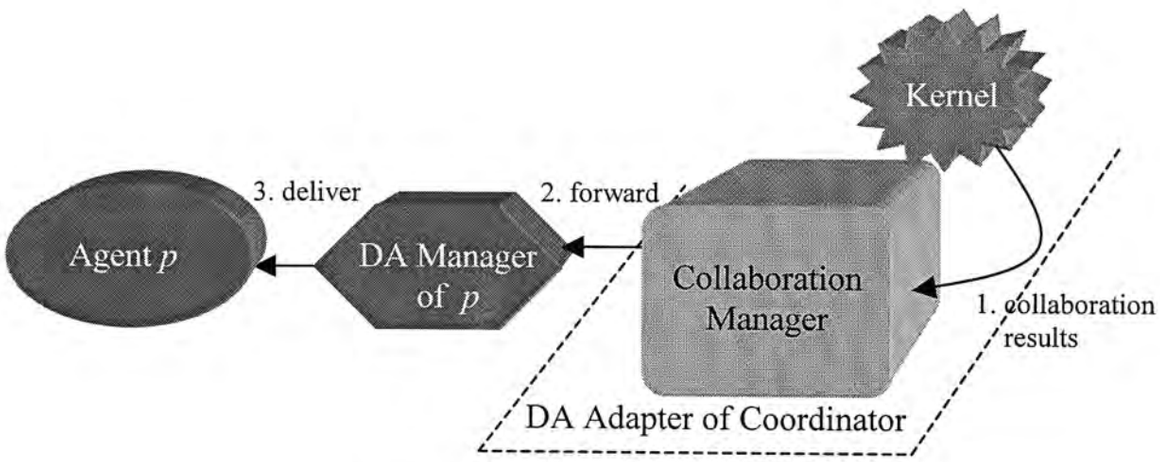

Figure 6.4: Results delivery phase

Chapter 7

Protocol Suite

In this section we describe the protocol suite designed in CoDAC to enforce the requirements identified in section 5.1. Since by the impossibility proof of distributed consensus in asynchronous environment with partial failure [FLP85], where at least one collaborating agent may fail, it is impossible to distinguish a crashed agent from an agent connected through a slow channel. In order to preserve the atomicity of each collaboration, we have to sacrifice the full temporal uncoupling property. Hence these protocols involved are designed as timed asynchronous where a timeout delay ∂ is defined to trigger fault detection of failure.

In section 7.1 we first look into the group membership protocol that enforces consistent group membership with fault recovery capability. Next, we describe the underlying commitment protocol to ensure atomic commitment of a collaboration cycle in section 7.2. In section 7.3, we will examine the uniform reliable multicast protocol.

7.1 The Group Membership Protocol

The group membership protocol is used for the management of the set of participating agents among which each agent can alternatively perform the coordinator role. In this sense, the participating agents serve as replicated object for enhancing high availability for the coordination service.

7.1.1 Join Protocol

When an agent c starts a new group, it initializes its local view of the current group membership: $V_0(c)$ to $\{c\}$ and serves as the default coordinator. Each group is associated with an ID, say g, for instance. The coordinator synchronizes the registration of each agent for joining the group. If an agent p wishes to join the group g, it would bring about a view change to reflect the new group membership. Such changes in the view will be delivered to each participating agent atop the group membership protocol:

1. q first sends a *join_req* predicate to the coordinator c.
2. In response, c verifies the request from q (e.g. checks for any duplicated agent ID, makes sure if q is reaching the right group, etc). If q passed the verification, then c generates the next version of view, $V_{n+1}(c)$, where

   $V_{n+1}(c) = V_n(c) \cup \{q\}$

   Suppose $\exists$ agent p in g such that $V_n(c) = \{c, p\}$, then $V_{n+1}(c) = \{c, p, q\}$ and q is assigned with the lowest priority among g.
3. Next, c will send a *new_view* predicate along with $V_{n+1}(c)$ to each agent in $V_{n+1}(c)$ inside a transaction.
4. Each recipient votes either yes or no to indicate its readiness to install $V_{n+1}(c)$.

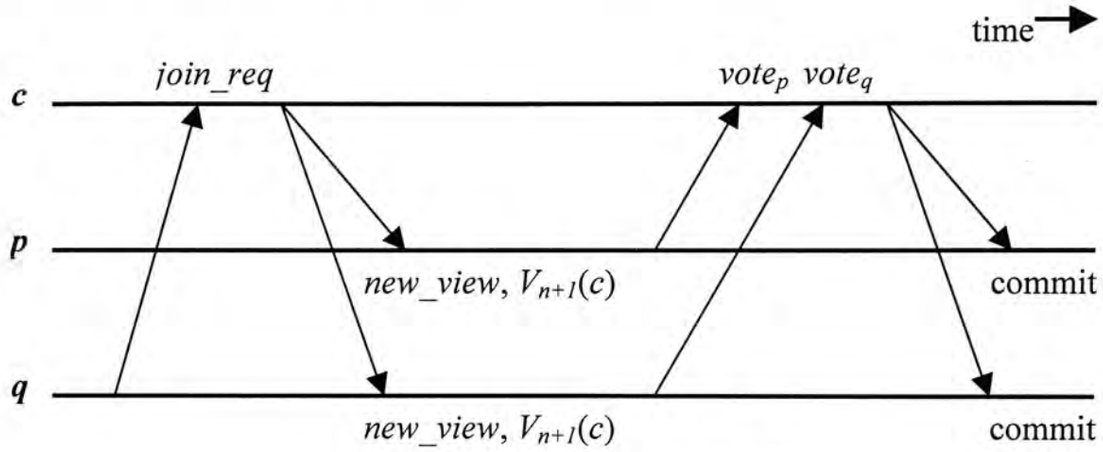

Figure 7.1: The join protocol

5. c collects all the votes to make the decision whether to commit or abort the transaction. If every involved agents votes yes then the decision is to commit. Otherwise the coordinator repeats step 3 until the transaction can commit.
6. As the transaction commits, every agent receives the *new_view* predicate and installs $V_{n+1}(c)$ as the current view wherein q becomes a new member of g.

The join protocol is summarized in Figure 7.1. For simplicity, we only show the cross-agents messages involved in this protocol (the same holds for all figures in this chapter). Obviously, when we say a predicate P is sent to all agents in $V_x(c)$ it implies P is delivered to c and other group members altogether (i.e. by the validity of $V_x(c)$), but internal messages exchanged among different components within c is not shown in the figure.

7.1.2 Leave Protocol

Similarly, if p wishes to leave the group g, it would cause a view change as well. The protocol for an agent to unregister with the current collaboration group is described as follows:

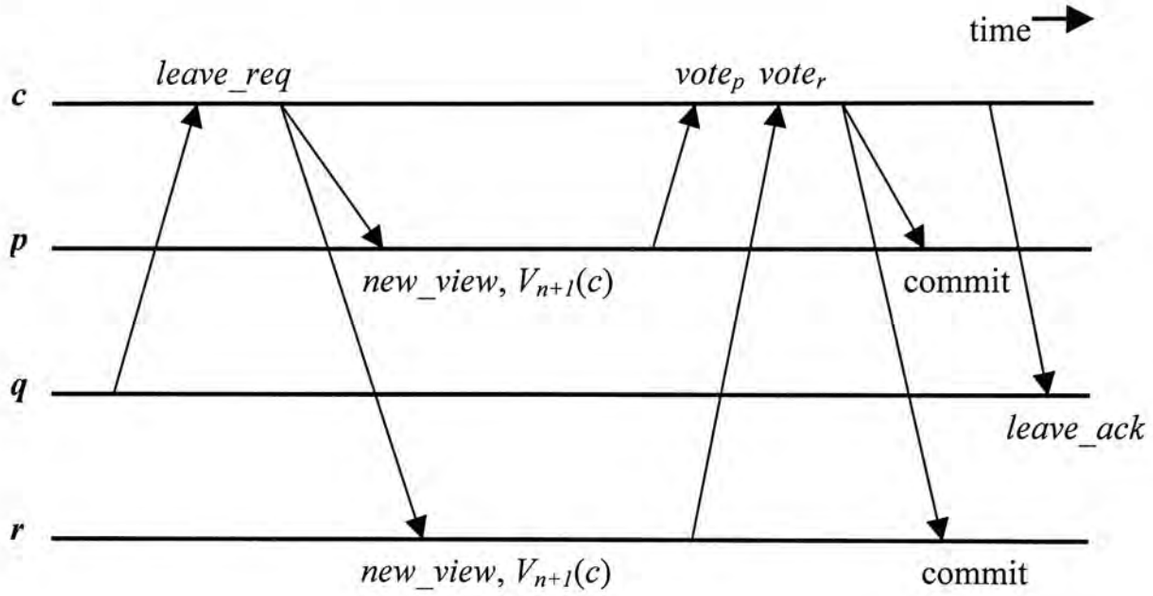

Figure 7.2: The leave protocol

1. q first sends a *leave_req* predicate to c.
2. In response, c verifies the request from q (e.g. checks for any invalid or unknown agent ID, etc). If p passed the verification, then c generates the next version of view, $V_{n+1}(c)$, where

   $V_{n+1}(c) = V_n(c) \otimes \{q\}$

   Suppose $\exists$ agents p and r in g such that $V_n(c) = \{c, p, q, r\}$,

   then $V_{n+1}(c) = \{c, p, r\}$
3. Next, c will sends a *new_view* predicate along with $V_{n+1}(c)$ to each agent in $V_{n+1}(c)$. This is delivered inside a transaction in a similar way to the join protocol.
4. Each recipient votes to indicate its readiness to install $V_{n+1}(c)$.
5. c collects all the votes involved and decide to commit as long as every agent in $V_{n+1}(c)$ votes yes. Otherwise, c repeats step 3 until the transaction can commit.
6. As the transaction commits, every agent in $V_{n+1}(c)$ receives the *new_view* predicate and installs $V_{n+1}(c)$ as the current view, after which, c returns a *leave_ack* predicate to q to grant the unregistration.

The protocol described above is summarized in Figure 7.2.

7.1.3 Recovery Protocol

The recovery protocol is designed for the replacement of the coordinator whose failure is detected by a collaboration member. The underlying mechanisms for failure detection will be described in Chapter 8. For the time being, we look into the recovery protocol first.

There are altogether two protocols for failure recovery. These two protocols are to be applied depending on the subjects to be recovered, namely the ordinary members and the coordinator. An ordinary member can be any agent but the coordinator within the group. Suppose *c* detects the failure of *q* and has to update the view to reflect such failure. This would initiate the recovery protocol presented in Figure 7.3. This protocol proceeds similar to the leave protocol described in the last section except that c assumes a *leave_req* has been issued from *q* implicitly and leaves out the *leave_ack* for *q* at the end.

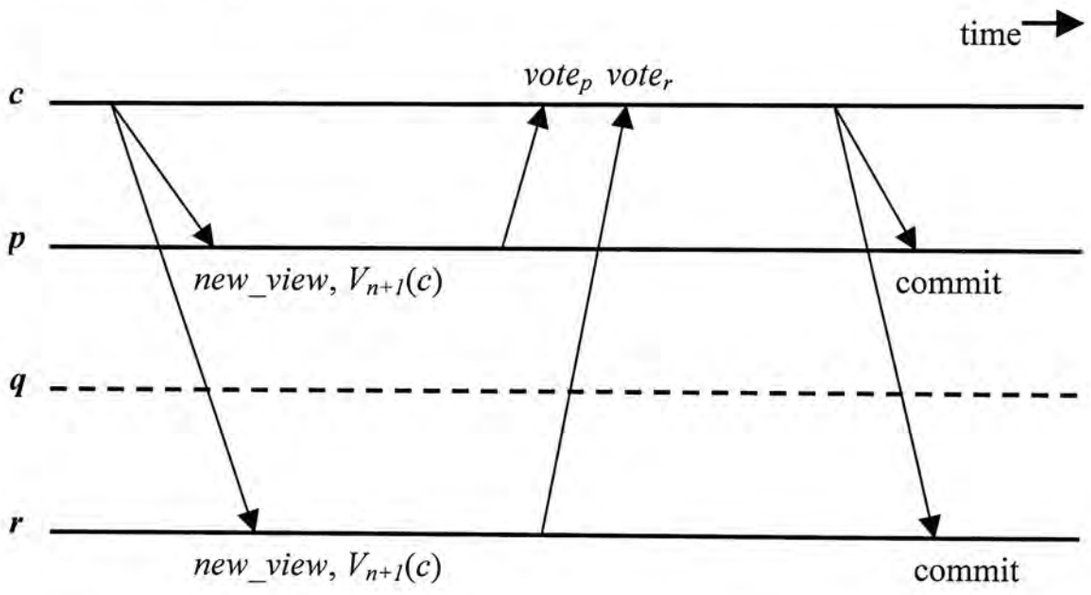

Figure 7.3: The recovery protocol for ordinary members

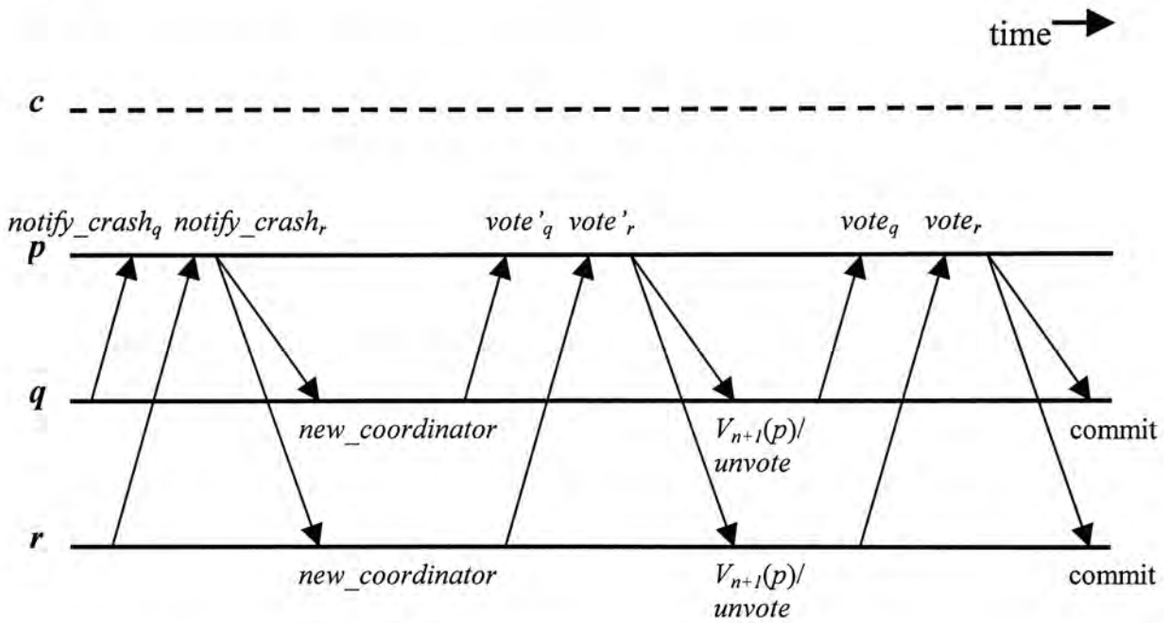

Figure 7.4: The recovery protocol for the coordinator

On the other hand, if the subject to recover is the coordinator *c*, then the protocol for recovering such failure would initiate an election to appoint exactly one of the ordinary members to replace the failed coordinator. Suppose the failure of *c* is being detected through the mechanisms described in Chapter 8 then the recovery protocol will be initiated and proceed as follows:

1. Each agent who has detected the failure of *c* first sends a *notify_crash* predicate into the channel of its predecessors (i.e. members having higher priorities than itself) in view(q) except *c*, one by one every ∂ time units (i.e. a time out delay) in descending priority, and stops when it gets a *new_coordinator* predicate from any of its predecessors.
2. If an agent gets any *new_coordinator* predicate from its predecessor before it times out, just let the predecessor handles the recovery. Otherwise proceed to 3.
3. An agent with no detectable predecessor, say *p*, computes a new group membership in g, $V_{n+1}(p)$ by removing *c* and any failed predecessor (detected in step 2) from $V_n(p)$. Then *p* sends a *new_coordinator* predicate to each agent in $V_{n+1}(p)$.

4. Upon receiving a *new_coordinator* predicate, each agent in $V_{n+1}(p)$ replies with either a yes or no vote in terms of its willingness to appoint p as the new coordinator.
5. As long as the yes votes gathered by p constitute a majority of the original view $V_n(p)$, it is guaranteed that no other competitor can get a majority vote. p can thus coordinate all recipients to install $V_{n+1}(p)$ and become the new coordinator for g mutual exclusively. Otherwise, p returns an *unvote* predicate to all agents that have voted yes such that they can vote for another coordinator.
6. In either case, the $V_{n+1}(p)$ or the *unvote* predicate will be delivered inside a transaction to enforce consistency. The recipients vote in terms of their readiness to install $V_{n+1}(p)$ or to deliver the *unvote* message. The transaction commits as long as every recipient votes to commit.

Figure 7.4 summarizes the above protocol.

7.1.4 Proof

In this section, we prove the uniqueness and validity defined in 5.1.1.

Uniqueness: The proof for uniqueness is trivial. For all three protocols involved in group membership management, each new version of view $V_x(c)$ is defined by the coordinator c (either the default coordinator or a newly elected coordinator) who, in turn, coordinates all members in $V_x(c)$ to install $V_x(c)$ inside a transaction. Such that:

$$\forall\, p \in V_x(c),\ V_x(p) = V_x(c)$$

Therefore, the uniqueness property holds.

Validity: Suppose p is correct and $V_x(p)$ is defined

we assume $p \notin V_x(p)$

Since $p \in V_x(c)$ (p is correct)

and $V_x(c) = V_x(p)$ (proven in uniqueness)

$\Rightarrow$ $p \in V_x(p)$ which contradicts to the assumption that $p \notin V_x(p)$

Hence, the validity property holds as proven by contradiction.

Further, by the atomic property of the group membership protocols, it is trivial to prove that

$\forall\, q \in V_x(p)$, $V_x(q)$ is eventually defined and $V_x(q) = V_x(p) = V_x(c)$

7.2 Atomic Commitment Protocol

During the results delivery phase described in section 5.3, the collaboration results embedded in message could be simply delivered inside a transaction consistently. In this way, however, we do not know whether each participating agent agrees with the final decision (i.e. the collaboration results) in the first place. In order to let those participating agents to have a voice in the final decision, we have designed the atomic commitment protocol as a five round protocol [SC98].

After the coordinator c has finished computing the collaboration results R, it then requires to coordinate all agents in $V_n(c)$ to deliver R consistently in order to terminate the collaboration transaction. The protocol involved is as follows:

1. c sends a *deliver_req* predicate enclosed with R to every agent within the collaboration context (i.e. $V_n(c)$).
2. In response, each agent checks its own state to see if it can commit to R. Every agent then returns the appropriate *vote'* (either yes or no) to the coordinator to indicate its willingness to deliver R.
3. c collects all the votes among the group

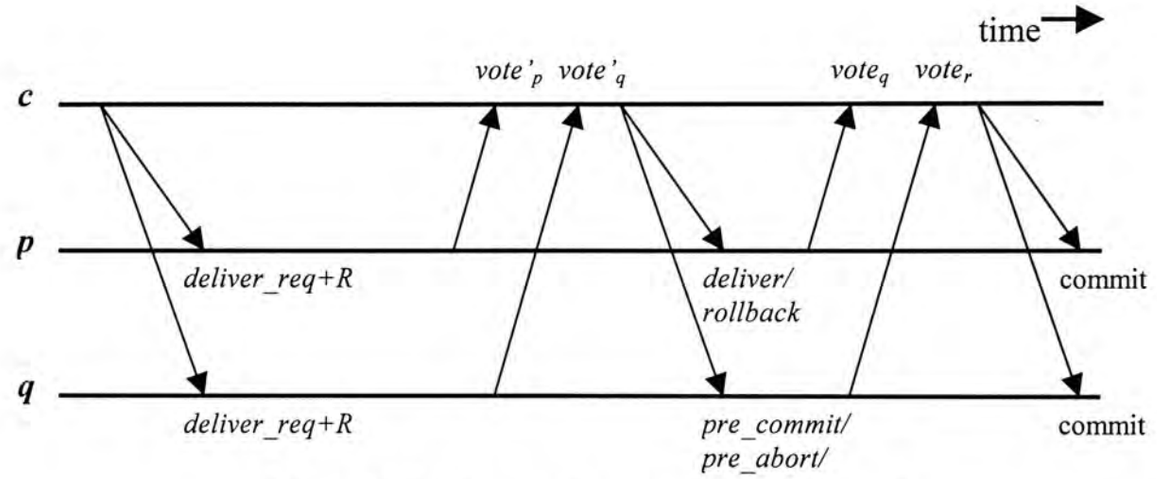

Figure 7.5: The atomic commitment protocol

- If none of the participants vetoes the transaction, the decision will be to deliver *R*. The coordinator thus sends a *deliver* predicate to every agents involved.
- Otherwise, the coordinator will coordinate the rollback of *R* by sending a *rollback* predicate to every agent.

4. In any case, the *deliver* or the *rollback* predicate will be delivered to the participating agents inside a transaction. Hence, each agent has to vote in terms of its readiness to deliver the *deliver* or the *rollback* predicate.
5. *c* gathers all the votes and decides to commit as long as all agents vote yes. As a result, all agents within the group either deliver or rollback *R* consistently. Otherwise, if at least one of the agents voted no, then the coordinator repeats step 4 until all agents voted yes such that the transaction could commit.

Figure 7.5 summarizes the above protocol.

7.3 Uniform Reliable Multicast

The multicast service is synchronized at the coordinator *c* that serves as a multicast server. Each multicast message will be delivered to all group members through a

transaction with enhanced atomicity. Whenever an agent p needs to multicast a message, it issues a request to the c embedding the multicast message M. Each M will be delivered from c to all agents in $V_n(c)$ atomically. The protocol involved is as follows:

1. p first sends a *multicast_req* predicate enclosed with M to c.
2. In response, c verifies the request from q (e.g. checks if M is valid, etc).
3. Next, c will write a *multicast* predicate along with M into the channel of each agent in $V_n(c)$. This is delivered inside a transaction and commits in a two-phase commit manner to guarantee M being sent to all intended recipients atomically.
4. As long as the transaction commits, every agent receives the *multicast* predicate and delivers M.

The above protocol is summarized in Figure 7.6. Once M is delivered, it is totally up to the recipient to interpret the content of these messages and take appropriate actions if necessary.

Above all, each multicast message is totally ordered in both the request and delivery phases. First, at the request phase, the *multicast_req* predicate is totally ordered by the request message timestamp together with sender's priority (as described in section 5.4) before being processed by the coordinator. Next, at the

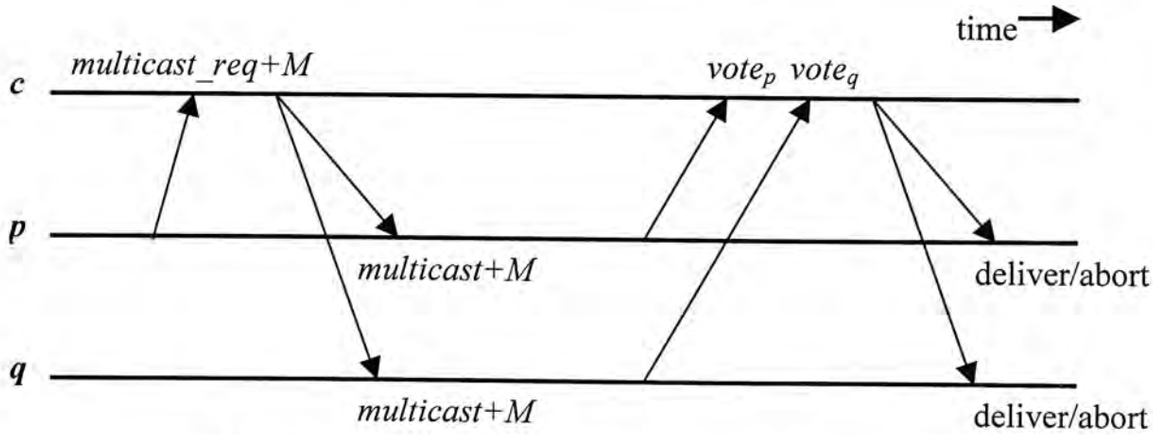

Figure 7.6: The uniform reliable multicast protocol

delivery phase, the *multicast* predicate is totally ordered in the same way.

By the atomicity of the two phase commit protocol, it is trivial to prove that if any member in g has received m, then virtually all members must have received m. Hence, the conditions in section 5.1.3 hold.

Chapter 8

Implementation

This chapter covers the implementation of each individual component, the messaging mechanism as well as the protocol suite with respect to the CoDAC framework.

8.1 Interfaces and Classes

This section gives the details about the implementation of various interfaces and classes defined in the CoDAC framework.

8.1.1 The CoDACAdapterInterface

Despite its complexity, the implementation of DA adapter is transparent to the agent developers. For instance, the DAAdapter class implements the interface, CoDACAdapterInterface which entails a minimal set of operations accessible by the associated agents in order to simplify the complexity of multiagent programming. Agent developers need only to identify the functions entailed in this interface before

engaging into collaboration groups via the DA adapters. In the meantime, this interface does not grant more access privilege than necessary for the participating agents so as to prevent malicious agents from causing harm to other group members through manipulating the DA adapter.

The definition of the CoDACAdapterInterface is listed in API 8.1. To begin with, the addEventListener() operation associates its subclass (i.e. the DAAdapter class) with agents that implements the CoDACEventListener interface (see section 8.1.2), whereas the joinCollaborationGroup() operation lets the associated agent to engage into the desired collaboration group given the group ID. The first startService() operation is to be invoked by the group coordinator who has instantiated a DA adapter locally. In particular, this operation activates the DA adapter, which, in turn, discovers the Jini lookup service for registering its clone as a service proxy. On the other hand, the second startService() operation is to be invoked by agents whose DA adapters are downloaded from the lookup service. In this case, the agent must pass its agent ID to the DA adapter such that the latter can engage into collaboration groups and operate on behalf of the agent (in case of the group coordinator, the agent ID has already been passed through the constructor operation of the DAAdapter class). The second input parameter is an array of ServiceRegistrar objects which are to be used for locating the JavaSpaces and the TransactionManager upon the Jini framework. In either case, both startService() operations require an instance of DAManager as an input parameter because this class inherits the java.rmi.server.UnicastRemoteObject class which is unserializable and cannot be stored in the lookup service (read section 8.1.4 for details). Therefore, the DA manager must be instantiated locally by the agent and then passed to the associated DA adapter. In general, both startService() operations start the execution thread of a DA adapter after which the adapter operates continuously in a self-managing manner.

The DA adapter continues to operate until either the suspendService() or terminateService() operation is invoked. Both operations stop the execution thread of the DA adapter while the latter un-registers the agent from the collaboration group and performs the necessary clean-up of resources. In other words, the

```
package cse.CoDAC.shared

public interface CoDACAdapterInterface
{
        public void addEventListener(CoDACEventListener listener);
        public void joinCollaborationGroup(String groupID);
        public void startService(DAManager mgr) throws IOException;
        public void startService(String agentID,
                                 ServiceRegistrar[] registrars,
                                 DAManager mgr)
                                 throws IOException;
        public void suspendService() throws OutStandingTxnException
        public void terminateService() throws OutStandingTxnException
                                              UnregistrationFailedException
        public void resumeService(DAManager mgr)
                                  throws IOException;
        public void multicast(Object content);
        public void requestResources(Object instruction)
                                     throws UnauthorizedActionException;
        public void submitResources(Object resutls);
        public void startVoting(Object content)
                                throws UnauthorizedActionException;

}
```

API 8.1: The CoDACAdapterInterface definition

terminateService() operation is to be invoked by agents who wish to leave a collaboration group. This method will throw an UnregistrationFailedException if the coordinator does not grant the unregistration. On the other hand, the suspendService() operation is normally invoked when the agent needs to migrate to another host. This operation dissociates the agent from any unserializable objects in the DA adapter such that it can migrate through the object serialization facility. Both the above operations will throw an OutStandingTxnException if the DA adapter is currently involved in an outstanding transaction, in this case the requested operation cannot be granted until the transaction terminates.

After the migration completes, the agent can resume the DA adapter by invoking the resumeService() operation. For the same reason as the startService() method invocation, an instance of DAManager is required as an input parameter. If an agent has suspended for a long time, its membership may have expired when it resumes

(read section 8.4 for details). In this case, the DA adapter will re-register with the collaboration group automatically on behalf of the associated agent.

The multicast() operation is used to initiate a uniform reliable multicast described in section 7.3. This operation takes an instance of the generic java.lang.Object class as the medium for storing the content of a multicast message. Once delivered, it is up to the recipients to typecast the content back to its original class for interpretation.

The requestResources() operation lets the coordinator to request all group members to devote their computation results for collaboration purpose. This method authenticates the identities of the callers upon invocation. If the caller is the coordinator, the authentication passes and the operation triggers the peer adapters to fire a GatherResourcesEvent to the associated agents. Otherwise, it throws an UnauthorizedActionException to the caller. Agents who notify of the request invoke the submitResources() operation to contribute their resources in return. To boost flexibility, the java.lang.Object class is chosen utilized as the medium for storing the computation resources. Hence, an agent can submit any serializable Java object to the coordinator. The kernel will typecast these objects back into their original class before analyzing upon them. The details on resources gathering and analysis are given in Chapter 6.

Similarly, the startVoting() operation verifies whether the caller, who intends to initiate a voting in terms of view update or results delivery, is actually the current coordinator.

8.1.2 The CoDACEventListener

As the CoDACAdapterInterface entails how an agent invokes the behavior of the DA adapter through method invocation. In reverse, the DA adapter signals the associated agent to take appropriate actions by generating certain events. Hence, an agent must implement the CoDACEventListener interface, as defined in API 8.2 in order to interact with the DA adapter. In doing so, an agent must implement its actions in

```
package cse.CoDAC.shared

public interface CoDACEventListener implements EventListener
{
        public void notifyResourceRequest(GatherResourcesEvent evt);
        public void notifyDeliveryRequest(PrepareDeliveryEvent evt);
                                        throws VetoDeliveryException;
        public void notifyCommitDelivery(CommitDeliveryEvent evt);
        public void notifyRollbackDelivery(RollbackDeliveryEvent evt);
        public void notifyMulticastMessage(DeliverMulticastEvent evt);

}
```

API 8.2: The CoDACEventListener API

response to certain events. For example, as mentioned earlier, the notifyResourceRequest() operation is signaled by the DA adapter by firing a GatherResourcesEvent, after which the involved agent should react by invoking the submitResources() defined in the CoDACAdapterInterface to submit its individual computation in return.

The notifyDeliveryRequest() operation is triggered by the PrepareDeliveryEvent, which encloses with some kinds of information (e.g. coordination instruction) to be delivered to the agent. The agent can retrieve these coordination instructions from the event object to determine whether it agrees to deliver (follow) this instruction. If it does not agree to follow the instruction, it must throw a VetoDeliveryException to the DA adapter. Otherwise, the DA adapter assumes an implicit agreement.

The commitDeliveryRequest() and rollbackDeliveryRequest() operations implement the respective behaviors of an agent to deliver and rollback the coordination instruction forwarded from the DA adapter. These operations are triggered by the CommitDeliveryEvent and RollbackDeliveryEvent respectively and both operations implement the actions to preserve consistency on the final state of the agent in accordance to the final decision to deliver or to rollback that instruction.

The notifyMulticastMessage() operation simply notifies the agent to deliver a message sent from the multicast service within the group. The content of the

multicast can be retrieved from the DeliverMulticastEvent as a generic Java object and is up to the agent to comprehend its meaning.

8.1.3 The DAAdapter

The DAAdapter class plays an essential role in the CoDAC framework. As each agent can alternatively perform the role of the coordinator, the DA adapters provide supports for both the coordinator and the ordinary members. Hence, for replication purpose, both sets of operations for the coordinator and the ordinary members are integrated altogether into the DAAdapter class. API 8.3 distinguishes the operations of the DAAdapter class into 3 categories, namely coordinator, ordinary members and all agents in general.

The first set of operations grants the coordinator access to the Jini lookup and transaction services in enhancing atomicity and recoverability. For instance, the atomicWriteAll() operation enforces atomic message multicasts and operates along with both the coordinateAbort() and the coordinateCommit() operations to coordinate all participants to commit and abort a collaboration transaction atomically. The createProxy() operation duplicates the DA adapter as a service proxy to be registered with the lookup service via the registerWithLookup() or the reregisterService() operation. Above all, the findKernel() operation locates and downloads the desired kernel from the lookup services to deliver the reasoning logic to the coordinator. Similarly, the findTransactionManager() operation locates the transaction manager within the Jini framework for coordinating atomic transactions. The reviseCurrentView() operation updates the current group membership whenever an entity joins or leaves the group. The operations, startService(), startVoting() and requestResources() are inherited from the CoDACAdapterInterface and are explained in section 8.1.1.

```
package cse.CoDAC.service;

public class DAAdapter implements Serializable,
                                  CoDACAdapterInterface,
                                  Cloneable,
                                  Runnable
{
        //Methods implemented for the coordinator

        protected boolean atomicWriteAll(CoDACMessageEntry msg);
        protected void coordinateAbort(CoDACMessageEntry received);
        protected void coordinateCommit(CoDACMessageEntry received);
        protected CoDACAdapterInterface createProxy();
        protected CollaborationCore findKernel(ServiceRegistrar reg);
        protected TransactionManager findTransactionManager(ServiceRegistrar reg);
        protected void registerWithLookup();
        public void requestResources(Object instruction)
                                        throws UnauthorizedActionException;
        protected void reregisterService(ServiceItem item);
        protected void reviseCurrentView(CoDACMessageEntry received);
        public void startService(DAManager mgr);
        public void startVoting(Object content)
                                throws UnauthorizedActionException;

        //Methods implemented for ordinary members

        protected void listenServiceEvent(ServiceRegistrar registrar,
                                          ServiceTemplate template);
        protected void prepareForRecovery();
        protected void replaceCoordinator();
        protected void requestForReplacement(String recipient);
        protected void requestNextCanadidate();
        public void startService(String agentID,
                                 ServiceRegistrar[] registrars,
                                 DAManager mgr);
```

API 8.3: The DAAdapter API

```
    //General methods

    protected void abortTxn(Long txnID);
    public void addEventListener(.CoDACEventListener listener);
    protected void analyzeIncomingMessages(CoDACMessageEntry received);
    public Object clone();
    protected void commitTxn(Long txnID);
    protected View currentView();
    protected JavaSpace findJavaSpace(ServiceRegistrar reg);
    protected JavaSpace findJavaSpace(ServiceRegistrar reg,
                                      CoDACMessageChannel template);
    protected void installNewView(View view);
    public void joinCollaborationGroup(String groupID);
    protected void joinCollaborationGroup(CoDACMessageChannel channel);
    public void multicast(content);
    protected void notifyResourceRequest(CoDACMessageEntry req);
    protected boolean openChannel();
    protected CoDACMessageEntry readMessage();
    protected void reset();
    public void resumeService(DAManager mgr)
                              throws IOException;
    public void run();
    public void submitResources(Long xid, java.lang.Object content);
    public void suspendService() throws OutStandingTxnException;
    protected void vote(CoDACMessageEntry req);
    protected void voteAgainst(CoDACMessageEntry req);
    protected void voteFor(CoDACMessageEntry req);
    protected void writeAll(CoDACMessageEntry msg);
    protected void writeExcept(CoDACMessageEntry msg, String name);
    protected void writeMessage(CoDACMessageChannel outChannel,
                                int type,
                                Object content,
                                Long txnID,
                                Transaction txn);
    protected void writeMessage(CoDACMessageEntry msg,
                                Transaction txn);
    protected void writeMessage(String recipient,
                                int type,
                                Object content,
                                Long txnID,
                                Transaction txn);

}
```

API 8.3: The DAAdapter API (Cont')

The second set of operations is specific for the ordinary collaboration members to detect failures and initiate replacement of the current coordinator. First of all, the listenServiceEvent() subscribes the DA manager to a set of remote events that would be triggered by the availability of the service proxy for fault detection purpose, (this will be explained in section 8.4). The prepareForRecovery(), the requestForReplacement() and the requestNextCanadidate() operations notify the potential candidates about the crash of the current coordinator and request these candidates to take over the collaboration group. The replaceCoordinator() operation initiates an election to appoint the caller as the new coordinator. The startService() operation is inherited from the CoDACAdapterInterface.

The remaining sets of operations are accessible to all participating agents in general. For example the addEventListener() operation subscribes each agent to the events constantly fired by the DA adapter throughout the collaboration. The readMessage() operations retrieve all incoming messages deposited into the communication channel after which the analyzeIncomingMessages() operation analyze each message received in order to determine which operation to invoke in response. The abortTxn() and the commitTxn() operations abort and commit a transaction given the transaction ID. The currentView() operation returns the current version of view whereas the installNewView() operation updates the group membership by installing the given view as the current view. The two findJavaSpace() operations locate the JavaSpaces in the network for establishing remote communication channels through the openChannel() operation. Both joinCollaborationGroup() operations engage the DA adapter into a collaboration group. The fist operation identifies the target group by the group ID whereas the second operation identifies the target coordinator by its communication channel. The vote() operation determines whether to vote yes or no by invoking the voteFor() or the voteAgainst() operations respectively. Each of the three writeMessage() operations send messages to an individual recipient at a time whereas the writeAll() operation delivers the given message to all peers but does not make it in an atomic manner. The operations multicast(), notifyResourceRequest(), submitResource(),

suspendService() and resumeService() are inherited from the CoDACAdapterInterface.

8.1.4 The DAManager

The DAManager class inherits the java.rmi.server.UnicastRemoteObject in order to subscribe to remote events fire from the lookup service and the JavaSpaces (see section 8.4). However, it is unserializable and unlike the DA adapter, it cannot be uploaded to nor downloaded from the lookup service. Hence, it must be instantiated locally where it will be associated with the corresponding stub and skeleton.

```
package cse.CoDAC.shared;

public class DAManager extends UnicastRemoteObject
                        implements RemoteEventListener
{
        public static final int JOINREQ = 1;
        public static final int VOTEREQ = 2;
        public static final int VOTE = 3;
        public static final int CASTREQ = 4;
        public static final int CAST = 5;
        public static final int NEWVIEW = 6;
        public static final int COMMIT = 7;
        public static final int ABORT = 8;
        public static final int UNVOTE = 9;
        public static final int SUBMITREQ = 10;
        public static final int SUBMIT = 11;
        public static final int CRASH = 12;

        protected CoDACInternalEventListener listener;
        protected Hashtable collaborationMgrs;

        public void addListener(CoDACInternalEventListener listener);
        public void collectResources(CoDACMessageEntry resource);
        public void countVotes(CoDACMessageEntry vote);
        public void notify(RemoteEvent evt);
        public void startTransaction(Long xid,
                                     View view);
}
```

API 8.4: The DAManager API

This class defines a set of constant integers as the type identifiers for remote messages flowing around the CoDAC framework.The JOINREQ constant indicates a request to the coordinator for granting group membership whereas the VOTEREQ indicates a request to the members for making their votes. The VOTE constant signify a message that contains a vote from the sender. The CASTREQ constant presents a multicast request from the sender whereas the CAST constant signals a multicast message among the group. The NEWVIEW constant signals each member to install the attached version of view. The COMMIT and ABORT constants signal the message recipient to commit and abort a collaboration transaction respectively. The CRASH constant notifies the receipents about the crash of the current coordinator such that some member will initiate an election for a new coordinator, during which the UNVOTE constant can be used to notify the recipient to forget the vote it has made to the sender. The SUBMITREQ constant defines a request to the recipient to submit its computational resources to the coordinator while the SUBMIT constant distincts the submitted resources from other type of messages.

The DAManager class contains two attributes, a listener object implementing the CoDACInternalEventListener interface (see section 8.1.5) and a hash-table enlisting a series of CollaborationManager (see section 8.1.6) object instances. For instance, the addListener() operation associates the DA manager with the given listener object whereas the startTransaction() operation instantiates and store a collaboration manager to the hash-table using the given transaction ID as a key. The countVotes() and collectResources() operations forward the votes and the resources submitted from the participating agents to the relevant collaboration manager for assembling. The notify() operation notifies the listener object about the remote events fire from the lookup service or the JavaSpaces.

8.1.5 The CoDACInternalEventListener

The CoDACInternalEventListener entails the actions to take in response to the underlying event occurrence among the collaboration group. This interface is implemented as an inner class for the DAAdapter, in other words, a DA manager signals the associated DA adapter to react to various events via the CoDACInternalEventListerener interface. For example, the notifyServiceFailure() operation signals the DA adapters to elect a new coordinator once the current coordinator is detected to has failed. The notifyMessageArrival() simply signals the DA adapter to pick-up a remote message from the JavaSpaces as soon as it arrives. The notifyResourcesGatheringComplete() operation notifies and delivers the set of resources collected among the group to the coordinator for further analysis. The notifyTransactionCommit() and the notifyTransactionAbort() operations notify the DA adapter of the coordintor to commit and abort a collabroation transaction respectively.

8.1.6 The CollaborationManager

The collaboration manager is the actual entity that processes the votes and assembles the resources forwarded from the DA andapter via the DA manager. The

```
package cse.CoDAC.shared;

public interface CoDACInternalEventListener
{

        public void notifyServiceFailure();
        public void notifyMessageArrival();
        public void notifyResourcesGatheringComplete(Enumeration resources);
        public void notifyTransactionCommit(CoDACMessageEntry msg);
        public void notifyTransactionAbort(CoDACMessageEntry msg);


}
```

API 8.5: The CoDACInternalEventListener API

CollaborationManager class contains an listener object that implements the CoDACInternalEventListener as an attribute. For instance, the count() operation counts the number of yes and no votes respectively and notify the listener object to commit or abort via this CoDACInternalEventListener interface as soon as it accumulated enough knowledge to make the decision or as it times out. Similarily, the collect() operation assembles the resources gathered from the collaboration context and signals the kernel to analyze upon as soon as the resources are ready or as it times out. The run() operation is inherited from the Runnable interface to start its execution thread for timing the collaboration transaction.

8.1.7 The CollaborationCore

The CollaboratonCore interface defines only one operation that its subclasses must implement, there are altogether three operations defined as shown in API 8.6. The setHost() operation is invoked to associate the kernel to the host DA adapter as the former is downloaded from the lookup service. The start() operation simply signals the kernel to standby for request The examineResources() operation is invoked when the coordinator requests analysis on the set of resources gathered within the collaboration group in order to figure out some coordination instruction (as described in Chapter 6). This operation takes a series of elements, which represents the

```
package cse.CoDAC.shared;

public interface CollaborationManager implements Runnable
{
        protected CoDACInternalEventListener listener;

        public CollaborationManager(View view,
                                    CoDACInternalEventListener listener);
        public void collect(CoDACMessageEntry msg);
        public void count(CoDACMessageEntry msg);
        public void run();


}
```

API 8.6: The CollaborationManager API

```
package cse.CoDAC.service

public interface CollaborationCore implements Serializable
{
        public void setHost(CoDACAdapterInterface adapter);
        public void start();
        public Object examineResources(Enumeration enum);

}
```

API 8.7: The CollaborationCore API

computation results of the participating agents, as an input and returns the final collaboration result to the coordinator in the form of a generic Java object.

8.2 Messaging Mechanism

Agents engaged into a collaboration group communicate with each other by message exchange. Each DA adapter within the collaboration context is associated with a communication channel [FHA99]. These channels unite the collaborating parties on top of the JavaSpaces service (see Figure 8.1) and serve as logical queues that lodge a series of message entries at one end and deliver such entries at the other end. An entry is simply an object that can be stored in the JavaSpaces. At the input end, every DA adapter within the group can write messages into any channel. But at the output end, only the associated DA adapter may retrieve the entries from a channel. The APIs of the CoDACMessageEntry and the CoDACMessageChannel classes are shown in API 8.8 and 8.9 respectively.

The CoDACMessageEntry class comprises the medium of communication within the collaboration context. Each message entry has totally six identifiers namely the IDs of the sender and the receiver of that message, the ID of the group where the

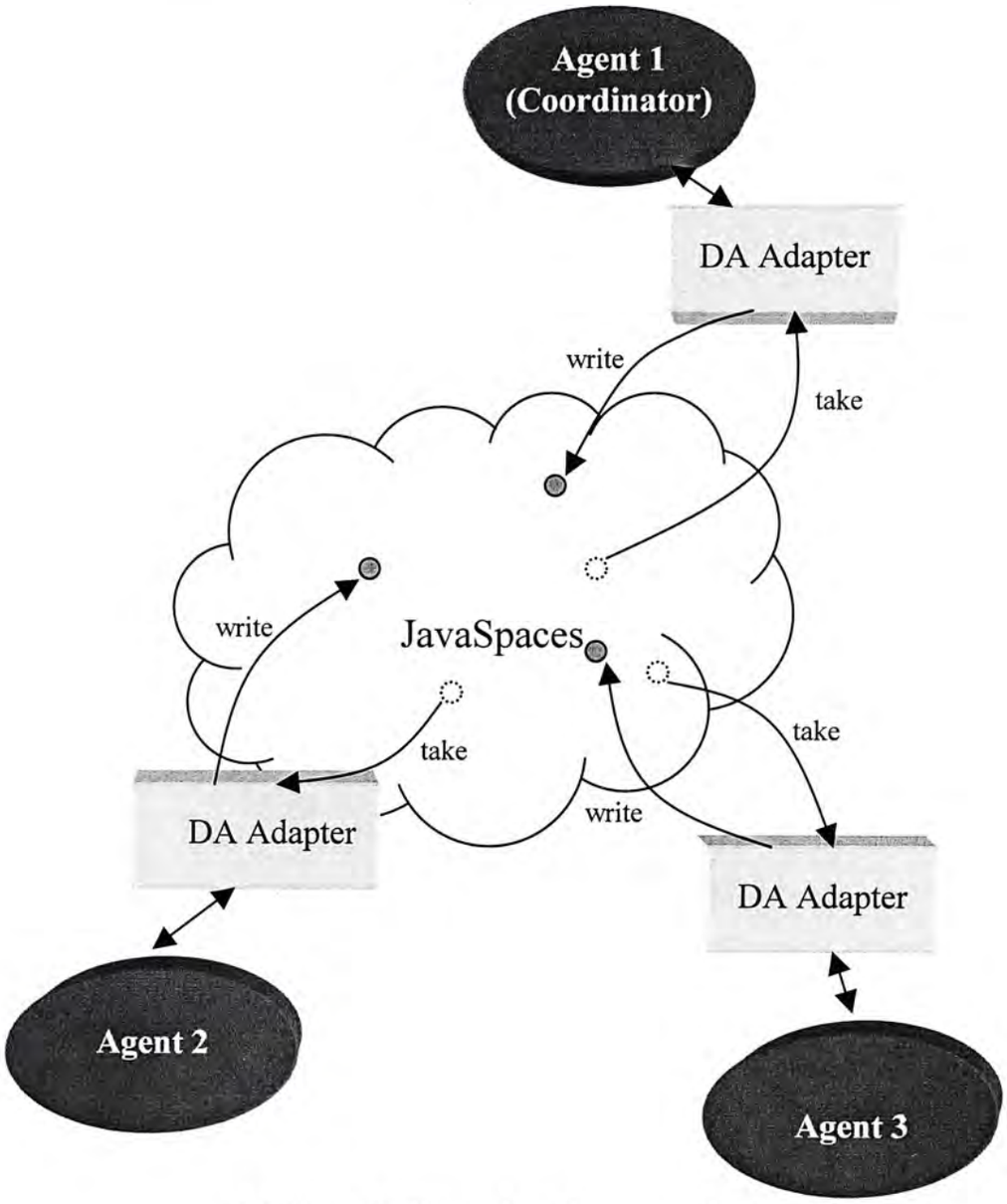

Figure 8.1: Space-based communication channel

message is circulated, the version number of the current view and the message timestamp derived from the Lamport logical clock [Lamp78] at which the message is sent, and the ID of the transaction within which the message is delivered (applicable to messages that are delivered within a transaction as described in Chapter 7).

The content of each message varies depended on the type of messages as defined in the class DAManager. The communication content embedded in a message entry can either be the control instructions circulating among the DA adapters (to enforce

consistent group membership, uniform reliable multicast or fault recovery), or the information shared among the participating agents (to collaborate for consistent actions). In particular for the latter case, the channel should be flexible enough for

```
package cse.CoDAC.shared;

public class CoDACMessageEntry implements Entry
{

        public String senderID;
        public String recipientID;
        public String groupID;
        public Integer viewVersion;
        public Integer timeStamp;
        public Integer msgType;
        public Object content;
        public Long codacTxnID;
        public Boolean totalOrder;
        public Boolean global;
        public Boolean nesting;

        public CoDACMessageEntry(   String sender,
                                    String recipient,
                                    String group,
                                    Integer version,
                                    Integer timeStamp,
                                    Integer type,
                                    Object content,
                                    Long txnID,
                                    Boolean orderFlag );
        public String toString();
        public void setGlobal();  //set global as true
        public void setNested(); //set nesting as true
        public boolean isTotalOrdered();


}
```

API 8.8: The CoDACMessageEntry API

delivering various types of data. Therefore, the content of a message is stored as an instance of the generic java.lang.Object. In this sense, the messaging service in CoDAC can deliver virtually all serializable Java objects.

Above all, the totalOrder flag indicates whether the message should be processed in total order. Causal messages like the instruction to coordinate the installation of a

```
package cse.CoDAC.shared;

public class CoDACMessageChannel implements Entry
{
        public String channelID;
        public String groupID;
        public JavaSpace javaSpace;

        public CoDACMessageChannel() ;
        public CoDACMessageChannel(String name, JavaSpace space) ;
        //write message into queue
        public Lease appendMessageQueue(    CoDACMessageEntry msg,
                                            Transaction txn,
                                            long timeout     )
                                    throws  UnusableEntryException,
                                            TransactionException,
                                            java.lang.SecurityException,
                                            java.lang.InterruptedException,
                                            java.rmi.RemoteException ;
}
```

API 8.9: The CoDACMessageChannel API

new view or the collaboration results, and the request for computation resources to the agents are processed in total order. These messages are totally ordered by the priorities of the group (in case of inter-group message exchange) and the sender agent, together with the message timestamp.

On the other hand, non-causal messages such as the votes and the resources devoted from the agents are simply processed in fist come first served manner. A message entry, once instantiated by the sender, will be written into the communication channel of the recipient and the underlying JavaSpaces will notify the recipient to pick up that message.

After all, as long as agents are inhabited in a hierarchy, the scope of a message needs to be specified. The flag global in CoDACMessageEntry indicates the scope of a message as either local or global. If the flag is true, then the scope is global and the message will be delivered to all agents that constitute the entire hierarchy otherwise the scope is local and the message will be delivered to all agents within the local group only. In addition, the flag nesting defines the way of delivery for global

messages. This attribute specifies whether a global message is to be delivered inside a closed or open nested transaction. This nesting property will be detailed in the next section.

The CoDACMessageChannel class implements the back-end of the communication channel. As shown in API 8.9, this class is, by itself, an entry, hence, each channel, once instantiated, can be stored into the space. Each channel is identified by both the channelID and the groupID attributes. Normally, the ID of a channel is identical to that of the associated agent (i.e. the ID of the recipient for every message being written into this channel) whereas the groupID attribute equals to the ID of the group that the host agent currently engaged in. In special cases where inter-group communication is desired, the local group coordinator will be granted with an additional communication channel whose channelID is assigned with the ID of that local group whereas the groupID attribute is the ID of the ancestral group.

The CoDACMessageChannel class is implemented as a reactive tuple space that provides a simple method appendMessageQueue() to write a message entry at the end of the queue. Any interested party who wishes to write a message to another first matches and reads the corresponding channel entry from the space with the specific groupID and agentID of the intended recipient, and then invokes its appendMessageQueue() method to deposit a message entry to the channel. The deposited message entry is actually written and stored in the space awaiting the recipient agent's pick up.

JavaSpaces provide support for distributed object persistency, concurrent access and distributed transaction. In particular, distributed transactions are enforced by the two-phase commit protocol [Sun99e]. Under this transactional semantics, multi-operation and multi-space updates can complete atomically. In this case, an object implementing the net.jini.core.transaction.Transaction must be passed as an input parameter to the appendMessageQueue() operation.

8.3 Nested Transaction

As defined in section 5.5, a collaboration can take place either in a global or local scale. For instance, global collaboration transactions involve coordination among separate local groups. Such inter-group collaboration, in turn, can perform either as a closed or open nested transaction [Elm92].

In the first case, the collaborations of the participating agents in local groups correspond to sub-transactions nested within the global collaboration transaction. These sub-transactions are managed by the local group coordinators under the coordination of the root coordinator in harmony with the top-level atomicity. This goal is achieved by sharing the same object instance of Transaction throughout the hierarchy such that the Jini transaction manager could coordinate the entire group of agents to commit or abort atomically.

For the second case, the local transactions will be separated from the global transaction such that each top-level collaboration transaction can commit immediately before spanning down to the next level. A local coordinator must coordinate the local collaboration transaction to commit as the top-level groups did and then requests the consecutive lower level groups to commit accordingly and so on. As the sub-transactions are independent from the global transaction, each local group instantiate a separate Transaction object and let the transaction manager to coordinate commit or abort individually. Obviously, the state of a global transaction with open nested transaction may be inconsistent as it proceeds, but the final state must be consistent as it terminates. Therefore, in addition to the distributed object persistency provided in JavaSpaces, each individual local transaction must be backed with a compensating transaction [Elm92]. The compensating transactions undo the committed transactions, such that if a local group aborts, then all collaboration groups on top of it must rollback through compensating transactions in order to preserve the consistency of the final state.

The open nested transaction is more favorable to the actual network environment as the network is usually not stable enough to support large-scale extensive communication sessions. Further, due to partial failures and network latency, simultaneous connection among the entire collaboration group during a global transaction is impractical.

8.4 Fault Detection

Aside from the dynamic changes in the group membership triggered by requests (i.e. join or leave requests), there are circumstances that an agent may be terminated gracelessly due to various unexpected site failures or isolation from the group subjecting to a combination of communication failures. In these cases, the agent will not be able to collaborate with its peers and will be regarded as virtually gone before it explicitly un-registers itself. Although it may be against one's free will, the membership of a failed or isolated agent must be reclaimed to keep the collaboration running. For this reason, the group membership protocol described in Chapter 7 is designed to tolerate and recover from such faults in the distributed environment. But on top of that, such failures should be detected effectively in order to trigger the recovery in the first place.

The overhead for fault detection in CoDAC is minimal as it takes advantages of the leasing feature in the Jini framework. A lease is a grant of guaranteed access over a time period. Access to many of the services in the Jini system environment is lease based. Each lease is negotiated between the user of the service and the provider of the service. The holder of the lease may renew or cancel the lease before it expires. If the leaseholder does neither, the lease simply expires and the leased service is freed. For instance, our fault detection utilizes the lease model in two ways:

1. When a coordinator starts a collaboration group, it registers a serialized instance of its DA adapter to the lookup service as a service proxy as mentioned in section 6.1. This registration is granted with a lease along with the service ID. The leaseholder (i.e. the DA adapter of the coordinator) is held liable to renew the lease repeatedly throughout its life span such that the coordination group remains visible in the neighborhood and open to any interested agents. Such lease renewal, in turn, testifies the existence of the coordinator. Upon joining a group, an agent subscribes to all events associated with the collaboration group service with the given service ID. If the coordinator crashes and is no longer active, the lease will eventually expire. After which the lookup service will free this collaboration service from its storage and fire a distributed event to notify the subscribers. Such removal of a registered service proxy from the lookup service is regarded as failure in the coordinator and is detected from remote events.
2. Whenever a DA adapter opens a communication channel, it instantiates a CoDACMessageChannel and put it into the space as described in section 8.2. When this channel entry is written into space, it is granted with a lease that specifies a period of time for which the space guarantees to store this entry. The DA adapter as a leaseholder must constantly renew the lease such that interested parties could retrieve this channel from the space and leave messages in it. In the meantime, this channel entry symbolizes the availability of the DA adapter as well as the associated agent. If the agent terminates, so does the DA adapter, and the lease of the channel will eventually expire. By then, the channel entry will be freed from the space and no longer retrievable by other agents. In this sense, the agent is regarded as failed.

The first approach is specific for the detection of failure in the coordinator. This is done by the collaboration members who simply register to the lookup service for subscription to the relevant events. When a coordinator is detected to have crashed, a recovery protocol will be triggered to select a single collaboration member to take its place.

On the other hand, the second approach is applicable to both the coordinator and the collaboration members. When a coordinator fails to write messages to a member whose channel is not available in the space, the coordinator will reclaim its membership by assuming this failed member has issued a *leave_req* and proceeds with the leave protocol described in section 7.1.2. The functioning of the collaboration members is, to some extent, considered "don't care" in the sense that the failure of one member does not directly affect the operation of another. After all, the necessary recovery is nothing more than updating the group membership or rolling back those outstanding transactions that a failed member has involved in. Therefore, the presence of the collaboration members is only checked when they need to be involved (i.e. whenever the coordinator needs to write messages to them) where their leases on the channels may be expired long before being noticed by the coordinator.

On the contrary, the coordinator is the core of the collaboration group. It needs to be available all the time to serve newcomers and send off those intending to leave. The coordinator initiates the start and coordinates the end of collaborations, which would be blocked otherwise without it. Apparently, it is necessary to keep exactly one coordinator working at all time. For this reason, we need to detect any failure in the coordinator as early as possible. Therefore, in addition to the first approach of fault detection, whenever a member detects the channel of the coordinator has gone as it tries to reach the coordinator, it triggers the recovery protocol for replacement. This is necessary, as there may be circumstances that the lease on the coordinator's channel expires before the lease on the collaboration service proxy does such that the collaboration can be resumed earlier. After all, the Jini remote event model does not guarantee that every subscriber will receive those events atomically for some agents may miss an event even if it has actually fired. To ensure the failure of a coordinator to be detected by at least one member in a timely fashion, CoDAC adopts both approaches to monitor the presence of the coordinator.

In either case, once an agent is detected to have failed, the recovery protocols will be invoked to update the group membership. Refers to Figure 7.4, the first step is

necessary because, on one hand, the Jini remote event model does not deliver events to every subscriber in a consistent manner, some members might receive such events while some others do not. On the other hand, *p* may detect the failure of *c* earlier than its predecessors do as it attempts to request services from *c*. Therefore, in any case, *p* should assume its predecessors may not necessarily be notified of the failure in *c*, and should notify them explicitly.

Every member has a voice in electing the new coordinator as common link failures might isolate *c* from p but not from *q* or *r* causing *p* to has a false perception that *c* is crashed. Besides that, there may be more than one candidate competing as the coordinator at a time. In such cases, other members (e.g. the DA adapters of *q* and *r*) are crucial to make the right votes to ensure exactly one coordinator exists. For instance, we adopt the simple majority (i.e. $\lceil (N+1)/2 \rceil$ where N is the total number of members in a group) as the election criteria to preserve the exactly-once semantic even when subjected to network partitioning.

At the end, the newly elected coordinator must register a clone of its DA adapter to the lookup service as the lease on the original proxy has expired. This newly created service proxy must be registered with the same service ID and group ID in order to keep the collaboration group open to any potential participant in the neighborhood. A new participant can download this proxy to engage into the collaboration group as usual. This new proxy operates identically to the expired one as they are virtually the same. After that, the coordinator must obtain an instance of kernel from the lookup service and plug into the associated DA adapter before it becomes capable to coordinate the group.

8.5 Atomic Commitment Protocol

In Chapter 7, we have described both the agent and the associated DA adapter as a whole in terms of the design of the protocol suite. At the implementation level, the DA adapters perform all the work on behalves of the agents in the group membership

protocol suite. The same holds for the multicast protocol while the DA adapters simply notify the end results to the associated agents. However, the atomic commitment protocol is the only protocol that involves the associated agents in making the decision to commit or abort. In this section, we will reveal the underlying interactions among the agents, the DA adapters and the collaboration manager.

8.5.1 Message Flow

As mentioned, after the kernel has finished computing the collaboration results R, it returns R to the collaboration manager. The collaboration manager is then responsible to coordinate all agents in $V_n(c)$ to deliver R consistently in order to terminate the collaboration transaction. The interactions involved is as follows:

1. The collaboration manager sends a *deliver_req* predicate enclosed with R to every DA manager within the collaboration context (i.e. $V_n(c)$).
2. Next, each DA manager fires a PrepareDeliveryEvent, embedded with R, to the associated agent.
3. In response, each agent checks its own state to see if it can commit to R. The agents may throw a VetoDeliveryException to vote against delivering R, or it may remain silent to indicate an implicit agreement.
4. The DA managers return the appropriate *vote*'s (either yes or no) to the collaboration manager on behalf of the participating agents
5. The collaboration manager collects all the votes among the group
   - If none of the participants vetoes the transaction, the decision will be to deliver R. The collaboration manager will coordinate all DA managers to delivery R by initiating a Jini transaction to forward a *deliver* predicate to every DA manager.
   - Otherwise, the collaboration manager will coordinate the rollback of R by initiating a Jini transaction to deliver a *rollback* predicate to every DA manager.

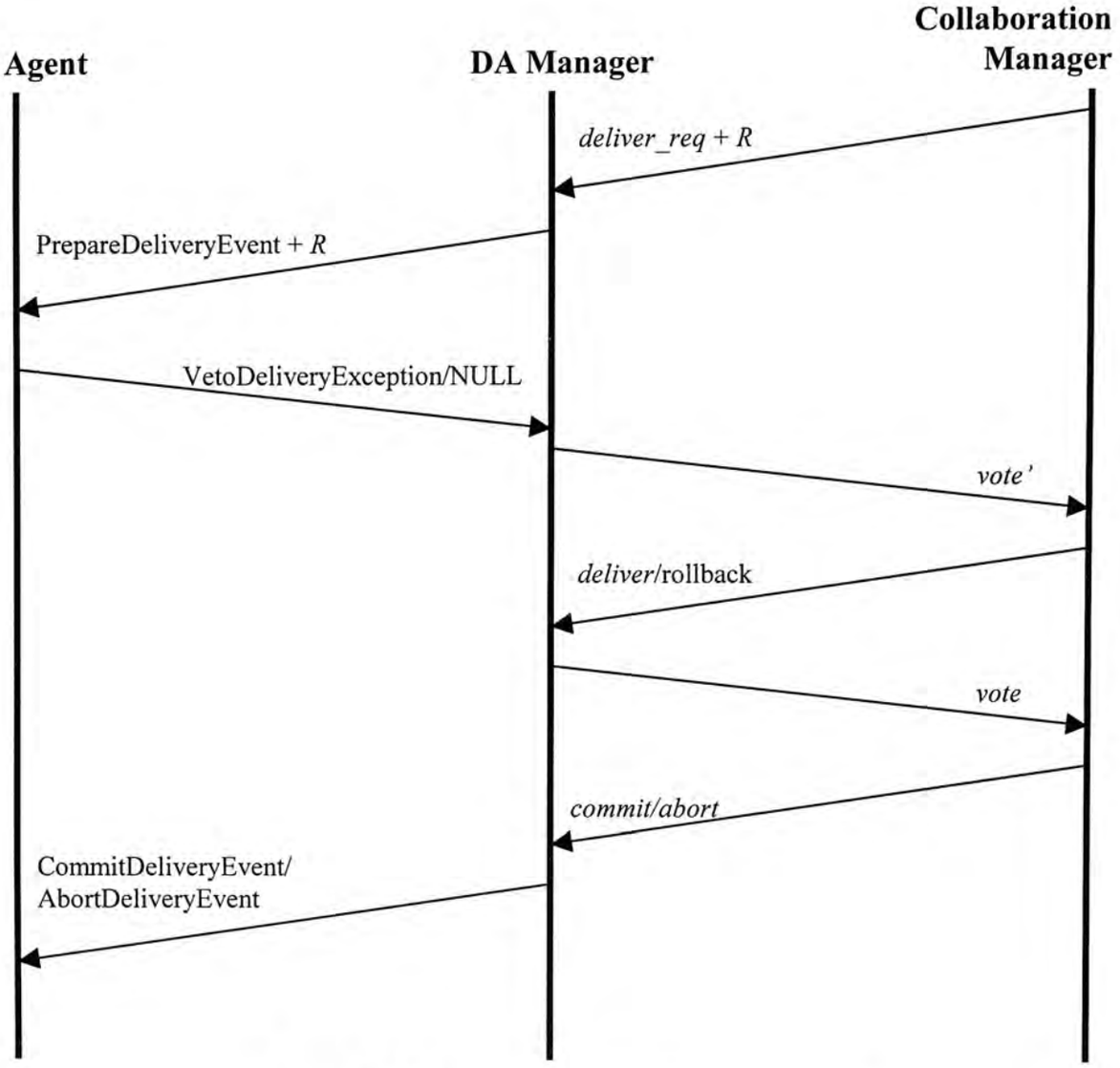

Figure 8.2: The atomic commitment protocol

6. Finally, each DA manager receives either a *deliver* or *rollback* predicate as the transaction terminates. The DA manager then signals the agent whether to deliver or abort *R* by firing the CommitDeliveryEvent or AbortDeliveryEvent respectively.

Figure 8.2 summarizes the above protocol. For simplicity, only one agent and one DA manger is shown to interact with the collaboration manager. Delivery of each *R* is totally ordered by the transaction ID and terminates in sequence.

Note that a Jini transaction corresponds to a distributed transaction between the Jini Transaction Manager [Sun99e] and the JavaSpaces, which is transparent to the agents and even the DA adapters. In other words, the *vote* shown in the above figure does not indicate the DA adapter actually voted for or against delivering the *deliver* or the *rollback* predicate generated in step 5. Instead, these votes are actually made by the underlying tuples in the space to indicate their readiness to store the relevant message entries. The same situation holds for all $vote_x$ ($x = p$, q or r) in Figures 7.2 to 7.6. On the contrary, the *vote'* does denote the votes made at the free will of the agents through the DA adapters and the same holds for $vote'_x$ in Figure 7.4.

8.5.2 Timeout Actions

Whenever the delivery of R starts from step 1, there are two phases in the protocol where some CoDAC entity is waiting for remote messages: in the beginning of step 5 and step 6. As remote messages may get lost or their delivery time may vary due to link failures or network latency, these phases are bounded to a timeout delay ∂to trigger fault discovery. The actions triggered by a timeout are explained as follows.

In step 5, the collaboration manager is waiting for votes from all the DA managers. At this stage, the collaboration manager has not yet reached any decision. In addition, no participating agent can have decided to commit. Therefore, as it times out without getting all vote to make the decision, (e.g. because of a vote is lost or delayed, the agent has crashed or even the request has not reached the agent in the first place) the collaboration manager can decide to abort and proceed to step 6 by sending a rollback predicate to every DA manager.

In step 6, a DA manager that voted Yes is waiting for a *deliver* or *rollback* predicate in return. In this case, the DA manager cannot unilaterally decide to rollback because the Jini transaction guarantees that either one of these two predicates will eventually reach all DA managers as long as the collaboration manager (and the associated coordinator) keeps functioning, although the delivery

time may vary after all. Therefore, the DA manager should not decide to rollback unless it gets a *rollback* predicate or has certified the coordinator as crashed. In other words the timeout triggers a fault discovery and the necessary recovery procedure. This is done as follows:

1. When a DA manager $dmgr_p$ times out in step 6 of the commitment protocol, it retrieves the coordinator's channel in the space and write a *decision_req* predicate to it. If the channel can not be found in the first place, then the coordinator may have failed and $dmgr_p$ thus proceeds with the recovery protocol described in section 7.1.4. Otherwise, $dmgr_p$, waits for another ∂ units of time before it re-issues the *decision_req*. $dmgr_p$ may also break the loop and proceed with the recovery protocol anytime it receives a distributed event from the lookup service that indicates the expiration of the registered proxy.
2. On the other hand, the collaboration manager, in response to the decision_req, checks if it has gathered enough votes to make the decision. If it possesses enough knowledge to decide or if it has actually decided but the decision somehow has not been delivered to the agents yet (perhaps due to network latency), then the collaboration manager retransmits the decision to all DA managers inside a Jini transaction given the same transaction ID. Otherwise, it waits until either all votes are gathered or its timer expires and to deliver the appropriate decision by then.
3. In the worst case where the original coordinator has crashed, the new coordinator c' elected from the recovery protocol coordinates all agents to rollback. The decision is to rollback because there is no way to tell what vote the original coordinator has made before it fails and the vote made by any member that fails in between the commitment and the recovery protocols is unrecoverable too. In any case, the atomicity is well preserved because the Jini transaction model guarantees no participating agent can have decided to commit thus far. Hence, c' can rollback the delivery of R by distributing a *rollback* predicate inside a Jini transaction to all agents in $V_{n+1}(c')$.

Chapter 9

Example

To help understand the application of CoDAC, we give a simple example to illustrate various functionaries of the CoDAC framework in assisting multiagent collaboration. In this example, we simulate an auction with multiple auction agents engaging into a collaboration group of top of CoDAC facilities. The system model and the implementations are covered in the following sections.

9.1 System Model

An auction proceeds in a sequence of rounds. Each agent engaged in the auction group constantly quotes a bid at each round. An auction agent bids repeatedly with the amount raised by a predefined unit in successive rounds until the amount exceeds the maximum price specified by the agent user. The group coordinator assembles all bids made at each round and screens out the bid that quoted the highest price (i.e. the highest bid). This highest bid concludes the current round and is treated as the

collaboration result in a CoDAC collaboration group. Hence, the information about the highest bid will be delivered to every agent within the group.

Upon requested to deliver the collaboration result, the bidder who made this highest bid simply agrees to deliver the result, other agents check whether they afford to quote a higher price in the next round, if yes, then the agents veto the collaboration result. Otherwise, they give up bidding and agree to deliver the result. The auction proceeds to the next round as long as at least one of the agents vetoes the collaboration results. At the end of an auction, every participating agent will install the collaboration result, which announces the winner of the auction who can possess the auction item, atomically and consistently.

9.2 Auction Lifecycle

In this section, we explain how the lifecycle of our simulated auction matches to the three phases described in Chapter 6 that constitute a general collaboration lifecycle.

9.2.1 Initialization

At the very beginning, the kernel for the auction group, denoted as the auction kernel, must be made available to the auction agents, through the KernelLauncher class. Upon execution, the kernel launcher prompts a simple dialog as shown in Figure 9.1 to assist the organizer of the auction to initialize the auction by specifying the auction item (i.e. the item open for bid), together with the base price for that item (i.e. the minimum price one must quote in order to bid the item). Once the organizer finishes input by clicking the OK button, the launcher creates an instance of AuctionKernel and initializes both its base price and item attributes as user input values. Afterwards, the launcher discovers every lookup service in the network neighborhood and

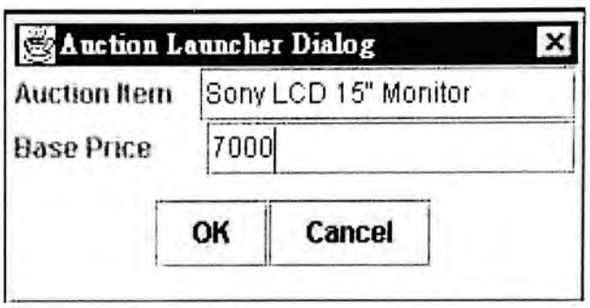

Figure 9.1: The kernel launcher dialog

registers a serialized instance of the kernel object to every lookup service available as shown in Figure 9.2. Once the serialized kernel is granted with a service ID, it has been registered and stored as a service proxy successfully onto the lookup service, and will be downloadable to the bidders on request. After all, the launcher must remain active throughout the life span of the collaboration group, as it is responsible to constantly renew the lease on the kernel. Otherwise, if the lease on the kernel expires, any new coordinator elected for replacement would have nowhere to download the auction kernel and the auction could not proceed.

Once the kernel is ready, the bidders could launch their representative agents using facilities provided by their custom agent platforms. In this example, we choose

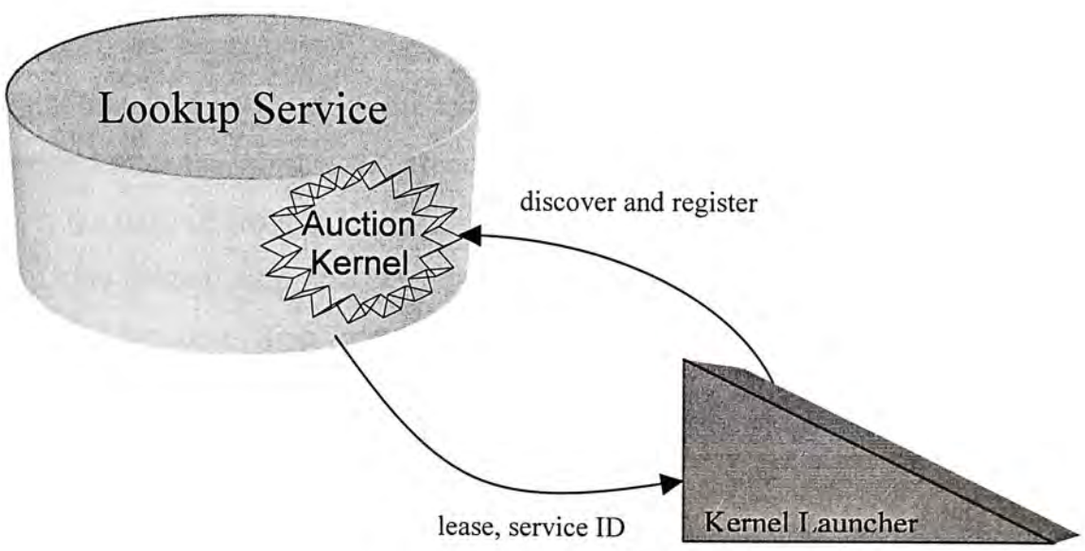

Figure 9.2: Registration of kernel

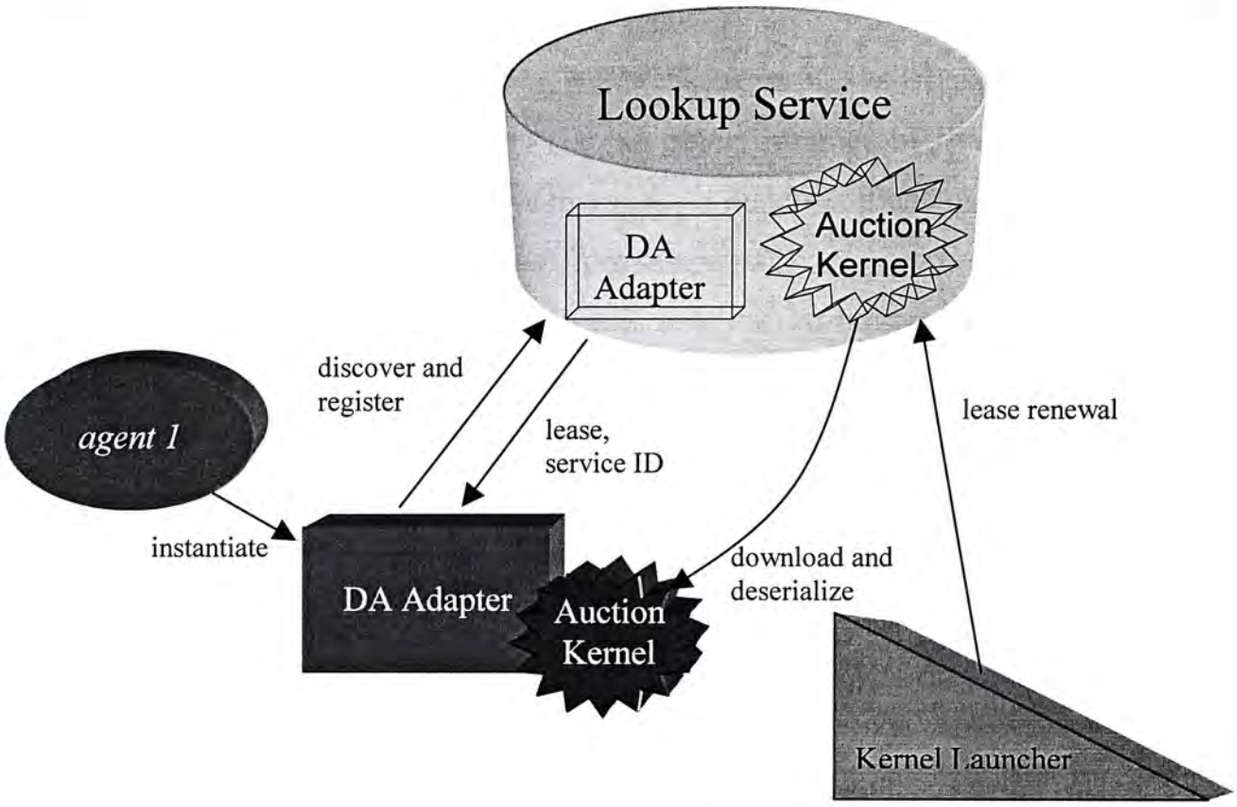

Figure 9.3: Initialization of the coordinator

Grasshopper [IKV98] as the application test bed. Suppose, our simulated auction is expected to take place at a collaboration group with ID, *group1*. Then each agent instantiated will first discover the lookup service to look for any registered instance of DAAdapter in the form of service proxy having its groupID attribute set as *group1*. Suppose we have *agent1*, instantiated from the class AuctionAgent, being the first agent launched and none of the existing DA adapters registered in the lookup service entitled with the *group1* ID. Hence, *agent1* creates its own instance of DAAdapter with the desired group ID.

Upon instantiation, the DA adapter discovers all available lookup services within its neighborhood. Next, it duplicates itself and registers its clone with each lookup service available as shown in Figure 9.3. Once the registration is granted, the DA adapter of *agent1* is held liable to constantly renew the lease on its clone. In the meantime, *agent1* becomes the coordinator for *group1* and hence it has to download an auction kernel for its DA adapter to deliver the necessary logic to host an auction.

On the other hand, suppose *agent2* is launched next. Since, agent2 is instantiated from the same class as *agent1*, again, it first discovers all available lookup services to look for a matching service proxy (i.e. an object instance of DAAdapter with the groupID attribute set as *group1*). Obviously, a match is found this time and hence *agent2* simply download the adapter from the network as shown in Figure 9.4. Once deserialized, the cloned DA adapter issues a request to the original DA adapter for engaging into *group1* on behalf of *agent2*. All successive agents engage into *group1* in a similar manner as *agent2* does throughout the life span of *group1* and the auction may start when significant auction agents are engaged and ready.

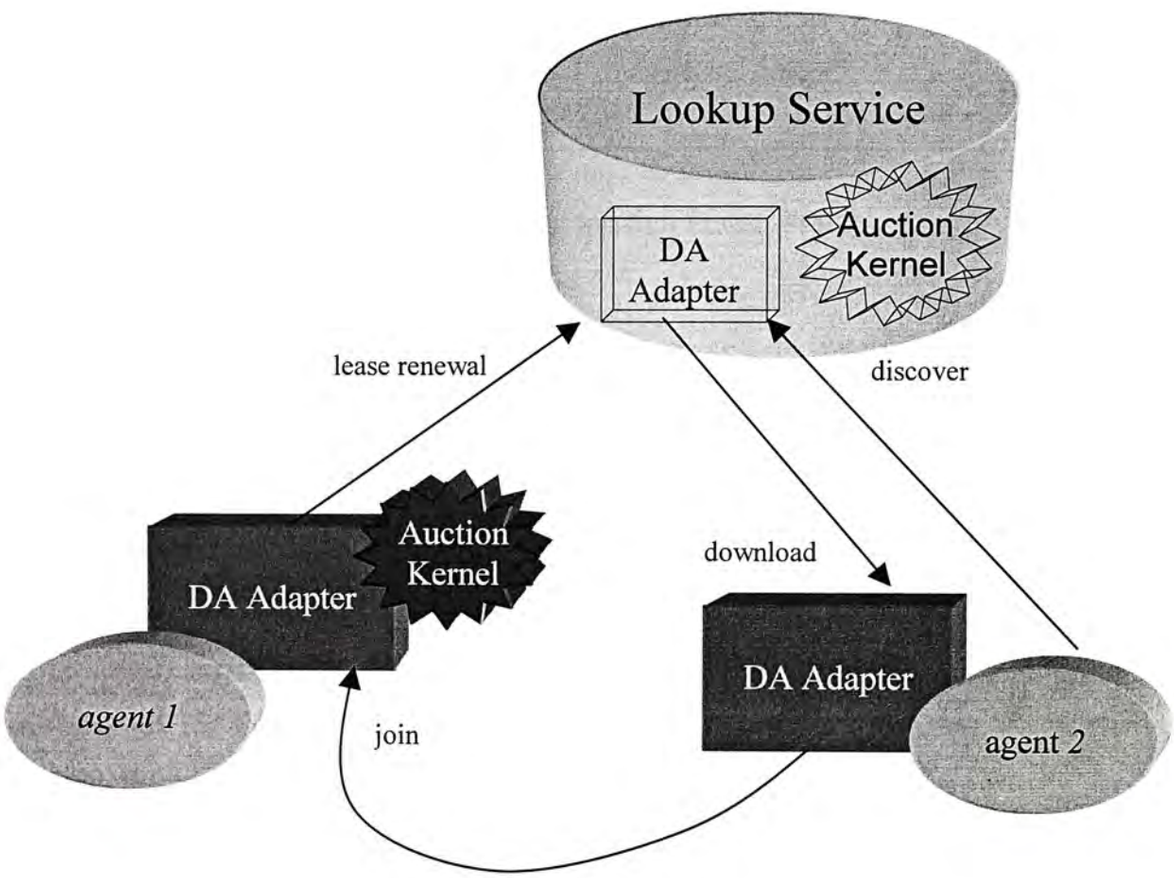

Figure 9.4: Initialization of the collaboration group

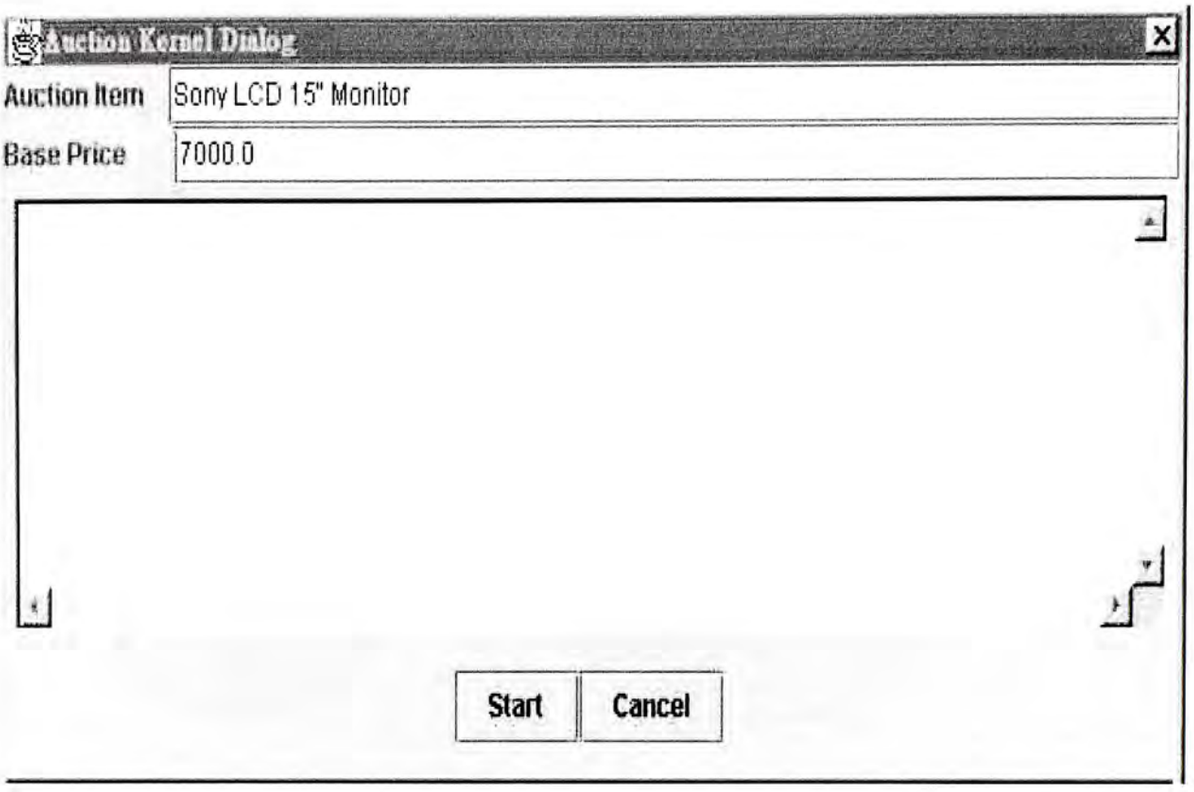

Figure 9.5: The auction kernel dialog

9.2.2 Resource Gathering

For demonstration purpose, we have associated the auction kernel with a GUI denoted as Auction Kernel Dialog as shown in Figure 9.5. This dialog displays the item opening for bidding as well as the base price to begin with. Further, this dialog will show relevant information on-screen from time to time for demonstrating the progress of an auction. For instance, the kernel will be activated manually by clicking the start button on the dialog.

Once the auction kernel is activated, it starts requesting the participating agents for computational resources. The relevant resources in our model are simply the bids that the agents could afford to quote in each round. In other words, the kernel starts the auction by requesting all involved agents to place their bids. The requests are actually sent to agents through the DA adapters as shown in Figure 9.6. Note that

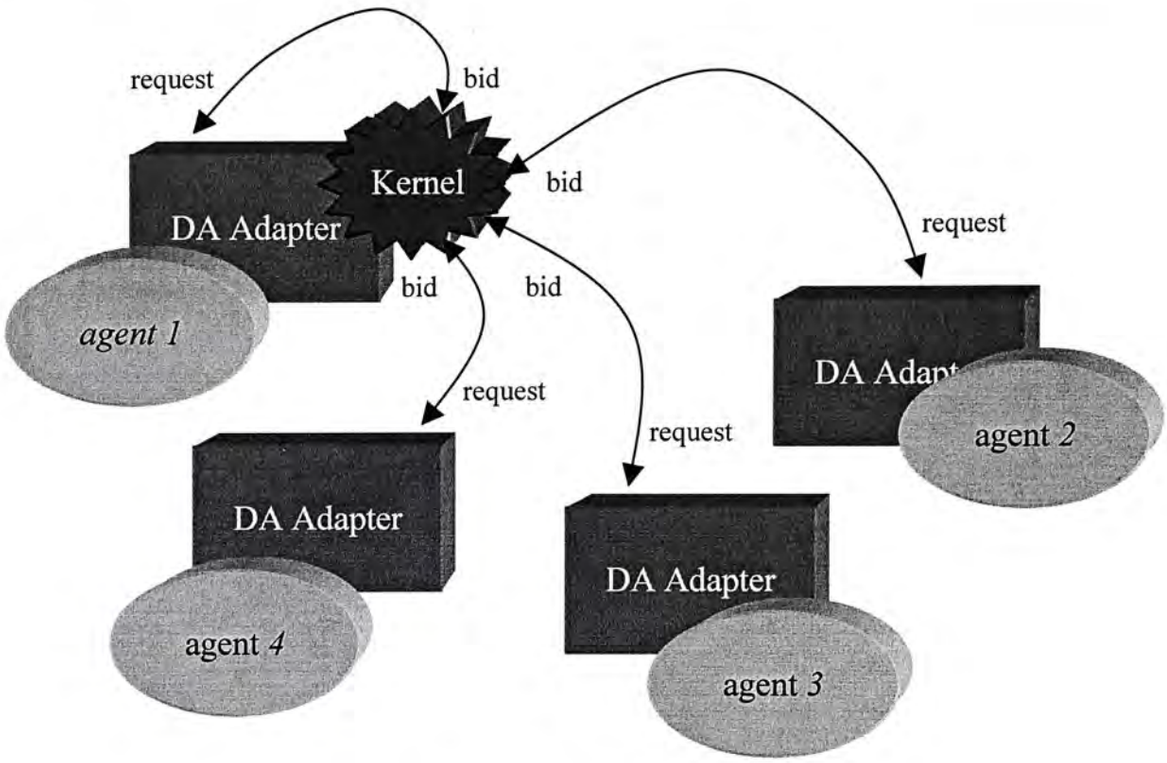

Figure 9.6: Bid gathering

starting (or ending) an auction, or any kind of collaboration in general, is the job of the coordinator. However, there is no clear specification on where the coordination logic should inhabit. Normally, the coordination logic can be implemented either in the agent or the kernel classes. For simplicity, instead of replicating the coordination logic into every instances of agent (for fault-tolerance purpose), we choose to integrate the relevant logic into the AuctionKernel such that the AuctionAgent class needs only to deliver the necessary logic to follow the kernel's coordination. In some cases, where an actual coordinator agent is desirable, the coordination logic should be integrated to the agent whereas the kernel simply performs the analytic works to assist the coordinator agent in making decisions.

On the other hand, the AuctionAgent class is associated with a GUI as well, to ease demonstration. As shown in Figure 9.7, the Auction Agent Dialog accepts two constraints, namely the increment price and the maximum price. The increment price I defines the amount an agent should raise in its bid for each successive round

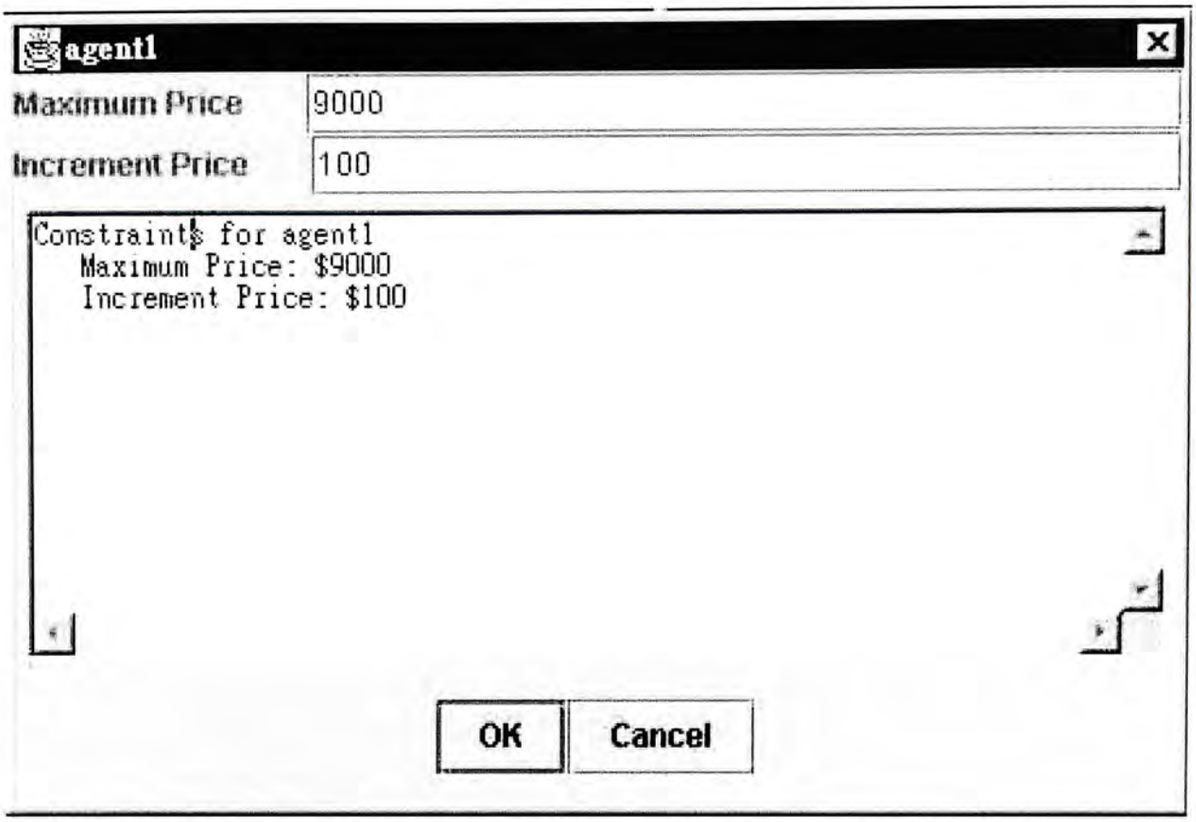

Figure 9.7: The agent dialog

whereas the maximum price *M* defines the maximum amount that an agent afford to quote for the auction items.

In response to the request from the kernel, an agent return a bid base on the following strategy summarized in Table 9.1:

In either case, the agents define the amount they afford to quote and the associated DA adapters bid on behalves of the agents. Afterward, the kernel assembles all the bids from the auction agents and proceeds to the next phase.

9.2.3 Results Delivery

Upon analyzing the bids assembled among the auction group, the kernel screens out the bid quoting the highest price *H* in the current round. The collaboration result *R* is defined as *H* together with the ID of the corresponding bidder as shown in Figure 9.8. To begin with, a request to deliver *R* is sent to each auction agent as

shown in Figure 9.9. In response, if the agent is the one who quoted *H*, it simply agrees to deliver *R*. Other agents check if *H* is larger that their *M's*, if so, the agents cannot afford to bid at a higher price that *H* in the next round and thus give up and agrees to deliver *R*. Otherwise, an agent may veto delivering R if it affords to quote a higher bid in the proceeding rounds.

	Case	**Action**
1	First Round	quote the base price defined for the item
2	Proceeding Rounds	define $P = min\{H+I, M\}$, where *H* denotes the highest price quoted at the previous round if $P > H$, quote *P* otherwise, give up

Table 9.1: Auction strategy

Each DA adapter will return a vote to the coordinator to indicate the agent's intention to deliver *R*. If at least one agent voted to veto delivering *R* then the coordinator will coordinate all agents to abort *R*. Otherwise, the auction terminates as the coordinator coordinates all agents to deliver *R*. At the end of an auction, every agent obtains the information about which agents has win the bid and at what price did that agent quoted for the auction item.

If the decision is to abort, the auction will proceed to the next round as the kernel initiates another request for bid. The cycle will loop back to the resource gathering phase and proceed until one agent win the bidding as all others give up. The on-screen display on the agent dialogs is shown in Figure 9.10 where *agent2* has quoted the highest price among the group to bid for the LCD monitor. *agent2* is thus the winner of the auction and every auction agent will install the final result.

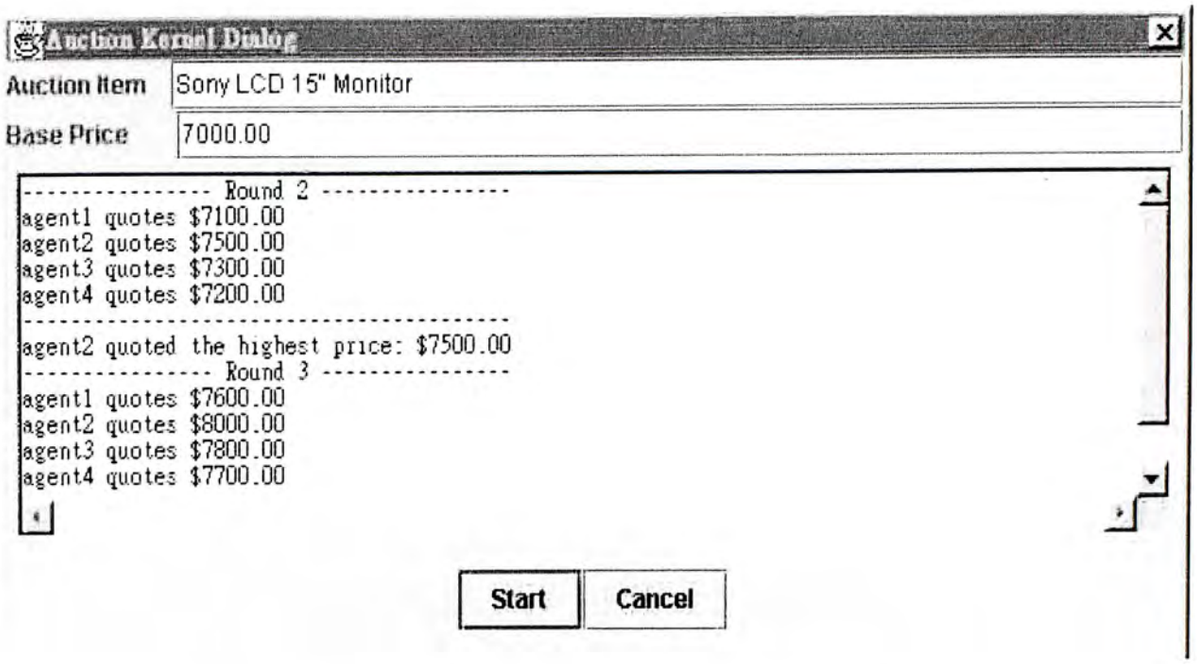

Figure 9.8: On-screen output from the kernel

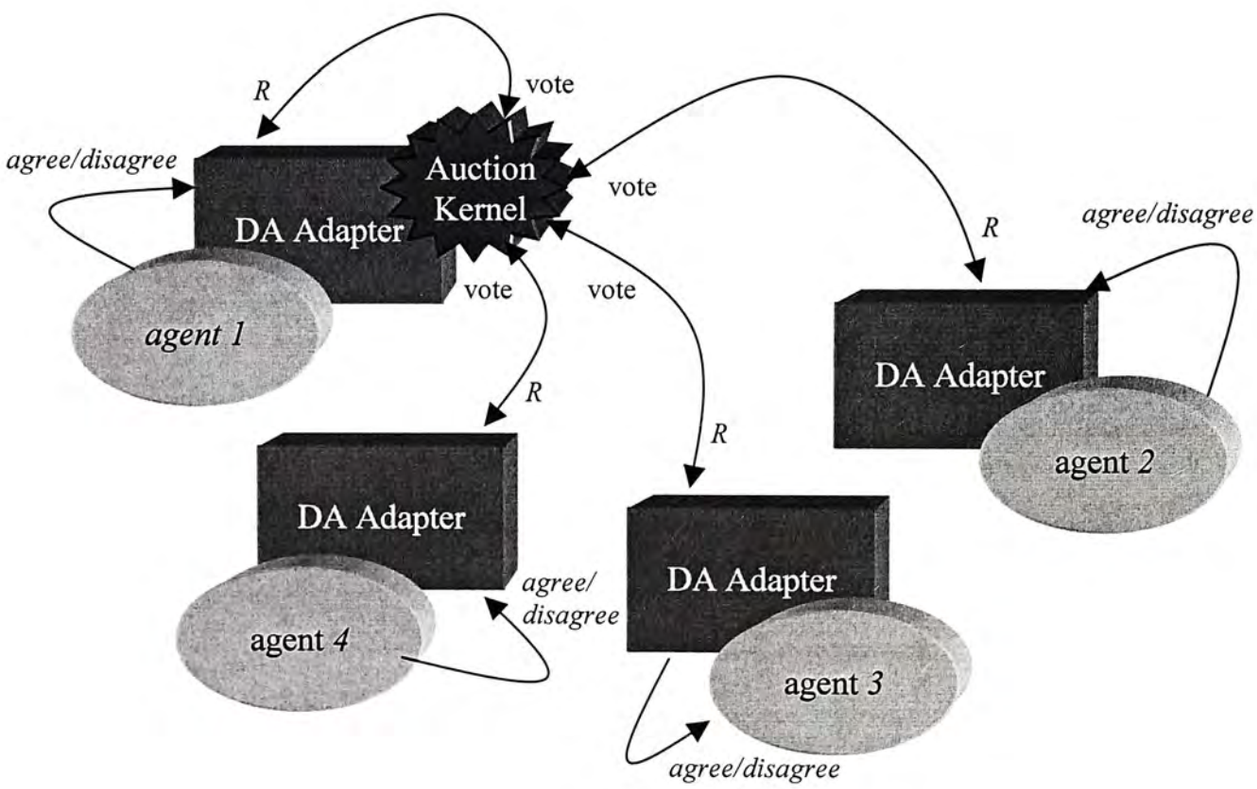

Figure 9.9: Result delivery

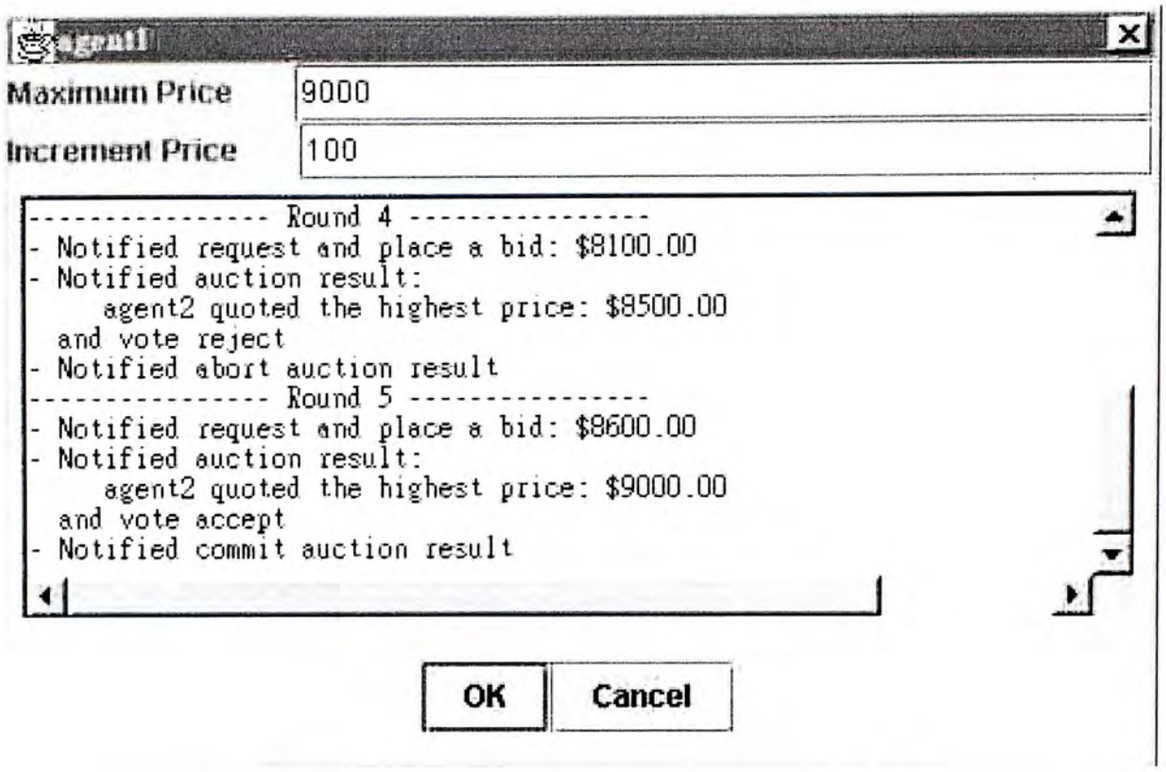

Figure 9.10: Auction termination on-screen display

Chapter 10

Discussions

This chapter summarizes the characteristics of the collaboration framework implemented in CoDAC as the key contributions it delivers. These characteristics include compatibility, hierarchical group infrastructure, flexibility, atomicity and fault tolerance. We will explain each of the above characteristics and their implication to the multiaget paradigm in details in the following sections.

10.1 Compatibility

CoDAC is designed to be loosely coupled to the underlying mobile agent platform, and is not bound to any specific agent system implementation. CoDAC is developed on top of the standard Jini framework and is compatible with virtually all mobile agent frameworks that operate properly atop of the Java 2 platform. For instance, CoDAC is implemented on the Grasshopper agent framework and tested to be operable on various platforms like Concordia and Mole.

The major benefit from the compatibility on heterogeneous mobile agent platforms is in the realization of collaboration among heterogeneous agents on top of various agent platforms. The DA adapter in CoDAC provides a common interface for heterogeneous agents to communicate and exchange messages atop platform independent Jini and JavaSpaces technologies. As a result, heterogeneous agents can interpret the messages from each other and take appropriate actions as long as they agree on a common communication protocol. The group of collaborating agents may be diverse not only in their contents but also in the forms their contents are realized. CoDAC provides a way to manage and coordinate multiple groups of heterogeneous agents. For example, agents developed and operating on Concordia and Grasshopper respectively can collaborate in either a single group or in separate local groups. On top of that, CoDAC offers the core functionality to manage the group membership, which enables each participant to collaborate seamlessly with one another as a whole.

Such heterogeneous collaboration may sound unfavorable yet unavoidable. For instance, various legacy systems like database and file systems are developed separately and distributed all over the open network. They are likely to be wrapped by heterogeneous resource agents. Hence, a collaborative information retrieval application may involve a set of heterogeneous user agents distributed over the variety of available platforms, where each agent is needed to interact with the appropriate resource agents in order to gain access to the underlying information. To take advantages of CoDAC's cross-platform compatibility, the dispatched user agents, heterogeneous in their implementation, can engage in the same collaboration group and can appear to be homogeneous.

10.2 Hierarchical Group Infrastructure

Complex large-scale collaboration can be simplified by decomposing into small subgroups. For instance, although each collaborating agent may have distinct

properties, some of which share some kind of similarity with one another, for examples, the roles they assume, the goals they pursue, the places they physically reside, etc. Based on these similarities, the collaborative agents can be categorized readily into a number of local groups that constitute a hierarchy as a whole. Agents within a local group perform a subtask with respect to the comprehensive work and different local groups collaborate to attain the global objective. Agents with common interests cooperate in local peer-to-peer collaboration, whereas agents with different interests negotiate via inter-group collaboration.

More precisely, lower level local groups typically compose of comparatively homogeneous and interchangeable agents that often work independently and pursue a common goal. The higher level local groups, on the other hand, behave more like an agent team [BMD99]. Each agent team comprises members (i.e. lower level local groups) that each has specific tasks or functions that requires more dynamic interchange of information, coordination and adjustment to demands.

This hierarchical infrastructure adds scalability and modularity to multiagent systems and the inhabiting agents respectively. On one hand, it decentralizes the coordination effort of the global coordinator through delegating to each local coordinator the obligation to manage the inhabitants in its local domain. In this sense, an agent can be added as a leaf to the hierarchy dynamically without affecting the configuration of the rest of the infrastructure. Scalability is particularly important to applications like information retrieval, data mining and network communities where the population of the agents is huge and highly dynamic. With supports from the CoDAC framework, agents can be plugged into the system dynamically where the necessary authorization, authentication and coordination works are essentially performed at the local domain, whereas the reconfiguration of the high-level groups is virtually nil.

On the other hand, within the hierarchical model a comprehensive task can be broken down into a set of component tasks to its lowest level. Each component task can be routine and general enough such that generic agents can be developed and reused to handle it. These generic agents can be plugged-and-played into different

applications and systems with negligible management burden. Furthermore, each agent can be upgraded independently without affecting the overall system. With the enhanced modularity in CoDAC, agents can experience the software component paradigm and feature as interchangeable building blocks for various applications and systems.

The infrastructure also brings benefits to communication within the hierarchy. The message traffic incurred in serving some coordination purpose is kept at the minimal extent as the underlying message exchange in pursuing a goal is restricted to circulate among the interested parties only (e.g. within a local group) wherever necessary. Instead of delivering each message to the entire collaboration group, the unwarranted message delivery to the unintended recipients can be eliminated readily. This saving is significant when the scale of the collaboration group is large.

10.3 Flexibility

CoDAC offers full flexibility in terms of both the language as well as the behavior of collaboration. As mentioned in section 8.2 the content of the messages swapping between the participating agents is not bound to any specific implementation in CoDAC. The underlying communication protocol among the participating agents can be based on various standard agent communication languages. For example, CoDAC is interoperable with a framework for KQML speaking software agents, the JKQML. This implies that agents can communicate with each other using KQML within the CoDAC framework. In general, any Java object that has implemented the java.io.Serializable interface can be used for communication purpose in CoDAC as long as the intended recipients implement a common interface and know how to interpret the collaboration language in the first place.

One the other hand, the hierarchical infrastructure in CoDAC is flexible in the sense that it can feature as the backbone for various communication models over the

Internet. For example, in the server group model [Adl95], a group of server agents can inhabit as a subtree in the CoDAC hierarchy with the so-called ServerGroup agent to operate as the local coordinator. The local coordinator decomposes the requested service into constituent tasks, and dispatches each of these tasks to the appropriate server agent within the group for concurrent execution. At the end, the coordinator collects and combines the results from those server agents into a single response for the client.

Further, the hierarchical relation is apparent in workflow systems [HS98b] which constitute of inter-related component tasks that share various control, data and temporal dependencies. Workflow agents can be readily categorized and mapped into local collaboration groups in CoDAC based on the component tasks they perform. These component tasks are typically strongly related to one another and are processed concurrently. The underlying workflow agents inhabit in separate local groups and coordinate as a whole to commit each workflow.

10.4 Atomicity

CoDAC embeds atomicity into the collaboration of mobile agents in distributed environments. It provides transactional support for enforcement of mutual consensus within the collaboration group. Any result reached in a collaboration can be delivered to all involved parties consistently to attain common knowledge and signal all group members to take consistent actions. This atomicity of agent collaboration is the key to meet the requirements in the electronic commerce environment.

10.5 Fault Tolerance

As described in section 7.1, once the default coordinator crashes, one of the agents will be elected as the new coordinator through the group membership protocol in the

CoDAC collaboration model. The newly elected coordinator will be responsible for resuming the collaboration in the primary backup approach.

Given a local collaboration group g with n participating agents, each individually has an availability p, the probability that exactly m out of these n agents are available can be calculated using the binomial probability function: $f(n, m)={}_nC_m p^m(1-p)^{n-m}$. As an agent in g needs to collect a majority of votes to be able to replace the failed coordinator, the overall availability $A_g(n, p)$ of g can be calculated as the probability that a majority of agents is available:

$$A_g(n,p) = \sum_{i=K}^{n} \binom{n}{i} p^i (1-p)^{n-i} \qquad (*) \qquad \text{where} \quad K = \left\lceil \frac{n+1}{2} \right\rceil$$

Table 8.1 and Figure 8.1 show the availability of g for various values of n in typical distributed systems having p's above 0.75. We can observe that $A_g(n, p)$ is generally improving for $n > 2$ and the improvement is more significant for odd values of n. For instance, $A_g(7, 0.75)$ is 0.93, which shows a 24% improvement with respect to p. We can conclude that the number of agents in each group should be an odd number bigger or equal to 3.

Further, the availability of the global group A_G as the overall hierarchy can be calculated from equation (*) by substituting p by the availability of the local groups (for simplicity, we assume the hierarchy is symmetric such that the $A_g(n, p)$'s are the same for all the local groups). Suppose in a simple scenario with a total of 9 agents divided into 3 local groups, each having 3 agents. Let p be 0.75, then we have:

$$A_g(3,0.75) = 0.84$$

$$A_G = A_g(3,0.84) = \sum_{i=2}^{3} \binom{3}{i} 0.84^i (1-0.84)^{3-i} = 0.93$$

and

n	p		
	p=0.75	p=0.85	p=0.95
1	0.75	0.85	0.95
2	0.56	0.72	0.90
3	0.84	0.94	0.99
4	0.74	0.89	0.99
5	0.90	0.97	~1

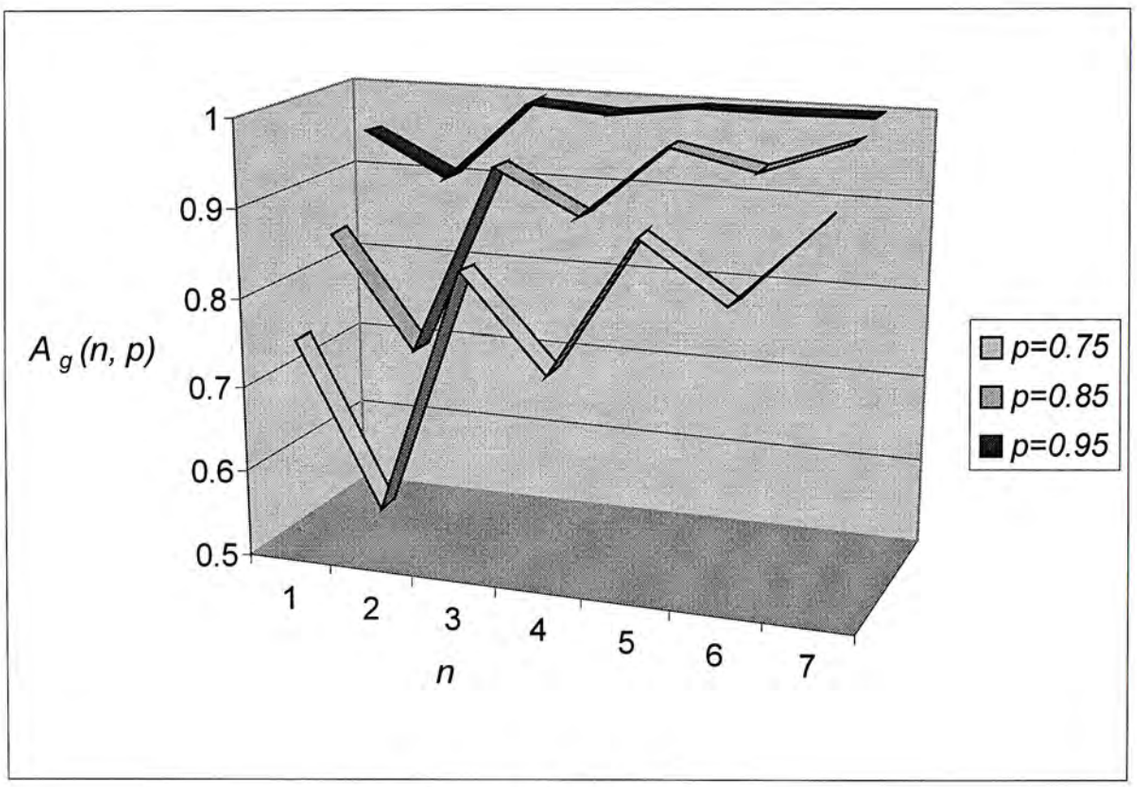

Figure 10.1: $A_g(n, p)$

Chapter 11

Conclusions and Future Work

11.1 Conclusion

We have introduced CoDAC as a comprehensive solution to general agent coordination problems. The major contribution of CoDAC is to embed atomic commitment capabilities into collaboration among distributed agents with enhanced fault tolerance. It delivers the core functionality to attain common knowledge within a collaboration group and signal all participants to take consistent actions. It fulfills the key proprieties in the component model to offer flexible and reliable coordination service to mobile agents distributed over the network with plug-and-play capability, encapsulated functionality and self-managing capacity. Beyond that, CoDAC breaks the gap between different agent platforms with its strong compatibility. Heterogeneous agents implemented and operating in different agent platforms can engage to the same collaboration group. In addition, the self-recovery in a collaboration group works fine regardless of the underlying heterogeneity. Furthermore, CoDAC presents a hierarchical group infrastructure which adds scalability to multiagent systems as the coordination effort decentralizes all over the

hierarchy where dynamic changes in the group membership can be handled effectively at the local domains.

11.2 Future Work

In order to exercise the atomic and fault tolerance features of CoDAC we suggest to have some practical experiences on electronic commerce and workflow management applications where transaction requirements must be fulfilled.

11.2.1 Electronic Commerce

First, let us consider a simple application for which a group of shopping agents roam through specific commercial sites to search for a stock of given items, says CPUs. At the beginning, the user can carry out a dialog with a system agent to state the set of constraints like the performance and the expected price of the desired CPUs, etc. The system agent then instantiates a shopping agent *c* as well as the kernel, which implements the logic to search for such item while enforcing the above constraints and to identify the best offer from a variety of vendors. Before launching *c*, the system agent first discovers the Jini lookup service and registers the kernel as a service proxy.

Upon arrival at a vendor, *c* first instantiates a DA adapter and registers a clone of this adapter to the lookup service such that other collaborating agents can access. Next, *c* downloads an instance of the kernel, and plugs it into its local instance of DA adapter in order to possess the capability to coordinate the group.

In the simplest case that all vendors share a common agent platform, *c* can start a collaboration group by duplicating itself and dispatching each of its clones to the rest of the interested sites. Otherwise, if the execution environments in different vendors are heterogeneous (e.g. some servers may be wrapped in Grasshopper agents whereas

the others are shielded by Concordia agents, etc), heterogeneous agents with different implementations must be launched explicitly by the user in order to interact with the appropriate vendors. In any case, as long as each shopping agent agrees on a common protocol and downloads an instance of DA adapter from the lookup service, it can engage into the collaboration group and collaborate seamlessly with other agents within the group.

The group is ready to collaborate after each shopping agent has settled on a specific host and obtained an instance of DA adapter. The local interaction between the agent and the vendor may be performed atop of message exchange or RPC, etc, depending on the platform implementation. Each shopping agent negotiates with the associated vendor over the prices and models of CPUs until *c* signals each agent to submit the offer it gets from the vendor. The DA adapter of *c* gathers the details of all available offers and delivers them to the kernel.

The kernel analyzes the gathered information, compares each offer against another and identifies the best deal that offers CPUs with the optimal performance while satisfying the price constraints. Having identified the vendor that offers the best deal, the coordinator defines the decision to commit the purchase from this specific vendor as the collaboration results and delivers it to every shopping agent within the group atomically.

In response to this collaboration result, each agent checks whether it is negotiating with that particular vendor. The agent residing on that specific vendor will commit the local transaction with that vendor to contract the deal. Whereas the rest of the participating agents abort any transaction they have been involved at their local hosts. This example will demonstrate how the CoDAC collaboration model is capable of collaborating heterogeneous agents and enforcing the exactly-once semantic on task accomplishment.

11.2.2 Workflow Management

Next, we consider a simplified order processing workflow as shown in Figure 9.1. In this example, each of the two component tasks, namely order verification and delivery scheduling, is performed by a local group. Each local group, in turn, collaborates with one another to complete the overall task

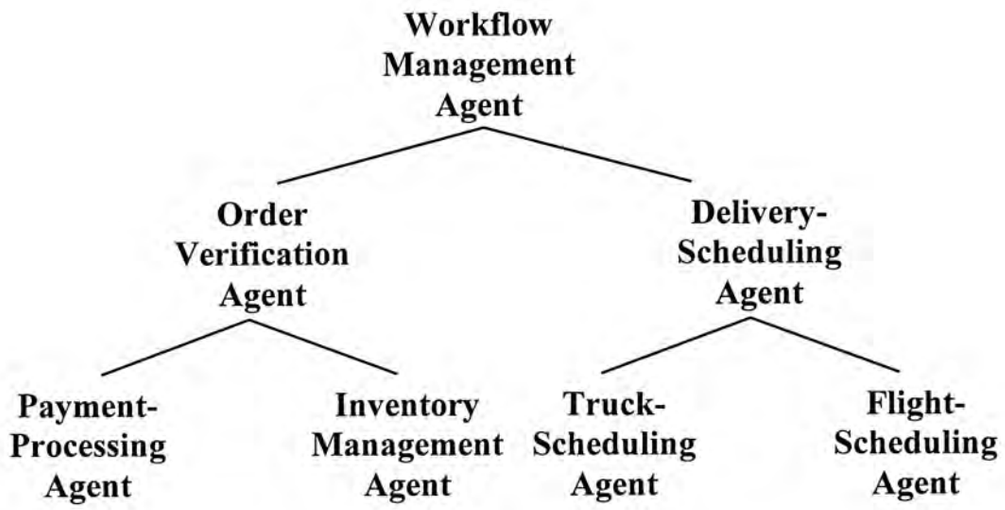

Figure 11.1: An order processing workflow system architecture

Suppose given a customer order with typical order information specifying the order quantity, the expected time of delivery along with the payment instruction. The order should be approved only if all the following requirements are met:

1. The inventory possesses all the requested stocks
2. The order will be delivered on time
3. The payment is authorized

As shown in Figure 9.1 the left and the right subtrees correspond to two local groups performing the order verification and delivery scheduling respectively. In the left subtree, the payment-processing agent interacts with the Payment Gateway to perform payment authorization whereas the inventory management agent locates the

requested items from the inventory. Both agents report to the local coordinator, the order verification agent, before making any permanent change to the database.

In the right subtree, suppose the order has to be delivered by air and then by truck, the local coordinator, delivery-scheduling agent, must coordinate the scheduling of flight and truck to meet certain constraints like the flight must be scheduled before the truck, the lag between the flight arrival and the truck available times should be short enough to eliminate warehousing cost, and above all, the delivery deadline must be met. The flight and truck-scheduling agents find out all possible flight and truck schedules that are likely to met the delivery deadline and let the local coordinator to identify the optimal schedule that meets all constrains.

Above all, both the two local groups are coordinated by the root coordinator, the workflow management agent. As long as the three key requirements are all met, the root coordinator approves the order and coordinates all workflow agents to commit globally. The order verification agent would commit the insertion of the new order entry to the customer order database. The payment-processing agent would commit the payment capture with the payment gateway. The inventory management agent would commit the change it made to the inventory database. The delivery-scheduling agent would commit the insertion of the new entry into the delivery schedule database. Should there be any of the key requirements failed to satisfy, The entire group of agents abort their operations and no change will be made to any of the databases involved.

This application will demonstrates the hierarchical infrastructure and exercises both local and global collaboration.

Bibliography

[Adl95] Adler, R.M., *Distributed coordination models for client/server computing*, IEEE Computer, Volume: 28, No. 4, April 1995, Page(s): 14 -22

[BHRS97] Joachim Baumann, Fritz Hohl, Kurt Rothermel, Markus Straßer, *Mole – Concepts of a Mobile Agent System*, http://www.informatik.uni-stuttgart.de/

[BMD99] Mark H. Burstein, Alice M. Mulvehill, and Stephen Deutsh, *An Approach to Mixed-Initiative Management of Heterogeneous Software Agent Teams*, Systems Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on, 1999, Page(s): 1-10 pp.

[CG89] Nicholas Carriero, and David Gelernter: *Linda in Context*, Communications of the ACM, April (1989), Volume 32, Number 4, 444 –458

[CR97] P. Ciancarini, D. Rossi, *Jada - Coordination and Communication for Java Agents*, Lecture Notes in Computer Science, Springer-Verlag (D), No. 1222, 1997, pp. 213-226.

[CTV+98] Ciancarini, P.; Tolksdorf, R.; Vitali, F.; Rossi, D.; Knoche, *A Coordinating multiagent applications on the WWW: a reference*

architecture, Software Engineering, IEEE Transactions on Volume: 24 5 , Page(s): 362 –375

[Edw99] W. Kenith Edwards, *Core JINI*, The Sun Mircosystems Press, Java Series, Prentice Hall, Inc, Sept 1999.

[Elm92] A. Elmasri (Ed.): *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, Inc. (1992)

[FG96] Stan Franklin, and Art Graesser, *Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents*, http://www.msci.memphis.edu/~franklin/AgentProg.html

[FLP85] Michael J. Fisher, Nancy A. Lynch, and Michael S. Paterson, *Impossibility of distributed consensus with one faulty process*, Journal of the ACM, 32(2) April (1985) 374-382

[HM90] Joseph Y. Halpern, and Yoram Moses, *Knowledge and Common Knowledge in a Distributed Environment,* Journal of the Association for Computing Machinery, Vol 37, No. 3, July 1990, pp549-587.

[HS98a] Huhns, M.N. and Singh, M.P. *All agents are not created equal,* IEEE Internet Computing, Volume: 2 3 , May-June 1998 , Page(s): 94 -96

[HS98b] Huhns, M.N. and Singh, M.P. *Workflow agents,* IEEE Internet Computing, Volume: 24, July-Aug. 1998 , Page(s): 94 –96

[IKV98] IKV++, *Grasshopper Technical Overview,* http://www.ikv.de/products/grashopper/

[KGN+97] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko, *Agent TCL: Targeting the Needs of Mobile Computers,* IEEE Internet Computing 1089-7801, 1997

[KT98] Neeran Karnik and Anand Tripathi, *Agent Server Architecture for the Ajanta Mobile-Agent System*, http://www.cs.umn.edu/Ajanta#papers

[Lewa98] Scott M. Lewandowski, *Frameworks for Component-Based Client/Server Computing*, ACM Computing Surveys, Vol. 30, No. 1, March 1998, pp.2-27,

[Maes95] Maes, Pattie, *Artificial Life Meets Entertainment: Life like Autonomous Agents,* Communications of the ACM, 38(1995), 11, 108-114

[OK97] Mitsuru Oshima and Guenter Karjoth, *Aglets Specification (1.0)*, http://www.trl.ibm.co.jp/spec_verson10.html

[OW94] Etzioni, Oren, and Daniel Weld (1994), A Softbot-Based Interface to the Internet. Communications of the ACM, 37, 7, 72-p;79.

[OZ98] A. Omicini, F. Zambonelli, *Co-ordination of mobile information agents in TuCSoN*, Internet Research: Electronic Networking Applications and Policy, Volume 8, Number 5, 1998, pp 400-413

[PS97] Holger Peine, and Torsten Stolpmann, *The Architecture of the Ara Platform for Mobile Agents*, Lecture Notes in Computer Science 1219, Mobile Agents, First International Workshop, MA '97, Berlin, Germany, April 1997, pp. 50-61,

[Rei94] Michael K. Reither, *A secure group membership protocol,* Research in Security and Privacy, 1994. Proceedings., 1994 IEEE Computer Society Symposium on , 1994 , Page(s): 176 -189

[RN95] Russell, Stuart J. and Peter Norvig, *Artificial Intelligence: A Modern Approach*, Englewood Cliffs, NJ: Prentice Hall (1995), page 33.

[RS97] Kurt Rothermel, and Markus Straßer, *A Protocol for Preserving the Exactly-Once Property of Mobile Agents,* http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/papers.html

[RS98] Kurt Rothermel, and Markus Straßer, *A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents.* Proc. 17th IEEE Symposium on Reliable Distributed Systems 1998 (SRDS'98), IEEE Computer Society, Los Alamitos, California, pp. 100-108.

[SBH96] Markus Straßer, Joachim Baumann, and Fritz Hohl, *Mole – A Java Based Mobile Agent System,* http://www.informatik.th-darmstadt.de/~fuenf/work/agent/projekte_e.html

[SC98] Suciu, O., Cristian, F.: *Evaluating the performance of group membership protocols*, Engineering of Complex Computer Systems, 1998. ICECCS '98, Proceedings. Fourth IEEE International Conference (1998) 13 –23

[SR00] M. Strasser, K. Rothermel, *System Mechanisms for Partial Rollback of Mobile Agent Execution.* In: Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000), IEEE Computer Society, Los Alamitos, California, pp. 20-28

[SRM98] Markus Straßer, Kurt Rothermel, Christian Maihöfer, *Providing Reliable Agents for Electronic Commerce*, Trends in distributed systems for electronic commerce: international IFIP/GI working conference, TREC'98, Hamburg, Germany, June 1998, pp.241-253,

[Sun99a] Sun Microsystems, *Jini™ Architecture Specification*, Version 1.1 Alpha, Nov (1999), http://www.sun.com/jini/

[Sun99b] Sun Microsystems, *Jini™ Lookup Service Specification*, Version 1.1 Alpha, Nov (1999), http://www.sun.com/jini/

[Sun99c] Sun Microsystems, *Jini™ Discovery and Join Specification*, Version 1.1 Alpha, Nov (1999), http://www.sun.com/jini/

[Sun99d] Sun Microsystems, *JavaSpaces™ Specification,* Version 1.1 Alpha, Nov (1999), http://www.sun.com/jini/

[Sun99e] Sun Microsystems, *Jini™ Transaction Specification*, Version 1.1 Alpha, Nov (1999), http://www.sun.com/jini/

[Szy97] Clemens Szyperski, *Component Software*, ACM Press Books, Addison-Wesley, 1997

[WJ95] Wooldridge, Michael and Nicholas R. Jennings, *"Agent Theories, Architectures, and Languages: a Survey,"* in Wooldridge and Jennings Eds., Intelligent Agents, Berlin: Springer-Verlag, (1995) 1-22
IEEE Vol. 1, No. 4: July-August 1997, pp. 58-67

[WPW+97] David Wong, Noemi Paciorek, Tom Walsh, Joe DiCelie, Mike Young, Bill Peet, Concordia: *An Infrastructure for Collaborating Mobile*

Agents, Lecture Notes in Computer Science 1219, Mobile Agents, First International Workshop, MA '97, Berlin, Germany, April 1997, pp. 86-97

[ZMG98] Michael Zapf, Helge Muller, Kurt Geihs, *Security Requirements for Mobile Agents in Electronic Markets*, Trends in distributed systems for electronic commerce: international IFIP/GI working conference, TREC'98, Hamburg, Germany, June 1998, pp.205-217

[Xop95] X/Open Company Lmited, *X/Open CAE Distributed Transaction Processing*, April (1995), http://www.siemens.com/servers/man/man_us/utm_man/xopen.htm

Publication List

1. "A Componentware for Distributed Agent Collaboration" to appear in Proceeding of the First International Workshop on Web Agent Systems and Applications (WASA-2000), published by IEEE Computer Society Press, 2000.
2. "A Micropayment System Based on Mobile Agents", to appear in Advances in E-commerce Agents: Broking, Negotiation, Security, and Mobility in LNAI series published by Springer-Verlag
3. "An Efficient Fault-Tolerant Protocol for Mobile Agents", in Intelligent Agent Technology edited by J. Liu and N. Zhong, pp. 441-445, World Scientific, 1999.