MILESTONE CHECKPOINTING IN MULTI-AGENT APPLICATIONS

Pingye Wang

A thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

October 2004

Your file Votre référence
ISBN: 0-494-04456-X
Our file Notre référence
ISBN: 0-494-04456-X

Abstract

Milestone Checkpointing in Multi-Agent Applications

Pingye Wang

Distributed multi-agent systems are useful in handling complex, realistic, and larger problems. However, they are susceptible to failures. A multi-agent system has some special characteristics such as sociality, which divides the system into different social groups so that intra-group coupling is much tighter than inter-group coupling. This thesis aims at developing a selective checkpoint based rollback-recovery protocol specially tailored for multi-agent systems, which combines selective checkpointing and logging technologies to achieve the objectives of simple and domino-free rollback and recovery, and fast output commit. A methodology is proposed to provide a step-by-step procedure for designers to follow in deciding where group checkpoints should be inserted. The methodology is demonstrated on an e-trading system implemented on the JADE agent platform. Experiments are conducted to show that the methodology can provide reasonable performance in multi-agent systems.

Acknowledgments

I would like to express my gratitude and respect to my supervisors Dr. Hon F. Li and Dr. Radhakrishnan Thiruvengadam for their invaluable guidance, encouragement and support during the whole period of my research work. Dr. Li and Dr. Radhakrishnan not only teach me knowledge at class, during meetings and through emails, they also try to enlighten me of how to do research, how to analyze and solve a problem.

I would also like to thank my colleagues Yu Zhang and Yu Li in the Distributed Systems research team for their kind support on my thesis.

Especially, I would like to give my special thanks to my parents and my husband whose patient love enabled me to complete this work.

Table of Contents

Lists of Figures

List of Tables

Table of Acronyms

CG	Checkpoint Group
CGS	Checkpoint Group Set
FIFA	the Foundation for Intelligent Physical Agents
ICGS	Intersection of Checkpoint Group Set
MASIF	Mobile Agent System Interoperability Facility
OMG	Object Management Group
OWP	Outside World Process
PWD	Piecewise Deterministic assumption

Chapter 1 Introduction

1.1 Motivation

Distributed multi-agent technology [Sycara98] represents a new paradigm for conceptualizing, designing and implementing software systems. It is used for complex, realistic, and large-scale problems. The strength of multi-agent systems lies in the fact that they are distributed and that agents communicate and cooperate in order to fulfill the application objective [Rana00]. However, as distributed systems, multi-agent systems are susceptible to the same faults that any distributed system is susceptible to, such as, process failures, communication link failures, or slow downs and software bugs. When a fault does occur in a multi-agent system, interactions between agents may cause the fault to propagate throughout the system and cause the entire system to fail. Thus, the issue of fault tolerance is a significant concern in the development of multi-agent systems. In the following part, we will first briefly review the existing fault tolerance techniques and agent techniques. Then, we describe the motivation of introducing existing fault tolerant techniques into the agent world.

In the literature, a large number of techniques have been developed for fault tolerance. Among them, checkpoint-based and log-based rollback-recovery protocols are two important techniques. For rollback-recovery protocols, a distributed system is treated as a collection of application processes that communicate through a network. The processes have access to a stable storage device that survives all tolerated failures. Processes achieve fault tolerance by using this device to save recovery information periodically during failure-free execution. Upon a failure, a failed process uses the saved information to restart the computation from an intermediate state, thereby reducing the amount of lost computation. The recovery information includes, at minimum, the states of the

participating processes, called checkpoints. Log-based recovery protocols may require additional information, such as logs of the interactions with input and output devices, events that occur to each process, and messages exchanged among the processes [Elnozahy02]. In a distributed system, the local state of each participating process is called a local checkpoint. A set of local checkpoints, one from each of the processes involved in a distributed computation, is called a consistent global checkpoint if the system state formed by the global checkpoint is one consistent global state that may occur during a failure-free, correct execution of a distributed system [Elnozahy02].

Checkpoint–based protocols rely solely on checkpointing for system state restoration. Checkpointing can be uncoordinated, communication induced, complete coordinated or selective coordinated. Log-based protocols combine checkpointing with logging of nondeterministic events, encoded in tuples called determinants [Alvisi98]. Depending on how determinants are logged, log-based protocols can be pessimistic, optimistic, or causal [Elnozahy02].

In uncoordinated checkpointing, processes take local checkpoints periodically without any coordination with each other. This approach allows maximum process autonomy for taking checkpoints and has no message overhead for local checkpoint. A process determines consistent global checkpoints by communicating with other processes to determine the dependency among local checkpoints. It could very well happen that processes took checkpoints such that none of the checkpoints lies on a consistent global checkpoint. A local checkpoint that cannot be part of a consistent global checkpoint is said to be useless, and a local checkpoint that can be part of a consistent global checkpoint is said to be a useful checkpoint. Upon a failure of one or more processes in a system, these dependencies may force some of the processes that did not fail to roll back, creating what is commonly called rollback propagation. This cascaded rollback may continue and eventually may lead to the domino effect [Wang93], which causes the

system to roll back to the beginning of the computation, in spite of all the saved checkpoints.

It is obviously desirable to avoid the domino effect and therefore several techniques have been developed to prevent it, such as communication-induced checkpointing and coordination checkpointing.

In communication-induced checkpointing, the number of useless checkpoints taken by processes can be reduced by forcing processes to take communication induced checkpoints in addition to checkpoints taken independently based on information piggybacked on the application messages received from other processes. Checkpoints taken by processes independently are called basic checkpoints, and the communication-induced checkpoints are called forced checkpoints [Manivannan99, Zambonelli98]. Communication-induced checkpointing algorithm can guarantee that a system-wide consistent state always exists on stable storage, thereby avoiding the domino effect and advancing the recovery line [Elnozahy02].

In complete coordinated checkpointing, processes synchronize their checkpointing activities so that a globally consistent set of checkpoints is always maintained in the system. The storage requirement for the checkpoints is minimum because each process needs to keep at most two checkpoints (one committed and one possibly not committed) in stable storage at any given time [Plank97]. The major disadvantages of coordinated checkpointing are that process execution may have to be suspended during the checkpointing coordination and it requires extra message overhead to synchronize the checkpoint activity [Manivannan99].

In selective coordinated checkpointing, a group of processes depending on each other, not all the processes in the system, participate in a coordinated checkpointing. Upon a failure of one or more processes in a system, only those processes that are dependent on each other need to rollback and the remaining processes unaffected can continue with their

computation. The disadvantages of selective coordinated checkpointing are that it has to keep track of dependencies, resolve the conflict when multiple checkpoint requests come to a process, and construct the recovery set after a failure [Elnozahy92]. Selective coordinated checkpointing is better when communication is sparse, while complete coordinated checkpointing is better when communication is frequent [Kalaiselvi00].

A fundamental goal of any rollback recovery protocol is to bring the system into a consistent state when the system fails. The reconstructed consistent state is not necessarily one that has occurred before the failure. It is sufficient that the reconstructed state be one that could have occurred before the failure in a failure-free, correct execution, provided that it should be consistent with the interactions that the system had with the outside world [Elnozahy02]. Checkpoint-based rollback-recovery protocols could not go beyond the reconstructed consistent state in recovery; however, log-based rollback-recovery protocols enable processes to replay their execution after a failure beyond the most recent checkpoint by using logging, which is useful when interactions with the outside world are frequent [Alvisi98].

A message-passing system often interacts with the outside world to receive input data or show the outcome of a computation. If a failure occurs, the outside world cannot be relied on for rolling back. For example, a printer cannot roll back the effects of printing a character, and an automatic teller machine cannot recover the money that it dispensed to a customer. To simplify the presentation of how rollback-recovery protocols interact with the outside world, we model the latter as a special process that interacts with the rest of the system through message passing. This special process cannot fail, and it cannot maintain state or participate in the recovery protocol. Furthermore, since this special process models irreversible effects in the outside world, it cannot roll back. We call this special process the "outside world process" (OWP) [Storm85].

Because of OWP, before sending a message (output) to outside world, the system must ensure that the state from which the message is sent will be recovered despite any future failure to maintain the consistent behavior of the system in recovery. This is commonly called the output commit problem [Storm85]. Similarly, input messages from OWP may not be producible during recovery, because it may not be possible for OWP to regenerate them. Existing methods to deal with output commit problem are checkpoint-based complete coordinated checkpointing mentioned above, and log-based rollback-recovery protocols. In coordinated checkpointing, before the system interacts with OWP, a consistent checkpoint has to be taken, resulting in considerable latency; however, log-based protocols can avoid taking expensive checkpoints before sending such messages.

Log-based rollback-recovery relies on the piecewise deterministic (PWD) assumption [Storm85], which postulates that all nondeterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in its determinant. Examples of nondeterministic events include receiving messages, receiving input from outside world, or undergoing an internal state transfer within a process based on some nondeterministic action such as the receipt of an interrupt. These protocols require that each process periodically records its local state and log the messages it received after that state. When a process crashes, a new process is created, given the appropriate recorded local state, and then the logged messages are sent to the newly created process in the order they were originally received. By logging and replaying the nondeterministic events in the exact original order, a process can deterministically recreate its pre-failure state even if this state has not been checkpointed. Log-based rollback-recovery in general enables a system to recover beyond the most recent set of consistent checkpoint. It is therefore, particularly attractive for applications that frequently interact with the outside world. Additionally log-based recovery generally is not susceptible to the domino effect. Logging a message may take time. Depending on

whether or not a process should wait for the logging to complete before delivering the message to the application, log-based protocols can be pessimistic, optimistic, or causal.

Pessimistic logging protocols [Johnson87] synchronously log recovery information on stable storage in order to simplify recovery. A failed process is restored to its state before the failure, and processes that survive the failure are not rolled back. In addition, no latency is incurred in sending messages to the outside world. Synchronous logging of recovery information however results in high failure-free overhead, unless special-purpose hardware is used. Optimistic logging protocols [Storm85] reduce failure-free overhead by logging recovery information asynchronously. Processes that survive a failure may however be rolled back. Furthermore, the latency of output commit is higher than in pessimistic message logging since a message cannot be sent to the outside world without multi-host coordination. This means that, in general, even optimistic message-logging protocols may block before sending a message to the environment. Causal logging protocols [Alvisi98, Elnozahy92] attempt to combine the advantages of low performance overhead of optimistic logging and fast output commit of pessimistic logging, but they require complex recovery algorithms. In addition, determinants are piggybacked on the application messages to track the causal dependency relationship.

In this thesis we want to focus on introducing traditional rollback-recovery protocols to multi-agent systems. We believe modification of these traditional protocols to adapt to the features of multi-agent systems can result in low overhead, limited rollback, and fast commit.

A software agent [Guessoum99, Lesperance02, Odell02], as the element of multi-agent systems, has no universally accepted definition. However, it is generally regarded as a computational entity that acts on behalf of users, and has the characteristics of autonomy, reactivity, pro-activeness, and sociality [Pivk, Odell02, Jennings01]. Autonomy is the ability of agent to perform a number of functions or activities without external

intervention. Reactivity is the ability to sense and react (stimulus-response) to external stimulus through simple actions. Pro-activeness is the ability to manage a set of behaviors to perform a mission. An agent can choose appropriate behaviors and can control behaviors to work concurrently and cooperatively to achieve its goal. Sociality is the ability to interact with humans or other agents through communication language in order to pursue its goal. A multi-agent system has four essential characteristics [Sycara98]: it is composed of autonomous software agents, it has no single point of control, it interacts with a dynamic environment, and the agents within it are social (agents communicate and interact with each other and may form relationships [Guessoum99]).

The sociality [Odell02, Lesperance02] is one most important feature of multi-agent systems. It consists of organizations, that groups of agents associated together by some common interest or purpose (such as mutual commitments, global commitments), a set of activities performed by the agents, a set of connections among agents, and a set of goals or evaluation criteria by which the combined activities of the agents are evaluated [Odell02]. An abstraction of the social aspects of an agent can be given as a role model [Miles01, Kendall99-1, Kendall99-2, Riehle98, Cabri01, Cabri02]. Roles are useful as they provide a way to describe a multi-agent system as analogous to an organization without placing heavy restrictions on the behavior of concrete agents at runtime. A role abstractly represents the goal, function, service, or identification of an agent within an organization. A role model describes roles and the relationship between roles. Relationship between roles can be defined by interaction protocols. Roles interact with each other through interaction protocols to pursue their common goal.

In a multi-agent system, there are roles whose responsibilities, permissions and protocols have to be fulfilled [Odell02]. As long as these roles in the society are fulfilled, the agent society can be regarded as stable and the design object of the system can be achieved. In agent-oriented programming, agents fulfill roles, and agents fulfilling a role must have

the ability to confirm to the obligation of the role. By mapping each agent to roles, an agent can fulfill one or more roles that define goals, tasks or functions of the agent. Therefore, agents communicate with protocols within the social group in which they participate to achieve their goals and the goals of the social group.

Because of the sociality of multi-agent systems, agents with common goals in a multi-agent system are divided into different social units, or groups. Each group provides a place for a limited number of agents to interact among themselves and support the interaction of those agents within the group [Odell02]. From the view of rollback recovery protocols, a social group is inherently a group for selective coordinated checkpointing. Complete coordinated checkpointing is better for a dedicated distributed system computing to carry out only one goal [Kasbekar01]. In such systems, restoring the task to a consistent state is equivalent to restoring the whole systems to a consistent state. However, in a multi-agent system containing multiple goals and numerous agents, it may be required to checkpoint or roll back the states of agents in a given goal. In such systems, complete coordinated checkpointing is too expensive for taking unnecessary local checkpoints, and upon a failure it may roll back unnecessary agents that are not involved.

Selective checkpointing is done groupwise, and unlikely processes, these groups may split or merge randomly at runtime [Kasbekar01]. Because of casual dependency among different groups caused by message exchange, recovery upon failures may cause domino effect if no other facilities are provided. Furthermore, a multi-agent system interacts within a dynamic environment that may involve frequent interaction with the outside world. Thus, we want to propose a rollback-recovery protocol on top of the selective protocols that solves the casual dependency among group checkpoints and output commit problems in multi-agent systems. We will investigate how isolated groups of agents can be checkpointed group-wise and how some agents can be rolled back selectively while

others in the system continue to execute. The goal of this thesis is to be able to identify, checkpoint, and roll back only agents in common social groups in a multi-agent system, and guarantee the consistency of the system state after a rollback.

1.2 Related Work in Agent Technologies

We have briefly reviewed the characteristics of agents and multi-agent systems. In this section, we will introduce the agent techniques proposed in the recent years. First, we will describe the existing role based analysis methodologies that facilitate the analysis and design of agent-based systems. Then, we will introduce agent-oriented methodology, which includes agent architecture and agent development framework. Agent architecture specifies the overall structure, logical components, and the logical interrelationship of an agent, and agent development framework provides an agent programming and execution environment.

1.2.1 Role Based Methodology for Agent Oriented Analysis and Design

Existing software development techniques, such as object-oriented analysis and design, are unsuitable for the task of supporting the development process of multi-agent systems, because they can not adequately capture an agent's flexible, autonomous problem-solving behavior, the richness of an agent's interactions, and the complexity of a multi-agent system's organizational structures. Therefore, in recent years, many agent-oriented methodologies and modeling techniques have been suggested to fulfill the task [Shehory01]. Many approaches build upon and extend existing methodologies and modeling techniques (for example object-oriented methodology, knowledge methodology, UML, and design patterns) to the design of agent systems [Wood00, Tveit01, Iglesias98]. For example, Burmeister's approach [Burmeister96] extends object-oriented methodology and suggests three models: agent model, organizational model, and

cooperation model for analyzing an agent system. Odell et al. suggest in "Extending UML for Agents" [Odell00] that further extension to UML called agent UML (AUML) can represent all aspects of agents. Others, such as agent-oriented methodology (AOM) [Wooldridge99] and GAIA [Wooldridge00] extended AOM, concentrate on the modeling aspect of agent-based system, and specially tailored to the analysis and design of agent-based systems. Among all these approaches, role modeling becomes more and more attractive for the ability of modeling the interactions in terms of protocols and the organizational structures of a multi-agent system.

1.2.1.1 Role Model Analysis Method in Agent-Based System

E A Kendall [Kendall99-1, Kendall99-2, Kendall00] presents a methodology for identifying, specifying, designing, and implementing agent systems on the basis of agent roles and role modeling. He indicates that role theory deals with collaboration and coordination, role models emphasize patterns of interaction, and as interaction and collaboration are essential aspects of any agent system, role modeling offers a promising approach for agent analysis and design. Role models could provide an abstraction that can unify diverse aspects of an agent system including acquaintance models, collaboration protocols, and task models. In object-oriented software engineering, classes stipulate the capabilities of individual objects, while role models emphasize how objects interact with each other. A role model comprised of roles and relationships between them describe a structure of interacting objects. A role defines a position and a set of responsibilities within a role model. An object role has responsibilities made up of services and tasks, and has collaborators that perform other related roles. In agent systems, objected-oriented role models can be extended to represent patterns of agent interactions. Agents are defined as extensions of objects that realize autonomous, proactive, social, reactive, and intelligent behaviors for the roles assigned to them.

Therefore, an agent role has more responsibilities than an object role. These include goals, obligations, and interdictions. An agent role involves co-ordination and negotiation with others, implemented typically in the form of interaction protocols. In agent system analysis, system goals should be partitioned, and assigned to individual roles. Any agent application will in fact encompass many specific instance role models that can be abstracted from scratch by analyzing the application requirements carefully. Often, a role model catalogue, which contains a set of commonly occurring agent role models and FIPA protocols, can be used to facilitate the analysis. In agent system design, the roles that a given agent needs to play have to be identified, and an agent is viewed as a set of roles. A class is required for every role. An Agent class can be designed by composing or assembling all of the responsibilities, interfaces, expertise, and protocols from the individual roles.

Role models of an application in agent-enhanced workflow are given as an example in figure 1-1 to elaborate the use of role model during agent system analysis and design. In figure 1-1, the relevant role models appear in the top half of the diagram, while the agents in the application that play the roles appear in the bottom half. A rounded box represents a role; an arrow between roles indicates collaboration between them; a filled circle signifies that more than one agent can play a role at the same time; a box indicates an agent; and a dashed line indicates assigning one role to an agent. In the diagram, role models include a supply chain role model and a bureaucracy role model. The supply chain is modeled at a high level as a predecessor role and a successor role, and the bureaucracy is comprised of a manager role and a subordinate role. Enterprise1 comprising a set of agents is a manufacturing company with a hierarchical structure of the bureaucracy role model, and aPlantManager agent inside it plays the roles of a successor in the supply chain and a manager in the bureaucracy management.

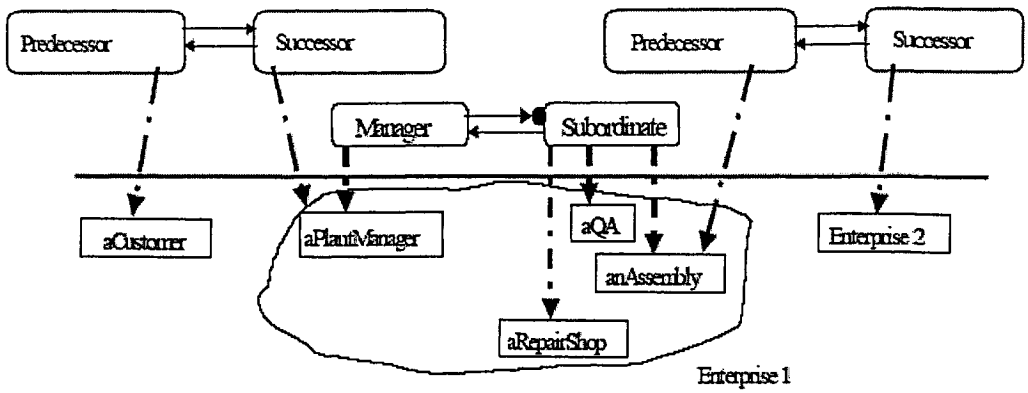


Figure 1-1: Role model for agent–enhanced workflow

1.2.1.2 Gaia Methodology for Agent-Oriented Analysis and Design

GAIA proposed by Wooldridge [Wooldridge00] is a general methodology for agent-oriented analysis and design that allows an analyst to go systematically from a statement of requirements to a design that is sufficiently detailed that it can be implemented directly. GAIA views an agent-based system as an artificial society or organization consisting of various interacting roles. Thus, in the analysis stage, the objective is to model the roles and their interactions, which results in a role model and an interaction model. The role model identifies the key roles in the system, and is comprised of a set of role schema, one for each role in the system. A role is viewed as an abstract description of an entity's expected function. The interaction model captures the dependencies and relationships between the various roles, and uses links between roles to present them. A set of protocols is defined in the interaction model, one for one type of inter-role interaction. A protocol is viewed as an institutionalized pattern of interaction formally defined and abstracted from any particular sequence for execution steps. In design stage, the aim is to transform the analysis models into a sufficiently low level of abstraction that traditional design techniques (including object-oriented) may be applied. The design process generates three models: an agent model, a service model, and an acquaintance model. The agent model identifies the agent types (an agent type is viewed as a set of

roles) that will make up the system, and the agent instances that will be instantiated from these types; the services model identifies the services with each agent role, and the services that an agent will perform are derived from the list of protocols and responsibilities associated with a role; the acquaintance model defines the communication links that exist between agent types. Gaia is concerned with how a society of agents cooperates to realize the system-level goals, and what is required of each individual agent in order to do this. It does not concern how an agent realizes its services, and leaves this problem to a particular application domain.

1.2.2 Agent Architecture and Agent Development Framework

Generally, agent oriented methodology includes agent architectures and agent development frameworks. Agent architecture is in the abstraction level of agent model, which specifies the overall structure, logical components, and the logical interrelationship of an agent. By defining the agent architecture, we find a way to explain and predict agent behaviors, and to support the design of agents and their interaction in an implemented system. Agent development framework is in the application programming level, which supports building practical agent applications from the perspective of software design. The objectives of agent frameworks are to provide a rapid prototyping development environment for the systematic construction and deployment of agent oriented applications and to encourage code reuse and standardization of agents.

1.2.2.1 Agent Architecture

Many agent architectures have been proposed in the literature. However, there is no standard agent architecture due to lack of standard definition of an agent. In an agent-based application, the individual researcher should select the agent architecture depending on the requirements analysis, and customize it. Most of the existing agent

architectures belong to three categories: deliberate architecture, reactive architecture, and hybrid architecture.

The deliberative architecture [Rao91] is based on the physical-symbol system hypothesis, which assumes that the world is deterministic, accessible, static, symbolic, and its operators can be fully specified. This architecture is highly influenced by the traditional AI research. It uses a symbolic model to explicitly represent the world, and make decisions through logical reasoning based on pattern matching and symbolic manipulation. However, the architecture does not perform very well for the difficulty in modeling of the world and relying too much on inadequate sensors. Deliberative agents are slow, inherently sequential, require large memories, are not scalable, and are not useful in a dynamic and noisy environment. The BDI (belief-desire-intension) architecture proposed by Rao & Georgeff in 1991 [Rao91] is the most popular deliberative architecture. A BDI agent contains a set of beliefs that represent what the agent "knows", a set of desires/goals that represent what the agent is trying to achieve, a set of intentions to achieve the agent's current goals, and a set of plans that are combinations of actions that achieve certain outcomes or respond to events and are used by the agent to further its intentions. When an event occurs, the agent looks for relevant plans responding to this type of event, examines each plan with its appropriateness to its current situation, and finally selects and starts executing the most appropriate plan found. Additionally the agent performs ongoing reasoning functions to decide what goal to pursue or alternatively what event to react to, how to pursue the desired goal, and when to suspend/abandon the goal, or change to another goal.

The reactive architecture [Brooks86] is the opposite of the deliberative architecture, which is based on purely reactive behaviors under the assumption that the world is complex and non-symbolic. Reactive agents have little or no knowledge of the world, and are constructed in a way that allows them to react to a changing environment by their

"instincts". The reactive architecture is inherently parallel, fast, operates on short time scales and is scalable; however, the purely reactive scheme does not perform well when performing complex tasks like planning and goal reaching. The best-known reactive architecture is the "Subsumption Architecture" proposed by Brooks in 1986 [Brooks86]. The architecture proposes a layered design of competing task-accomplishing behaviors. Lower layers exhibit more primitive kinds of behavior, and have precedence over layers further up the hierarchy.

The hybrid architecture [Muller97, Fischer97] combines the deliberative and reactive architectures to have the advantages of both of them; therefore, it combines AI components and reactive elements into one design model. However, the disadvantage is that it is hard to design and it is not scalable. A well-known example is INTERRAP [Muller97]. INTERRAP consists of three layers that serve different purposes. Behavior based layer implements the reactive behavior of the agent, it reacts very fast to external requirements without any explicit reasoning. Local planning layer performs the planning process of an individual agent to achieve its goals, it is also responsible to monitor the plan execution of the agents current plan. Social planning layer is responsible for the coordination with the other agents in a multiagent system. The coordination with the other agents is achieved with explicit negotiation protocols. When an agent perceives information from the environment, if the event is just for reactive activity, the agent can directly handle it without planning. If the reactive activity cannot handle it, the local planner layer will be activated. Moreover, when the local planning layer cannot solve the problem, the control will be passed to the higher layer, the social planning layer, which will coordinate with other agents.

1.2.2.2 Agent Development Framework

Agent frameworks provide an agent programming and execution environment, which

may be seen as a middleware residing between the underlying host/network operating system and the application layer. Many agent frameworks have been proposed, they may focus on different issues, for example, MOLE [Straßer97] mobile agent system focuses on offering agent mobility, and RETSINA [Sycara96] offers reusable agents to build applications. Some existing frameworks put agent architectures as the basis of development for agent based systems, such as ZEUS [Nwana99, Zeus], which specifically defined their own agent architectures. Some may not provide such a definition. However, they all have an agent behavior engine, communication interfaces, and corresponding primitive processing objects. In order to enable interoperability of agents in different frameworks, the standardization of multi-agent technology is needed. The Mobile Agent System Interoperability Facility (MASIF) [MASIF] by Object Management Group (OMG) and the specifications promulgated by the Foundation for Intelligent Physical Agents (FIFA) [FIFA] are two standards highly accepted. In the following paragraph, we introduce several frameworks, and choose one as the programming environment we use in this thesis.

ZEUS [Nwana99, Zeus] is an FIPA compliant tool-kit for engineering distributed multi-agent systems. It allows the rapid development of Java agent systems by providing a library of agent components, by supporting a visual environment for capturing user specifications, an agent building environment that includes an automatic agent code generator and a collection of classes that form the building blocks of individual agents. ZEUS agents are composed of five layers: API layer, definition layer, organizational layer, coordination layer and communication layer. The API layer allows the interaction with non-agent world; the definition layer manges the task the agent must perform; the organizational layer manages the knowledge about the other agents; the coordination layer manages coordination and negotiation with other agents; and the communication layer allows the communication with other agents. The agent structure transfers

information from layer to layer, so any higher behavior depends on lower level capabilities. This architecture is actually a sequentially executing mechanism, and does not support concurrent activities. Moreover, ZEUS doesn't support agent mobility.

Grasshopper [Grasshopper] is a highly reliable and extensible pure Java-based mobile agent platform, providing all necessary functional capabilities to develop and run agent applications. It is composed of regions, places, agencies and different types of agents. An agency is defined as a runtime environment providing the functionality for the execution of an agent and various agent services. A place is a virtual location within an agency at which agents may be located for a particular service, e.g., exchange of information or trading. A region is a grouping of several agencies connected for a common purpose, which offers location transparency to their agents. A region registry provides comprehensive look-up functionality to locate agencies, places, services, and agents in the scope of the region. The registration and de-registration of agencies, places, services and agents is performed automatically. Grasshopper supports mobile agent based telecommunications applications, and also supports persistence that agents can be saved periodically, and in case of unexpected system failures, the agents can be recovered. Grasshopper conforms to the OMG MASIF standard for MA platform interoperability, and also supports FIPA agent communications among Grasshopper agents.

JADE [Jade, Poggi00] is Java Agent Development Framework for developing multi-agent systems and applications. It simplifies the implementation of multi-agent systems through a middle-ware that complies with the FIPA specifications and through a set of graphical tools that supports the debugging and deployment phases. JADE includes two main products: a FIPA-compliant agent platform and a package to develop Java agents. It includes an agent foundation class for writing customized agents, a library of protocol skeletons for tailoring agent conversation and a suite of development tools. The tools provide runtime agent management, directory facilitation monitoring and editing,

message exchange debugging, agent life-cycle control, and a conversation monitoring tool that draw a sequence diagram of agent interaction. JADE agents are implemented as one thread per agent, but they can have several behaviors that may execute cooperatively and concurrently. JADE agents achieve this with co-operative behavior scheduling, which schedules behaviors in a light and effective way. A JADE agent can send/receive Java objects that represent FIPA-ACL messages within the scope of interaction protocols. After reviewing several agent frameworks mentioned above, we decide to choose JADE as our agent-programming environment. The reason includes: first, JADE supports agent mobility and JADE agent can run in dynamic environment; second, the cooperative and parallel agent behaviors help implementing the role mapping, since agent roles can be mapped to JADE behaviors if a role modeling approach is used; third, JADE is a free and open source software, which makes it possible to change the source code when needed. ZEUS and Grasshopper are not chosen because ZEUS does not support current activities and agent mobility, while Grasshopper is not open source software.

1.3 Related Work in Rollback-Recovery in Message-Passing Systems

We have briefly overviewed the rollback-recovery protocols in the motivation subsection. However, as our objective is to propose a new selective-checkpoint based protocol, specially tailored for multi-agent systems, deeper insight into the existing rollback-recovery protocols would help us to understand and do the work more properly. In the following part, we will talk about the Chandy & Lamport's algorithm [Chandy85], which is the foundation of the checkpoint-based protocols, a selective coordinated checkpointing protocol for demonstration of the selection of checkpoint groups, an uncoordinated Checkpointing to show how to build the recovery line. Finally, several log-based protocols are discussed.

1.3.1 Chandy & Lamport's Algorithm

In the literature, many checkpointing algorithms have been proposed for message passing systems; however, most of them use Chandy & Lamport's algorithm [Chandy85] proposed in 1985 as the base, and modify the algorithm to meet their assumptions. The system model proposed by them is widely accepted and has become the basis of the rollback-recovery protocols.

Chandy & Lamport [Chandy85] models a distributed system as a finite set of processes and a finite set of channels. Processors communicate with each other by exchanging messages through channels. Channels are assumed to have infinite buffers, to be error-free, and to deliver messages in the order sent (FIFO). The communication delay of a message in a channel is arbitrary but finite. The state of a channel is the sequence of messages sent along the channel, excluding the messages received along the channel. A process is defined by a set of states, an initial state, and a set of events. An event in a process is an atomic action that may change the state of the process itself and the states of at most one channel incident on the process such as sending or receipt of a message. A global state of a distributed system is a set of component process and channel states. The occurrence of an event may change the global state.

Chandy & Lamport's algorithm is used for recording the global state of the system. To record a correct global state, all the processors in the system have to coordinate currently to log the channel states at the time of checkpointing. Special messages called "markers" that have no effect on the underlying computation are used for coordination and for identifying the messages originating at different checkpoints intervals. The algorithm starts by a centralized process P_0 through sending itself a "marker". Let P_f be the process from which process P_i receives the "marker" for the first time. Upon receiving the "marker", P_i records its local state and relays the "marker" along all its outgoing channels (no intervening events on behalf of the underlying computation are executed between

these steps). The state of the channel from P_f to P_i is set to empty and P_i starts recording messages received over each of its other incoming channels. Let P_s be another process from which P_i receives the "marker" beyond the first time. Process P_i stops recording messages along the channel from P_s and declares the channel state as those messages that have been recorded. A process ends its participation in the algorithm once it receives a "marker" message from all its incoming channels. From the view of checkpoint and recovery, the global state recorded by the Chandy & Lamport's algorithm is a consistent global checkpoint that forms a recovery line in the occurrence of failures.

1.3.2 Selective Coordinated Checkpointing

Chandy & Lamport's algorithm requires all processes to participate in every checkpoint. However, for large scalable systems, it is desirable to reduce the number of processes involved in a coordinated checkpointing session. Koo and Toueg [Koo87] proposed that only processes that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint need to take a new checkpoint without breaking the consistency of the global checkpoint.

A consistent global checkpoint forms a snapshot of a consistent system state, and a consistent global state is one global state that may occur during a failure-free, correct execution of a distributed system, which means that if the state of a process reflects a message receipt, the state of the corresponding sender process has to reflect sending that message. The notion of a consistent global state is central to a rollback-recovery protocol in that it forms a safe recovery line once a failure occurs [Elnozahy02].

Koo and Toueg use a two-phase protocol to achieve the minimal checkpoint coordination. In the first phase, the checkpoint initiator identifiers all the processes with which it has communicated since the last checkpoint and sends them a checkpoint request. Each process, upon receiving the request, in turn identifies all processes it has

communicated with since its last checkpoints and sends them a request, and so on, until no more processes can be identified. In the second phase, all processes identified in the first phase take a checkpoint. After the checkpointing protocol, the global checkpoint in the system can still maintain consistency because there is no message exchange between processes taking new checkpoints and processes not taking the new checkpoints,

1.3.3 Uncoordinated Checkpointing

Coordinated checkponting algorithms can guarantee the consistency of the recorded global state and avoid domino effects. However, coordinated checkpointing requires additional control messages to do the coordination and is not applicable when processes need to checkpoint independently of each other. Uncoordinated checkpointing is proposed to have the advantage that each process may take a checkpoint when it is most convenient or appropriate to do so according to its local decision. This leads to asynchronous checkpoints that can be taken without inter-process protocols. However, uncoordinated checkpointing has several disadvantages: the domino effect may happen, useless checkpoints may be taken, one process may have to maintain multiple checkpoints, and garbage collection algorithms may be invoked periodically. The algorithm proposed by Bhargava and Lian in 1988 [Bhargava88] is a typical uncoordinated checkpointing protocol that uses a rollback-dependency graph to determine the recovery line once a failure occurs.

Because no consistent global checkpoint is formed during checkponting, once a failure occurs, the system has to determine one during recovery. For this purpose, dependencies among the checkpoints of process during failure-free operation have to be recorded. Bhargava and Lian use the following technique to get the recovery line. For a process P_i, use $C_{i,x}$ to identify the x-th checkpoint of process P_i, use $I_{i,x}$ to identify the checkpoint interval between checkpoints $C_{i,x-1}$ and $C_{i,x}$. If process P_i at interval $I_{i,x}$ sends a message to

P_j, it will piggyback the pair (i, x) on m. When P_j receives m during interval $I_{j,y}$, it records the dependency from $I_{i,x}$ to $I_{j,y}$, which is later saved onto stable storage when P_j takes checkpoint $C_{j,y}$. If a failure occurs, the recovering process initiates rollback by broadcasting a dependency request message to collect all the dependency information maintained by each process. Upon receiving the request message, a process stops its execution and replies with its dependency information. The initiator then can form one rollback-dependency graph based on the information collected. In the rollback-dependency graph, each node represents a checkpoint, and a directed edge is drawn from $C_{i,x}$ to $C_{j,y}$ if either: $i \neq j$, and a message m is sent from $I_{i,x}$ and received in $I_{j,y}$; or $i = y$ and $y = x + 1$. An example is given in figure 1-2, where figure (b) shows the rollback-dependency graph for the execution in figure (a).

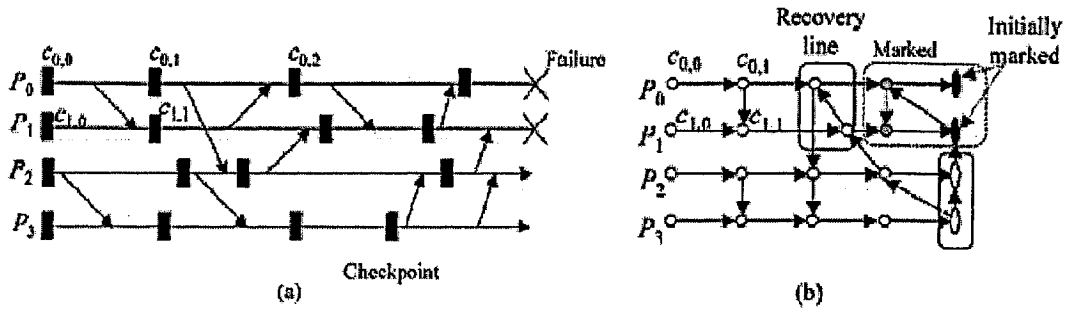


Figure 1-2: (a) Example execution; (b) Rollback-dependency graph.

To compute the recovery line, we still use figure 1-2 as the example. The algorithm first marks the graph nodes corresponding to the states of process P_0 and P_1at the failure point (shown in figure in dark ellipses). Then it uses reachability analysis to mark all reachable nodes from any of the initially marked nodes. The union of the last unmarked nodes over the entire system forms the recovery line.

For an uncoordinated checkpointing algorithm, the main disadvantage is the domino effect. We use an example as shown in figure 1-3 to elaborate.

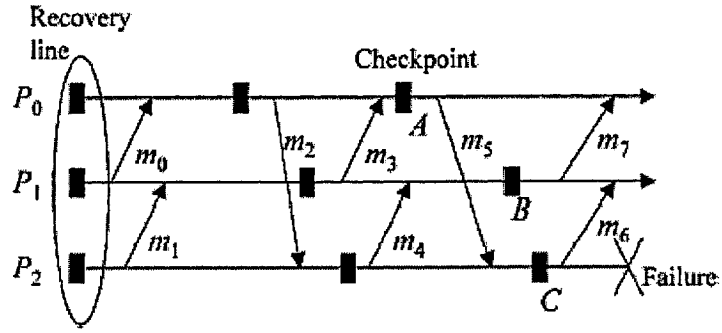


Figure 1-3: Rollback propagation, recovery line and the domino effect

Figure 1-3 shows an execution in which processes take their checkpoints. Horizontal lines represent time axes of processes; arrows represent messages from the sending process to the receiving process; and black bars are checkpoints taken by processes. Suppose process P_2 fails and rolls back to checkpoint C. Because the rollback invalidates the sending of message m_6, P_1 must roll back to checkpoint B to invalidate the receipt of that message, which in turn invalidates message m_7 and forces P_0 to roll back as well. This rollback propagation can continue and eventually cause the system to roll back to the beginning of the computation, in spite of all the saved checkpoints. This situation is called the domino effect.

1.3.4 Log-Based Rollback-Recovery Protocols

Uncoordinated checkpointing has the disadvantage of the domino effect. One solution to avoid it while keeping independent checkpointing is to use communication-induced checkpointing. Another solution is to use the message logging techniques that lead to log-based rollback-recovery [Elnozahy02]. Log-based rollback-recovery avoids the domino effect by enabling a process to replay its execution after a failure beyond its most recent checkpoint till its pre-failure state [Johnson87, Alvisi98, Elnozahy92]. This is attractive to applications with frequent interactions with the outside world. However, log-based recovery relies on the piecewise deterministic assumption (PWD) and can only tolerate

process crash failure, while checkpoint-based protocols have no such restriction [Elnozahy02].

Alvisi proposed in 1996 that all message-logging protocols require satisfying the "no orphan processes" [Alvisi96] consistency property when recovery is complete. Orphan processes are surviving processes whose states are inconsistent with the recovered state of a crashed process. This requirement guarantees that upon recovery, no process is an orphan and guarantees the consistency of the recovered system. Alvisi defined the property with a formal condition [Alvisi96]: $\forall e$: $\neg Stable(e) \Rightarrow Depend(e) \subseteq Log(e)$. Here, e is a nondeterministic event that occurs at process p. *Depend(e)* is the set of processes that are affected by e, which consists of p and any process whose state depends on the event e according to Lamport's "happened before" relationship [Chandy85]. *Log(e)* is the set of processes that have logged a copy of e's determinant in their volatile memory. *Stable(e)* is a predicate that it true if e's determinant is logged on stable storage. We call it the "always-no-orphan" condition. Satisfying the condition means that for any surviving process p that depends on an event e, either the event e is logged on stable storage, or the process p has a copy of the determinant of event e. If neither condition is true, then the process p becomes an orphan because it depends on an event e that cannot be generated during recovery since its determinant has been lost.

Pessimistic logging protocols implement a strong property of the always-no-orphan condition: $\forall e$: $\neg Stable(e) \Rightarrow Depend(e) = 0$. It stipulates that if an event has not been logged on stable storage, then no process can depend on it. Under this condition, a process delivering a message is not allowed to send any messages until the determinant of the message is stable, so that the observable state of each process is always recoverable because failure processes will exactly repeat their pre-failure execution and re-send the same message. The pessimistic logging has the advantage of no latency in interactions with outside world, simple recovery, and simple garbage collection. However, pessimistic

logging has a performance penalty incurred by synchronous logging. In order to reduce performance overhead, several techniques have been proposed such as the Sender-Based Message Logging protocol (SMBL) proposed by Johnson and Zwaenepoel [Johnson87]. SMBL reduces the performance overhead by keeping the determinants corresponding to the delivery of a message *m* in the volatile memory of its sender. The determinant of the message *m* consists of its data and the order in which it was delivered, including the identification of its receiver process, the sender sequence number of the sender process, and the receiver sequence number of the receiver process. The determinant is logged in two steps. First, before sending *m*, the sender logs its data in volatile memory. Then when the receiver of *m* responds with an acknowledgement that includes the order in which the message was delivered, the sender adds the ordering information to the determinant. SBML avoids the overhead of accessing stable storage but tolerates only one failure and cannot handle nondeterministic events internal to a process.

Optimistic logging protocols [Johnson91, Strom85] reduce the performance overhead by using asynchronously logging. For example, Strom and Yemini [Strom85] proposed to keep determinants in a volatile memory, and periodically flush them to stable storage. However, optimistic protocols do not implement the always-no-orphan condition and therefore permit the temporary creation of orphan processes. In recovery, it has to satisfy the property by rolling back orphan processes until their states do not depend on any message whose determinant has been lost. To perform the rollback correctly, optimistic logging protocols track causal dependencies during failure-free execution. Upon a failure, the dependency information is used to calculate and recover the latest global state of the pre-failure execution in which no process is in an orphan. Optimistic logging needs a complicated garbage collection algorithm because each process has to keep multiple checkpoints. Another disadvantage is that an output commit will generally require multi-host coordination to ensure that no failure scenario can revoke the output.

Causal logging protocols [Elnozahy92, Alvisi96, Elnozahy94] combine the advantages of both optimistic logging and pessimistic logging, but they require more complex recovery algorithms. Causal logging protocols implement the "always-no-orphan" property by ensuring that the determinant of each nondeterministic event that causally precedes the state of a process is either stable or available to that process, for example, the Manetho [Elnozahy92] system proposed by Elnozahy saves these determinants in the volatile log of the surviving processes. Each process in Manetho maintains in volatile memory an antecedence graph (AG) of its current state, and a log that maintains the data and identifier of each message it sends. An antecedence graph has a node representing each nondeterministic event preceding the state of a process, and the edges corresponding to the happened-before relationship among events. When a process sends a message, it piggybacks the AG on the message to propagate the causal information. The receiver, upon receiving the message, adds the receipt order indicated in the AG to its volatile log. By doing this, each process maintains one antecedence graph providing a complete history of the nondeterministic events that have casual effects on its state. Periodically, a process records its checkpoint, the volatile log and the AG on stable storage. During recovery, the failed process restores its checkpoint, message log, and the saved AG from stable storage. It calculates its AG of the pre-failure state by merging the AGs collecting from the surviving processes. Then, the recovery process replays its messages by requesting them from the senders' log and re-executes them.

1.4 Contributions and Outline

From the review of the existing rollback-recovery protocols, we find that all of them are designed for general distributed systems without additional assumptions. As a multi-agent system holds some special characteristics such as sociality and being goal-driven,

rollback-recovery protocols considering these new characteristics may work more efficiently than the traditional protocols.

In this thesis, we propose a new rollback-recovery protocol (strategy) that combines the selective coordinated checkpointing protocol and pessimistic log-based protocol. The selective checkpointing has the advantage of lower performance overhead; while the pessimistic log-based protocol has the advantage of no domino effect, fast output commit, and simple garbage collection. Our new protocol has the combined advantage of both of them. Compared with selective coordination checkpointing, our protocol avoids the dependency tracking. Compared with log-based protocol, our protocol records fewer messages.

As we have explained in the motivation part, sociality is the main characteristic of multi-agent systems, and related set of cooperated agents in the system forms different social groups to achieve their individual goals or/and common goals. Since agents in the same social group cooperate closely with each other and seldom interact with others outside the social group, the social group inherently becomes a group for the selective checkpointing. Based on the assumption that the knowledge of the agent system is clear, we propose a methodology to identify the social groups of the system. The proposed methodology describes how to analyze the role model built in the analysis stage of the system engineering in order to get the social groups. In addition, the methodology also identifies the suitable place for starting a checkpoint. The theory relies on the observation that the designer's static knowledge of the application domain can be used effectively to select source statements at which checkpoints should be inserted [Silva98, Huang93].

In the rest of the thesis, we will introduce the methodology for identifying a checkpoint group and checkpoint places in chapter 2, elaborate the new rollback-recovery protocol in chapter 3, use an application to demonstrate the methodology and the protocol in chapter 4 and 5, test the system performance in chapter 6, and draw the conclusion in chapter 7.

Chapter 2 Role-Milestone Based Methodology

2.1 Introduction

In a multi-agent system, related agents form different social groups to achieve their individual or/and common goals. Such a social group, from the perspective of checkpointing, is inherently a group for a selective checkpoint. In this chapter, based on the role model proposed by E. A. Kendall [Kendall00], we propose a role-milestone methodology to identify where checkpoints should be activated. The methodology requires knowledge of the agent system, and is applied at the design stage in order to build a fault tolerant multi-agent system. Hence it is a user-level fault tolerant solution, unlike the more common user-transparent solution. Input for this methodology is the role models designed in the agent system. The output is the social groups of agents, and the suitable places for inserting checkpoints that can be translated to the source codes in the later implementation stage. The methodology uses a set of milestone dependency graphs, obtained by analyzing the role models, to identify social groups and checkpoint placements. Each milestone graph is a labeled directed acyclic graph, which involves a set of related roles, and their milestones depending on each other to achieve their common goals. The agents playing these related roles associated with a milestone dependency graph form a social group in the runtime of the system. Some milestones satisfying pre-defined criteria can be selected as good places for inserting checkpoints. The analysis result is used to facilitate the selective rollback-recovery protocol design which will be presented in chapter 3.

The role-milestone based methodology is derived from the role model analysis agent methodology of E. A. Kendall. We extend the role model methodology to help building a robust agent system that has better performance and is easier to build. The extension

includes a role-based agent architecture that supports the ability that one agent plays multiple roles concurrently, and the mapping strategy between roles and agents that helps to increase the run time performance of the agent system. In the following parts of the chapter, we will introduce the agent architecture, mapping strategy, and the role-milestone based methodology one by one.

2.2 Role-Based Agent Architecture

Most of existing agent architectures do not provide support for multiple role-playing parts. However, as role modeling becomes more and more important, an agent architecture providing the facility for multiple role-playing becomes urgently needed. In this thesis, we propose the role-based agent architecture building upon the theory that roles are the basic elements of an agent. A role here has its architecture called the role architecture, which describes the logical components of a role. The agent architecture building upon roles is then as simple as the composition of the element roles. The role-based agent architecture is quite straightforward, and it simplifies the design of agents to design of roles. Moreover, the role architecture is not only more specific and accurate, but also enhances the component reuse as different agents playing the same role can share the same role component. The role-based architecture fully presented in this research meets the requirement that multiple roles could be mapped to one single agent according to the role model agent analysis of E. A. Kendall.

2.2.1 Role Architecture

Roles are described as basic elements of agents that have the ability to accomplish their responsibilities in our approach. Thus, we borrow some ideas from existing agent architectures to define the role architecture.

The role architecture consists of one knowledge part, one sensory part, one reactive part, and one proactive part as show in figure 2-1. The knowledge part stores the knowledge

including the local data of the role; the sensory part senses the environment changes and updates its knowledge periodically; the reactive part includes reactive behaviors that respond to stimulus without reasoning; the proactive part makes the decisions and exhibits goal-directed behaviors based on its knowledge, current state, and interaction with other roles to accomplish its goals.

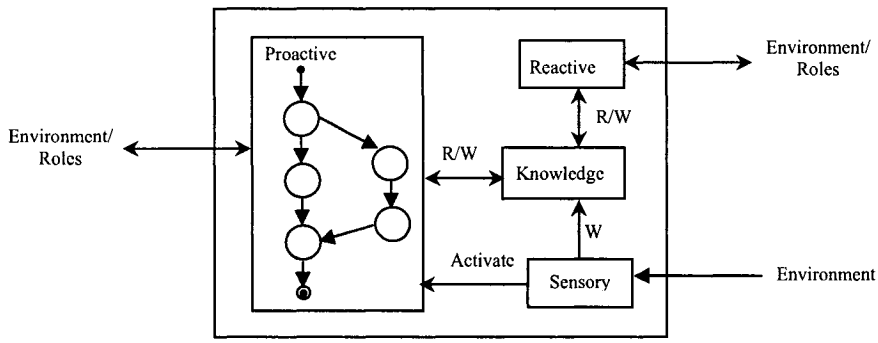


Figure 2-1: Role structure

The proactive part can be described as an activity diagram composed of different activities and transactions. Each activity presents one behavior implementing a sub-task of the role, which may read/write knowledge, interact with other roles, and change environment. A transaction might be a decision-making based on knowledge and current state, or delivery of a message from other roles. The proactive part has its initial state indicating the start of achieving its goals, end state indicating end of trying to achieve the goals. More than one end state might exist.

2.2.2 Agent Architecture

One agent may play multiple roles sequentially or concurrently, so one agent can be seen as the composition of multiple roles. During the composition, the common parts of different roles should be integrated together and shared among them in order to lower the resource cost and avoid the consistency mechanism. Based on the above-mentioned integration strategy, the agent architecture composed by different roles is shown in figure 2-2.

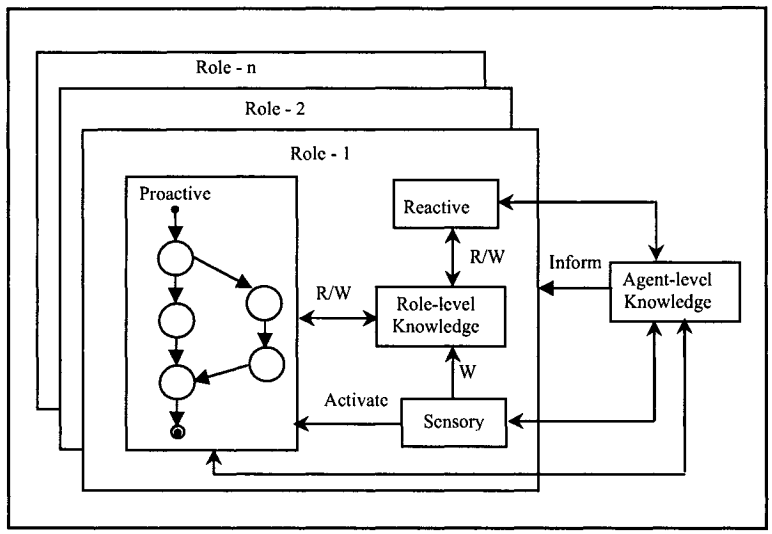


Figure 2-2: Agent structure

Each role in the agent architecture is a separate part; however, the full knowledge or part of the knowledge from different roles may be common and can be shared. The common knowledge is taken out from the role component and forms the agent-level knowledge, which can be read/written by the sensory, reactive, and proactive parts of each individual role. Interactions of roles inside the same agent depend on the agent-level knowledge in that if a role changes the agent-level knowledge, it will inform other roles about the changes.

2.2.3 Mapping Strategy between Roles and Agents

E. A. Kendall proposed in her agent analysis methodology that the roles that a given agent needs to play have to be identified in the agent system design. However, the methodology does not include rules to follow in the mapping process. Here, we propose a mapping strategy to guide the role assignment. The objective of the mapping strategy is to increase the running performance of the fault tolerant agent system under our rollback recovery protocol. The strategy includes two rules that help to reduce message communication while maintaining scalability of the system.

Rule 1: Start with a one-to-one mapping between roles and agents.

Rule 2: If two roles share a high volume of message traffic, try to merge them into one agent

2.3 Role-Milestone Based Methodology

Traditional rollback recovery technique provides a solution to make a distributed system fault tolerant; however, it is designed for a traditional distributed system, which does not take into account of the features of a multi-agent system such as sociality. In order to make a rollback recovery protocol work better in an agent system, a new protocol considering the agent features is presented. The role-milestone based methodology described here is proposed to facilitate the design of a selective rollback recovery protocol for the agent system. The methodology can identify the social groups of the agent system and locate suitable places for starting a checkpoint by analyzing some static knowledge of the application at design time. The identified social groups statically point out the selective checkpoint groups for the protocol, which avoids the complicated and inaccurate dependency tracking. The suitable checkpoint places provide static checkpointing, which has advantages such as simplifying the state saving, reducing size of checkpoints, and providing more flexibility in recovery [Silva98] comparing with dynamic checkpointing.

In the following part, we will describe the assumptions for the methodology, introduce the milestone dependency graph, which is the key of the methodology, and explain how to get the milestone dependency graph, and how to analyze the milestone dependency graph in order to get the social groups and decide on checkpoint placements.

2.3.1 Assumption

Not all agent systems are suitable to use the role-milestone based methodology and the selective rollback recovery protocol to achieve fault tolerance. Only those agent systems

that satisfy the following assumptions will be able to benefit from the methodology. First, roles are elements of agents, and each role has one individual goal and is responsible for accomplishing it. Second, roles collaborate with one another to achieve their individual goals. Third, collaborating roles have a common goal, and accomplishing the distributed individual goals are the ways to reach their common goal. Fourth, one agent can participates in multiple roles sequentially or concurrently. However, one agent should not play multiple instances of the same role. Fifth, one role may be played by multiple agents, even if they are the instances of the same agent. The assumptions are not too strict for agent systems, because as long as an agent system can be modeled by roles and their relationships, the agent system can be easily designed to meet these assumptions.

2.3.2 Milestone Dependency Graph

The milestone dependency graph is the key of the role-milestone based methodology. The graph uses milestones of related roles with a common goal as its vertices, and builds the casual relationship between the vertices. Roles in the graph can be mapped to application agents. Each vertex can be tagged with an effort that represents the computation/interaction required before reaching the milestone from the start of the common mission. By analyzing the graph, the social groups and checkpoint places can be identified.

2.3.2.1 Milestone

The notion of milestone is obvious in our everyday life, particularly in group planning and coordinated efforts that would take some time and face some anticipated or unanticipated events in the working environment. The agent paradigm obviously can incorporate this notion of milestone in coordinated agent activities to accomplishing a common mission. This is particularly relevant in the context of agent applications that correspond to automating traditionally human-oriented solutions. In the model of our

agent system, each role has its individual goal, and in its lifeline towards the individual goal, there might be some important states corresponding to some statements of the source program to be reached. These states are milestones toward the individual goal, and the individual goal is viewed as the last milestone of that role.

Milestones are defined by system designers with their own knowledge, and different designers may define different milestones. However, there are conventions for designers to follow. For example, a milestone may be an intermediate goal used to evaluate the progress towards the final set target, a scheduling event that signifies the completion of a major deliverable or a set of related deliverables, a flag indicating the completion of a part of project that may be needed by some specific time, a key event that defines the end of a phase or reaching a target or goal, or a scheduled event to measure progress, etc.

A milestone is inherently a suitable place for starting a checkpoint, because many distributed system are structured as a sequence of phases, where each phase consists of a transient part in which useful work is done, and a stable part in which the system cycles lots of time [Chandy85]. The milestone normally is the transient useful part to do the work, which contains fewer temporary variables and involves fewer temporary collaboration agents. Decision as whether indeed a checkpoint is to be taken depends on the analysis applied in the milestone dependency model to be presented later.

2.3.2.2 Effort of Milestone

Agents cooperate to achieve a common goal, but the cooperation process may fail because of some agent failure. To recover from this failure, all related agents can roll back together to a consistent checkpoint, and re-execute. As we have stated that milestones are good places to make a checkpoint, the question is which milestone should be selected to insert a checkpoint. The notion of efforts of milestones is proposed to answer this question.

Effort of a milestone is the computation/interaction required for reaching a milestone from the start of the mission. Effort can be estimated at design time, or measured by running the system if the agent system is already implemented. The estimated effort is somewhat inaccurate. However, it is still useful in the design stage. Details about the effort measurement are given later in subsection 2.3.3.3. The reason for using effort to select milestones for checkpoints is that after a long execution time, saving the state of the collaboration agents can avoid restarting the mission from the start point once failure happens. This strategy aims to optimize the system performance by taking into account the cost of checkpointing and the latency in recovery.

2.3.2.3 Milestone Dependency Graph

Roles with common interest have to cooperate to achieve their common goal, and the cooperation towards a common goal introduces dependency among milestones of different roles. Therefore, strictly speaking, all relevant roles contribute towards a milestone, and the over-all contribution can be reflected by having causal influence towards the milestone. Moreover, roles with one common goal have to collaborate together in a social group.

In order to capture the dependency relationship of milestones toward a common goal and calculate the over-all effort of a milestone, the milestone dependency graph is developed to graphically represent the inter-dependence of milestones. In the graph, a box presents a role, which contains the milestones belonging to this role. A vertex presents a milestone, a filled circle means the start point of a role towards its goal, and a filled vertex presents the individual goal of a role. An arrow between vertexes indicates the "happens-before" relationship, and for the arrow inside a role, a number or a set of numbers is attached to the arrow indicating the effort needed to reach the destination vertex from the starting vertex (to be explained later). For example, vertex A $\xrightarrow{a}$ vertex B means that milestone A

happens before milestone B, and the effort required from A to B is "*a*". The A "happens-before" B relationship indicates that A might lead to B, and if B happens, A must have happened. The "happens-before" relationship is transitive, for example, if A → B → C, A → C is omitted. A sample milestone dependency graph is shown in figure 2-3.

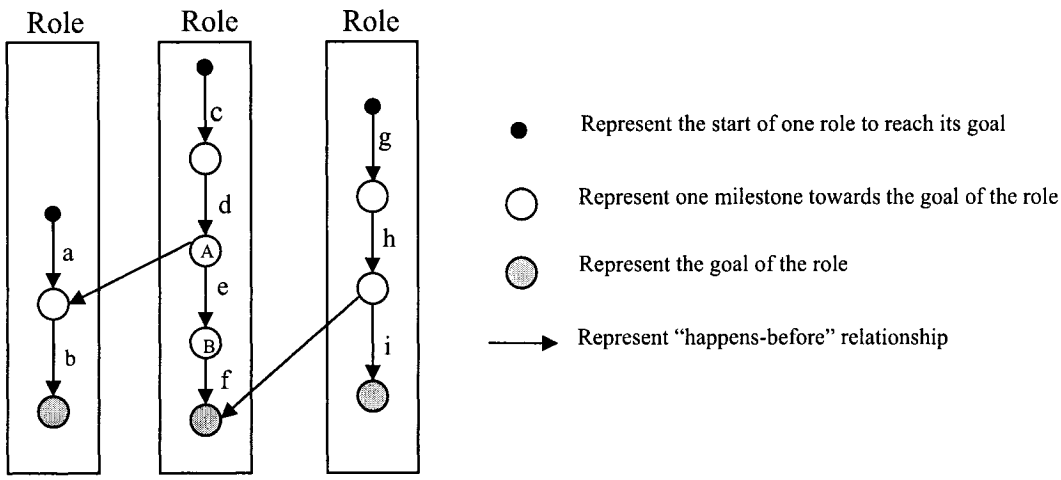


Figure 2-3: Milestone dependency graph

To measure the effort of a milestone, two approaches can be taken into account: i) individual agent efforts, which are localized to an agent, or ii) global agent efforts, which include the efforts of all relevant agents that contribute towards this milestone. Before we elaborate the two approaches, we first introduce three definitions.

Definition 1: effort (*A*, *B*) = individual computation/communication required, which is localized to a role played by the agent under consideration, to reach milestone *B* from milestone *A*.

Definition 2: effort $_{L}$ (*A*) = individual computation/communication required, which is localized to an agent, to reach milestone *A* from the start points (each role played by the agent has one start point; therefore an agent with *n* roles have *n* start points) of a common goal.

Definition 3: effort $_G$ (A) = global computation/communication required, which is globally required of all relevant agents, to reach milestone A from the start points of the common goal.

Definition 1 defines the incremental effort between two milestones in a same role. The effort attached to the arrow in milestone dependency graph is actually an incremental effort. Definition 2 and definition 3 are for the first and second approaches. "effort $_L$ (A)" is the individual agent effort of milestone A, and "effort $_G$ (A)" is the global agent effort of milestone A. Both the two approaches have their pros and cons, which one is better depends on the actual situation and the checkpoint recovery algorithm to be used. We extend the basic milestone dependency graph mentioned above to two extended graphs to elaborate the two approaches.

i) Individual milestone dependency graph:

Individual milestone dependency graph is for calculating the individual agent effort of one milestone. In the graph, only the roles played by a same agent and contributing to the common goal are introduced. Therefore, individual milestone dependency graph includes only one agent. The graph follows the rules of the milestone dependency graph. In addition, a dashed box is used to represent an agent.

ii) Global milestone dependency graph:

Global milestone dependency graph is for calculating the global agent effort of one milestone. In the graph, all the collaboration roles contributing to the common goal are introduced. Therefore, global milestone dependency graph includes more than one agent. The graph basically follows the rules of the individual milestone dependency graph. However, instead of only labeling the incremental effort on the arrow from one vertex to the other, the number of agents that participate in the transition is also labeled on the arrow. The reason is that: in the collaboration, one role may be played by multiple agents concurrently, and for each milestone, the number of agents reaching that milestone may

be variable. For example, one auctioneer agent may negotiate with ten bidder agents currently; however, not all bidder agents can reach the goal of winning the auction. Therefore, for the incremental effort attached to the arrow towards a milestone in the global milestone dependency graph, we have to estimate the number of agents contributing to the milestone in addition to the incremental effort required. The number of agents is estimated by the designer's knowledge of the system, and it is graphically denoted with a separating slash as shown in figure 2-4 (b).

Examples of the individual milestone dependency graph and the global milestone dependency graph are given in figure 2-4 (a) and figure 2-4 (b). In the two graphs, agents playing multiple roles are represented by dashed boxes. Individual milestone dependency graph includes only one agent, and global milestone dependency graph includes more than one agent.

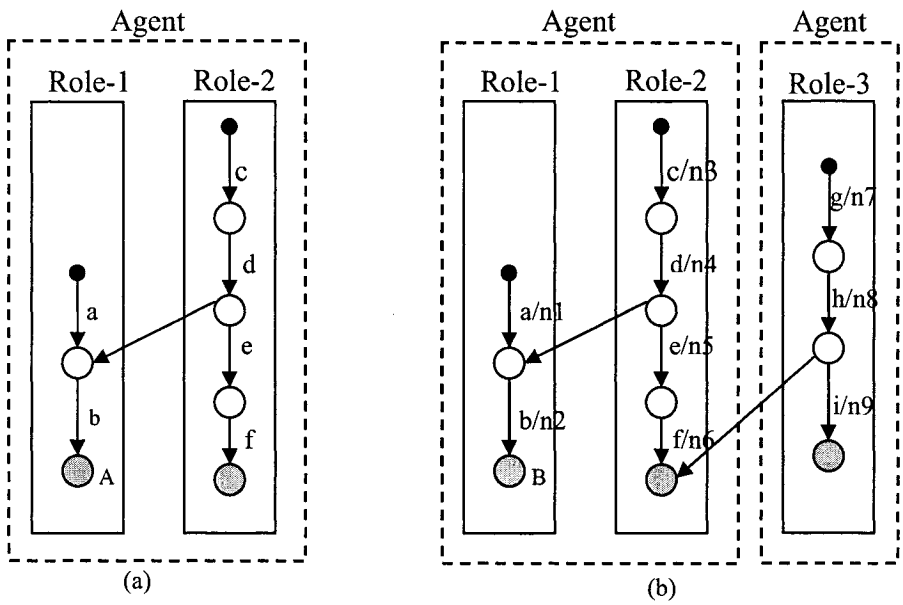


Figure 2-4: (a) Individual milestone dependency graph; (b) Global milestone dependency graph

2.3.3 Social Group Identification and Milestone Selection

A milestone dependency graph can be drawn from role models, and by analyzing the milestone dependency graph, we will be able to identify the social groups and select the milestones suitable for starting checkpointing. In the following, we explain the different usage of individual and global milestone dependency graphs, outline the steps to derive them, elaborate how to estimate the incremental effort and calculate the individual and global effort, and finally present the overall solution strategy.

2.3.3.1 Usage of Individual/Global Milestone Dependency Graph

Different rollback recovery protocols may be suitable for different situations. In the same way, the agent based selective checkpoint recovery protocol we try to propose follows the same principle. The individual and global milestone dependency graphs aim to answer the question and distinguish the situation where we should use a selective checkpoint.

Agents collaborate to achieve their common goals, and depending on the role/agent collaboration frequency, we classify the collaborating agents into two categories: closely-coupled agents, where many messages are exchanged in their lifelines; and loosely-coupled agents, where infrequent messages are exchanged in their lifelines. In the two situations, it is obvious that coordination checkpointing works better for closely-coupled agents, and log-based rollback recovery works better for loosely- coupled agents. So, we use individual milestone dependency graphs for loosely-coupled agents to select milestones suitable for starting independent checkpoints, and global milestone dependency graphs for closely-coupled agents to identify the social groups and select milestones for selective checkpointing. More details will be given in subsection 2.3.3.4.

2.3.3.2 Getting Individual/Global Milestone Dependency Graph

Both individual and global milestone dependency graph can be drawn from role models proposed by E. A. Kendall. The steps are:

Step 1: List the individual goals, common goals, and milestones for all of the roles.

Step 2: Identify the loosely-coupled and closely-coupled agents of each common goal.

Step 3: For each loosely-coupled agent, draw the individual milestone dependency graph.

Step 4: For closely-coupled agents of a common goal, draw the global milestone dependency graph.

Step 5: Estimate the incremental effort of each arrow belonging to one agent.

2.3.3.3 Estimate Effort

The incremental effort attached to the arrow of a milestone dependency graph can be estimated at design time. However, the incremental effort may be variable and not necessarily determinable, because milestones are often reached through iterative means and the number of iterations may vary. To overcome such issues, we provide a <*min*, *max*> range to bind the typical scenarios, and the incremental effort is represented by "(*min* + *max*)/2" to indicate the average between them. Instead of estimation at design time, another approach is to record the running time of the system. With this approach, the application system should already be implemented, and the source code should be instrumented as per milestone. After getting the incremental effort of each individual/global milestone dependency graph, we can calculate the individual/global agent effort of each milestone.

i) Individual agent effort calculation:

The individual milestone dependency graph is used to estimate the individual agent effort. The individual agent effort of one milestone is the sum of all the incremental efforts attached to the arrows pointing towards the milestone in the individual milestone dependency graph. For example, in the above figure 2-4 (a), "effort $_{L}(A) = a + b + c + d$".

ii) Global agent effort calculation:

The global milestone dependency graph is used to get the global agent effort. The global agent effort of one milestone is the sum of the products of the incremental effort and the associated number of agents attached to the arrows pointing towards the milestone. For example, in the above figure 2-4 (b), "effort $_G(B) = a \times n_1 + b \times n_2 + c \times n_3 + d \times n_4$".

2.3.3.4 Social Groups and Milestone Selection

To select the right rollback-recovery protocol and the proper milestones for starting checkpoints, we should consider both the local and global situation. The basic rule is to find a trade-off between introducing less failure-free overhead and at the same time minimizing the rollback distance during recovery.

For an agent with fewer messages exchanged, log-based rollback recovery fits it better. With the log-based protocol, the failure of an agent triggers recovery of that agent to its most recent checkpoint and then message replay is applied. Because fewer messages are recorded, the overhead is small and the runtime performance is better. This solution is a localized strategy whereby the effect is completely localized to the agent. Upon a failure, the computation lost is the individual agent effort from the failure point to the recent checkpoint. Therefore, we use an individual milestone dependency graph to capture the individual agent effort to reach a milestone. With the pre-known individual agent effort, all the milestones can be ordered, and one threshold can be defined to be the tradeoff between failure-free running performance and recovery speed. Every two milestones with their individual incremental effort beyond the threshold can be selected to start independent checkpoints.

For closely-coupled agents, the coordination checkpoint solution works better than the log-based solution. In coordinated checkpointing, an event can trigger a set of local checkpoints to be taken together among a set of agents to form a consistent state so that it can be used for recovery when a later failure occurs. In the meantime, the failure of one

agent will force all the related agents to roll back to their latest checkpoints. Selection of milestones for triggering coordinated checkpoints should be based on global incremental effort. The theory is that in the occurrence of a failure, the global incremental effort represents the maximum re-execution effort required during recovery.

The coordination checkpoint we talked above is actually a selective checkpoint solution. To identify the coordination agent group and to calculate the global effort, we use the global milestone dependency graph. Only agents playing the roles inside the global milestone dependency graph have to take part in the coordination checkpoint and therefore form one checkpoint group, one social group in other words. However, strictly speaking, the actual agents playing the roles may not be statically identified before a system run. In the dynamic running time, there may be multiple agent instances of a single agent type running in the system; and these agent instances may form different groups to achieve different instances of the same common goal. For example, a customer agent and a merchant agent have to collaborate to achieve an agreement of buying a product. During the execution, customer agent A_1 and merchant agent B_1 may collaborate for product P_1, and customer agent A_2 and merchant agent B_2 may collaborate for product P_2. But, the knowledge of a system is helpful in the dynamic group identification. If interaction protocols among the types of agents included in the global milestone graph are properly designed (such as a registration protocol which forces agents to register into the checkpoint initiator agent before participating in their common goal), the groups can be dynamically identified during the system run. For instance, in an auction system, if a subscribe protocol is defined between the bidder and auctioneer agent, at runtime an auctioneer agent can maintain knowledge of the participating bidders, which includes agents who should participate in a group checkpoint toward the common goal of achieving an agreement of buying a product between bidders and auctioneers.

Normally the selection of milestones for starting coordination checkpoints follows the same rule as the log-based solution. By ordering the milestones with their global agent efforts, we can define one threshold. Every two milestones with their global incremental effort beyond the threshold can be selected to start coordination checkpoints. However, global efforts of milestones are impossible to estimate when too many agents play a same role in the collaboration. For example, in the auction scenario, the number of participating bidder agents is variable, and may have huge difference. The wide range makes the global agent effort estimation not valuable. One alternative is to use the critical (longest effort) path in the global milestone independency graph. The critical path is a safe ground for serializing coordination so that these checkpoints only happen sequentially at measurable distances from each other. By measuring incremental effort along this critical path, a new coordinated checkpoint can be designed into the system.

Chapter 3 SCLR Protocol

3.1 Introduction

We have described the methodology for distinguishing the situation when selective checkpoint based protocol should be used, and elaborated how to identify selective checkpoint groups and the locations for starting checkpoints. In this chapter, we will describe the design of such a protocol named "Selective Checkpoint with Log Rollback-recovery protocol" (SCLR protocol for short).

The SCLR protocol combines selective checkpointing and logging together, which results in domino-free rollback, simple recovery protocol, simple garbage collection, and fast output commit. The system includes a checkpointing algorithm, a logging algorithm, a recovery algorithm, and a pruning algorithm. The checkpointing algorithm is used in pre-defined program locations to store the states of group of agents and ensure the consistency of the stored states. The logging algorithm is for logging nondeterministic events that may not be reproducible. The recovery algorithm uses the checkpoints and logs to ensure that once failures occur, the agent system can recover to a consistent system state and the outside world is not affected. The pruning algorithm is in charge of garbage collection that removes useless checkpoints and logs. The proposed rollback-recovery protocol is a user-defined solution where we assume the designer has the knowledge of the whole agent system.

In the following sections of the chapter, we will first describe the system model and the objectives of the proposed protocol. Then, we elaborate its details. We also provide the proofs for the correctness of the protocol. At the end, we briefly discuss how to use the protocol in a multi-agent system.

3.2 System Model

The distributed multi-agent system consists of a variable number of agents that communicate only through messages. The communication systems deliver messages reliably and in first-in-first-out order. Agents cooperate to execute a distributed application program and interact with the outside world by receiving and sending input and output messages. The system is asynchronous. There exists no bound on the relative speeds of agents, no bound on message transmission delays, and no global time source. Agents have access to a stable storage device that survives failures.

The execution of an agent consists of a sequence of piecewise deterministic state intervals [Storm85, Elnozahy92]. Each state interval starts by the occurrence of a nondeterministic event. Such an event may be: (i) the creation of an agent, (ii) an agent receives a message from another agent, which is called a message receipt event, (iii) an agent receives a message from the outside world, which is called an input event, and (iv) a decision event. A decision event occurs when an agent surveys a known and fixed set of alternatives, weighs the likely consequences of choosing each, and makes a choice [Klein93]. Decision events allow an agent to choose and follow different paths through a scenario, and each choice provides a path for the agent to follow toward the conclusion of the scenario.

The execution during each state interval of an agent is deterministic, such that if an agent starts from the same state and is subjected to the same nondeterministic events at the same locations within the execution, it will always yield the same output [Elnozahy92]. A concept related to the state interval is the piecewise deterministic assumption (PWD). This assumption states that the system can detect and capture sufficient information about the nondeterministic events that initiate the state intervals.

Figure 3-1 shows a sample execution of a distributed multi-agent system consisting of three agents *p*, *q* and *r*. The notion of "σ_i^p" denotes the i^{th} state interval of agent *p*. the

notion of "m_i^p" denotes the i^{th} message transmitted by agent p. Horizontal lines represent the process of the execution, an arrow between two horizontal lines denotes a message transmission, and a dashed arrow represents an input event from outside. A vertical bar denotes the beginning of each state interval. In the figure, state interval "σ_1^p" starts with a decision event, and state interval "σ_1^r" starts with an input event.

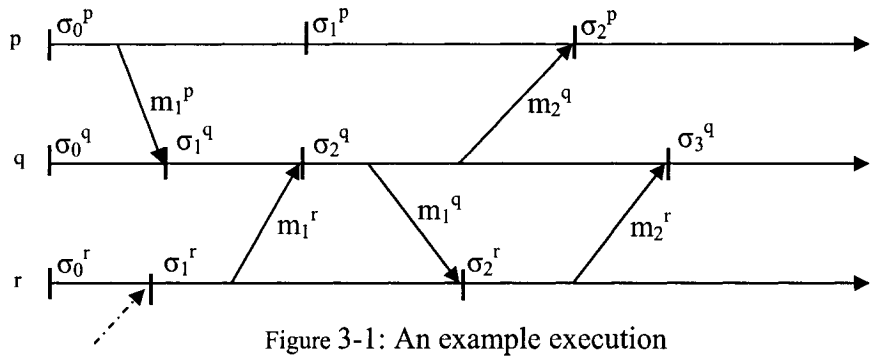


Figure 3-1: An example execution

An execution of the distributed system is represented by a run, which is a total ordering ρ that includes all of the events in the global history and that is consistent with each local history. If a run is pre-defined, the system can follow the same execution, which is the sequence of global states the system passes through during an execution. Each global state is a set of agent states and channel states. In the run, only one agent changes its state between any two adjacent global states. Thus, each pair of adjacent states defines an event that was executed by an agent, and the resulting sequence of events is ordered by the partially order happens-before relationship that represents potential causality.

An agent may fail independently, in which case it loses its volatile state and stops execution according to the fail-stop model [Schlichting83]. However, if the state information and the afterward events of the agent are saved on the stable storage device during failure-free execution, the failed agent can be recovered with the saved state information, and repeat its execution to its pre-failure state by replaying the saved events.

3.3 Motivations, Rationale and Objective

In a multi-agent system where many agents collaborate and interact frequently with each other and with OWP devices, either coordinated checkpoint based rollback-recovery protocol or log-based rollback recovery protocol will produce heavy overhead. Therefore, we want to take advantage of sociality of multi-agent systems to propose a selective checkpoint based rollback-recovery protocol particularly designed for such situations. The new protocol is supposed to have lower performance overhead, no domino effect, fast output commit, and simple garbage collection when implemented in multi-agent systems where many agents collaborate and interact frequently with each other and with OWP. The protocol should also be able to take advantage of the analysis results from the milestone dependency graphs described in chapter 2, which means that the checkpoint groups can be identified without the complicated and inaccurate dependency tracking, and the checkpoint places are predefined at the design time.

To be able to use the analysis results from the milestone dependency graphs, the new rollback-recovery protocol has to be a user-defined solution that is not transparent to designers. As we have mentioned in subsection 2.3.3.4, checkpoint groups can be identified with properly designed application protocols, so that checkpoint initiator agent could know the group members before it starts the group checkpoint (refer to subsection 2.3.3.4). The places for starting checkpoints are those program codes corresponding to selected milestones.

Existing selective checkpoint protocols have high latency of output commit, and they have to keep track of communication dependencies. On the other hand, pessimistic logging has been proved to work efficiently in systems with frequent OWP problems, and it has the benefits of no domino effect and simple garbage collection. Thus, combining pessimistic logging with selective checkpoint protocol provides a safe ground for our new SCLR protocol. Moreover, the idea of introducing logging into group checkpointing and

rolling back group members at the occurrence of failures leads to the possibility that not all events occurring in the system have to be logged. We use a scenario shown in the figure 3-2 to explain it.

In figure 3-2, a vertical line represents the lifeline of an agent; a dashed line denotes the time instance in the lifeline; a vertical black bar represents an individual checkpoint; a rectangle box indicates a group coordination checkpoint consisting of individual checkpoints inside the box; an arrow between two vertical lines denotes a message transmission; a cross-lines on a lifeline denotes that the agent of the lifeline crashes at that time; and a helix represents that during the lifelines of the agents covered under the area, there are heavy messages exchanged among them.

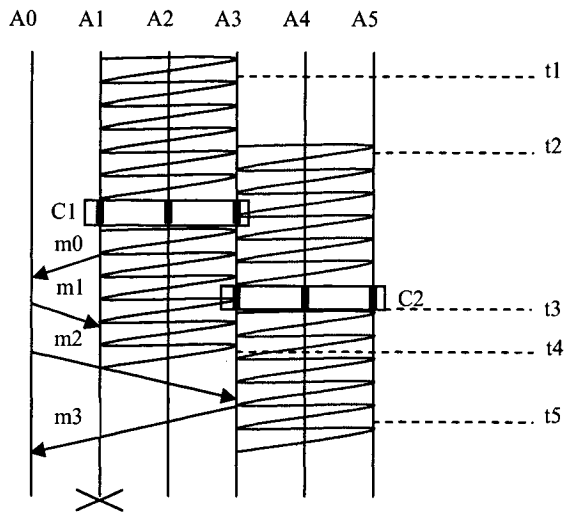


Figure 3-2: A runtime scenario

As shown in figure 3-2, agent A_1, A_2 and A_3 form a group G_1 and exchange messages heavily from time t_1 to t_4; agents A_3, A_4 and A_5 form another group G_2 with heavy messages exchanged among them from time t_2 to t_5. A_1, A_2 and A_3 take group coordination checkpoint C_1 consisting of individual checkpoints of $C_1(A_1)$, $C_1(A_2)$ and $C_1(A_3)$; A_3, A_4 and A_5 take group coordination checkpoint C_2 consisting of individual checkpoints of $C_2(A_3)$, $C_1(A_4)$ and $C_1(A_5)$. Agent A_0 sends and receives messages

occasionally. Therefore, from the checkpoint group view of an agent, after time t_3, A_1 and A_2 have a checkpoint group G_1, A_4 and A_5 have a checkpoint group of G_2, A_3 has two checkpoint groups of G_1 and G_2, and A_0 has empty checkpoint group. In the scenario, if logging technique is not adopted, failure of agent A_1 will cause domino effect that will force all the agents to roll back to the starting points. However, suppose an agent A has several individual checkpoints $C_1(A)$, $C_2(A)$, ..., $C_n(A)$ belonging to group checkpoints C_1, C_2, ..., C_n with the corresponding checkpoint groups at time t being G_1, G_2, ..., G_n respectively. Then, the intersection of the checkpoint groups of the agent A is denoted by ICGS(A) and defined as ICGS(A) = $G_1 \cap G_2 \cap ... \cap G_n$. We find that suppose A logs all the messages from agents not belonging to ICGS(A). Failure of A will not cause a domino effect, and only agents belonging to ICGS(A) have to roll back to C_n, which is the latest group checkpoint of A. For instance, if A_1 and A_2 log all messages sent by agents outside of group G_1, A_4 and A_5 log all the messages sent by agents outside of group G_2, A_3 logs all messages sent by agents outside of $G_1 \cap G_2$ (Actually A_3 logs all messages from others because $G_1 \cap G_2 = A_3$), and A_0 logs all messages from others (A_0 has no group). With this logging strategy, if A_1 crashes as indicated in the diagram, only A_1, A_2, and A_3 (which belong to ICGS(A_1)) have to roll back to their group checkpoint C_1 and re-execute. No other agents will be involved in the recovery process because the messages interchanged among A_1, A_2, and A_3 will be re-generated, and messages from other agents can be replayed as they have been logged before the failure. For example, in the recovery process, A_1 will replay the messages sent by A_0, and A_3 will replay the messages sent by A_0, A_4 and A_5. The message re-generation and replay will lead agents A_1, A_2 and A_3 to repeat their executions to their pre-failure states that are consistent to the survivor agents. With this logging strategy, garbage collection is simple too. Because there is no domino effect, a group checkpoint can be pruned if none of its individual checkpoints is the latest checkpoint of its corresponding agent.

The SCLR protocol explained above inherits the advantages of no domino effect, fast output commit, and simple garbage collection from pessimistic logging. Moreover, because in a multi-agent application, message exchanges among agents in different social groups are infrequent, the strategy can effectively lead to a significant reduction of messages to be logged at runtime.

3.4 Selective Checkpoint with Log Rollback-Recovery Protocol

We have described the rationale of the SCLR rollback-recovery protocol. In this section, we will elaborate the details of the protocol. First, we will introduce the concepts of Checkpoint Group (CG), Checkpoint Group Set (CGS), and Intersection of CGS (ICGS); then we will describe the data structures for supporting the protocol; finally we will elaborate the checkpointing algorithm, logging algorithm, recovery algorithm, and pruning algorithm included in the protocol.

3.4.1 Checkpoint Group (CG), Checkpoint Group Set (CGS), and Intersection of CGS (ICGS)

Checkpoint Group (CG), Checkpoint Group Set (CGS), and Intersection of CGS (ICGS) are three important concepts of the proposed protocol, which capture the checkpoint group changes of an agent during its lifetime. They focus on the group aspect of an agent from the view of checkpoint. Because we try to use social groups as our checkpoint groups, it is reasonable to believe that they also capture the social group aspects of a multi-agent system.

A Checkpoint Group (CG) is a set of agents participating in a group checkpoint. CG focuses more on checkpoint aspect. Different agents participating in a same group checkpoint have the same CG. We can present CG of a group checkpoint C as C.CG.

A Checkpoint Group Set (CGS) is a collection of CGs, each CG corresponding to one un-pruned group checkpoint that an agent holds. CGS focuses more on the agent aspect, which describes the group changes during one agent's lifetime. In the lifetime of an agent, its CGS may increase or diminish since new group checkpoints may occur and previous group checkpoints may be pruned by garbage collection. We can present CGS of an agent A as CGS(A).

Intersection of CGS (ICGS) is the intersection of all CGs in CGS of an agent, which is a set including those agents belonging to every CG in GGS. We can formally define it as ICGS = $\{A \mid A \in \text{CG for all CG} \in \text{CGS}\}$. ICGS is the key concept in our protocol. We can present ICGS of an agent A as ICGS(A).

If ICGS(A) is a set S, A has to log those messages sent by agents outside S, and in the occurrence of crash of A, only agents inside S have to roll back together with A. Details about logging and recovery will be explained next. We give an example in figure 3-3 to show the changes of CGS, where seven agents take selective coordination checkpoints from time to time in their lifelines.

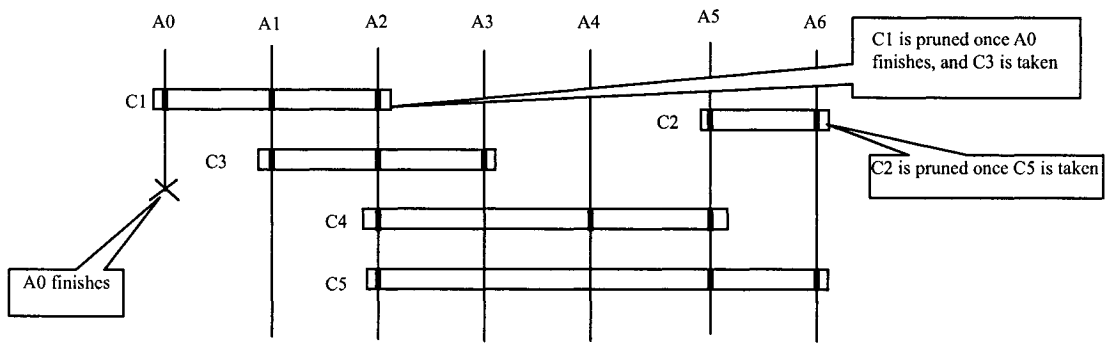


Figure 3-3: Coordination group set

In figure 3-3, a vertical line represents the lifeline of an agent; a vertical bar denotes an individual checkpoint; a box indicates a group coordination checkpoint. According to the diagram, CGS(A_5) after taking group checkpoint C_2 is $\{\{A_5, A_6\}\}$, CGS(A_5) after taking group checkpoint C_5 is $\{\{A_2, A_4, A_5\}, \{A_2, A_5, A_6\}\}$, ICGS($A_5$) after taking group

checkpoint C_5 is $\{A_2, A_5\}$, and CGS(A_6) after taking group checkpoint C_5 is $\{\{A_2, A_5, A_6\}\}$.

In the lifetime of an agent, the reduction in its CGS happens because of garbage collection and the increase is caused by the creation of new group checkpoints. Garbage collection is only needed when new checkpoints are created, and the diminishment of CGS helps to reduce the number of messages to be logged. Therefore, the best time to update CGS of an agent should be right after a new group checkpoint is created. In our user-level checkpointing algorithm, checkpoint places are pre-defined, so that at runtime a new group checkpoint is created only at the time when a milestone of a global milestone dependency graph is reached, which corresponds to the execution of a program statement of the checkpoint-initiating agent who then starts the group checkpoint. As one agent may participate in several common goals in its life, one agent may appear in several global milestone dependency graphs. Therefore, dynamically speaking, in the whole lifetime of an agent, the agent may take selective checkpoints in different goals, which increases the complexity of the checkpoint group changes in an agent's life. In theory, an agent may appear in both global and individual milestone dependency graph that involve both selective and independent checkpoint; however, in this chapter, we focus only on the selective checkpoint part. At the end of this chapter, we will give a brief view of how to mix these solutions together.

3.4.2 Data Structure

Each agent maintains a set of data structures to support the rollback-recovery protocol. The data structures include the following:

SI: The index of the current state interval of the agent. It is incremented each time a nondeterministic event occurs in an agent. SI is important for event replay.

CGS: a table maintaining the CGS of the agent.

SSN: a table maintaining the sequence numbers of the messages sent by the agent to each other agent with which this agent has communicated (SSN_p^q denotes the sequence number of the message sent from agent p to agent q). For each receiver agent q, the sender agent p maintains an SSN_p^q that starts from 1, and SSN_p^q is attached to the application message to indicate the order of messages sent to the receiver q. This is used for duplicate message suppression in recovery.

RSN: a table recording the highest SSN value received in a message sent by each other agent with which this agent has communicated (RSN_p^q denotes the highest sequence number of the message received by p from q). In the occurrence of one agent crash, the crashed agent will roll back to its latest checkpoint and re-execute. During the recovery, messages sent before the crash may be re-sent to the receivers, and the highest SSN of the receiver process is used to detect the duplicate messages. Moreover, in the occurrence of message delay, the highest SSN can be used to order the receiving messages.

Each of these data items except CGS must be included in the local checkpoint of an agent. When an agent is restarted from its local checkpint, its value will be restored along with the rest of the checkpoint data. If c is a local checkpoint, we use c.SI, c.SSN, and c.RSN to identify the above-mentioned data, and use c.CID, c.CG to identify the checkpint UID and the Coordination Group (CG). CGS is not included in the checkpoint as it can be calculated from the checkpoint history.

3.4.3 Selective Checkpointing Algorithm

The selective checkpointing algorithm stores the states of agent groups, which form a partial consistent state across the group members. We call the stored states as a group checkpoint, and call the state of each group member as the individual checkpoint of the group checkpoint. Therefore, group checkpoint indicates all its member individual checkpoints; however, it is the individual checkpoint that exists. Individual checkpoints

of a same group checkpoint have the same CID and CG. CID is a global unique id generated by the checkpoint initiator to represent a checkpoint UID.

As the algorithm is a user-level solution, designers can make use of their knowledge of the system to identify the group agents without tracking of the communication among them. The global milestone dependency graph proposed in chapter 2 provides a way to identify the group. Application protocols must be properly designed to make sure that the checkpoint initial agent knows the group members (CG) before starting the group checkpoint. Moreover, local milestones for starting group checkpoints are statically chosen from roles of agents by analyzing the global milestone dependency graph. When an agent finishes the last statement of the program codes corresponding to a selected milestone, the agent can be programmed to function as an initiator/coordinator and send marker messages to the group members to start a group-wise consistent checkpoint. In the following, we describe the checkpointing algorithm. The selective coordinated checkpointing algorithm is a blocking solution.

The algorithm proceeds as follows:

Step 1: The coordinator agent starts a new group-wise consistent checkpoint by generating CID and sending marker messages that contain CID and CG to each group member.

Step 2: Upon receiving a marker message, an agent starts a checkpoint and stops its computation.

Step 3: After an agent writes its individual checkpoint onto the stable storage, it sends an acknowledgment message to the coordinator agent.

Step 4: The coordinator agent waits for the responses from the group members, and if all the acknowledgement messages have been received, it broadcasts a release message to each group member. When an agent receives a release message from the coordinator, it resumes its execution.

The individual checkpoint of an agent includes CID, CG, all other data defined in subsection 3.4.2, and its application knowledge. Each individual checkpoint of a same group maintains a copy of CG. Markers can be application messages attached with CID and CG, and markers are non-determinant events, which may be logged if necessary according to the logging algorithm described below.

In the system, because an agent may participate in several different common goals concurrently, a checkpoint request may arrive before it finishes its first checkpoint. The interleaving can be avoided by making the checkpointing algorithm an atomic action. An agent cannot start another checkpoint before it finishes its current checkpoint in progress.

3.4.4 Pruning Algorithm

In our design, a group checkpoint consists of all individual checkpoints of its group members. When all the individual checkpoints are not the latest checkpoints of their owner agents, the group checkpoint becomes useless, and all the individual checkpoints can be pruned. Therefore, the rule for the garbage collection is that a group checkpoint and the events logged before the checkpoint can be pruned when all its members' individual checkpoints are not the latest checkpoints of their owner agents. The pruning algorithm should be called once new checkpoints are created as indicated in subsection 3.4.1.

Figure 3-4 shows the pruning algorithm. We use C to represent an individual checkpoint of agent p, use C.CID to indicate its checkpoint UID, use C.CG to represent CG of C, use C.CG(q) to represent the status of the member agent q *of* C.CG. The status of C.CG(q) are either "un-prune" or "prune", which presents that the individual checkpoint C_q of agent q (C_q belongs to the same checkpoint group of C with C_q.CID = C.CID) is the latest checkpoint of agent q or not respectively. At the time checkpoint C is created, all its member status is set as "un-prune". When agent p receives a "release" message from

checkpointing algorithm indicating that p creates another new checkpoint successfully, p sends "prune" messages attached with C.CID to all agents in C.CG indicating that C is no longer the latest checkpoint of p. If p receives a "prune" message attached with *cid* (*cid* = C.CID) from agent q, p can set C.CG(q) as "prune". When the status of all its member agents in C.CG of agent p is "prune", C and all events logged by p before C can be removed and CGS(p) is re-calculated to make sure that ICGS(p) is the largest possible set.

```
Agent p receives release(cid) message {
        C ← last checkpoint of agent p
        For each agent q: (q ∈ C.CG)
                Send prune(C.CID) message to agent q
}

Agent p receives prune(cid) message from agent q {
        C ← checkpoint of agent p where C.CID == cid
        C.CG(q) ← prune

        Call prune(C)
}

Procedure Prune(C) {
        Prunable ← true

        For each agent r: (r ∈ C.CG)
                If C.CG(r) ≠ prune
                        Prunable ← false
                        Return

        If Prunable == true
                Remove C
                For each logged event e
                        If e.SI ≤ C.SI
                                Remove e

         CGS ← Φ
         For each checkpoint c taken and not removed
                        CGS ← CGS ∪ c.CG
}
```

Figure 3-4: Pruning algorithm

3.4.5 Logging Algorithm

The rollback-recovery protocol we proposed uses partial logging to avoid domino rollback. Partial logging chooses a subset of messages to be logged, depending on the ICGS of an agent. This partial logging can reduce the number of messages to be logged.

In our design, receiver-based pessimistic logging is used to simplify the logging algorithm. We believe pessimistic logging will not introduce much overhead to our system because fewer events would be logged in our protocol. The events to be logged in the system include message receipt events, input events, and decision events. To replay an event in the recovery process, some information must be recorded to deterministically re-create it. A message receipt event can be determined by a tuple <*m*.source, *m*.ssn, *m*.dest, *m*.si, *m*.data>, where *m*.source and *m*.dest denotes respectively the identity of the sender agent and the receiver agent, *m*.ssn denotes the unique identifier (a sequence number) assigned to m by sender, *m*.si denotes the index of the state interval initiated by *m*, and *m*.data is the application data carried by *m*. In the same way, an input event is determined by <*m*.source, *m*.dest, *m*.si, *m*.data> where *m*.source denotes the identity of the outside process. A decision event is determined by <*m*.dest, *m*.si, *m*.data>.

Figure 3-5 shows the logging algorithm. Message receipt events are logged in their delivery. Input events and decision events are logged right after their occurrence but before handling them. An agent receives one message if the message is put in the agent's message queue by the communication system, and an agent delivers one message if the message is read from the message queue. Here, we assume the message queue of an agent is part of the underlying communication system, and it does not fail when the agent crashes.

Agent *p* sends data to agent *q* {
    $SSN_p^q \leftarrow SSN_p^q + 1$
    m.source $\leftarrow p$
    m.ssn $\leftarrow SSN_p^q$
    m.dest $\leftarrow q$

```
        m.data ← data
        send m to q
}

Agent q delivers message m {
        s ← m.source
        If RSN_q^s ≥ m.ssn then
                Ignore the message due to duplication
        Else if RSN_q^s == m.ssn -1 then
                SI ← SI + 1
                m.si ← SI
                RSN_q^s ← m.ssn

                If s ∉ ICGS(q)
                        Log m to stable storage
                Deliver m.data
}

Agent q receives data from environment process p {
        SI ← SI + 1
        m.source ← p
        m.dest ← q
        m.si ← SI
        m.data ← data
        Log m to stable storage
}

Decision event e occurs in agent p {
        SI ← SI + 1
        m.dest ← p
        m.si ← SI
        m.data ← e
        Log m to stable storage
}
```

Figure 3-5: Logging algorithm

3.4.6 Recovery algorithm

Suppose the most recent local checkpoint of agent p is $C(p)$, which belongs to a group checkpoint C. When agent p fails, p will be restarted from $C(p)$, and each agent q in ICGS(p) other than p (CGS(p) can be calculated with the checkpoint history of p) will be informed to roll back to $C(q)$, which belongs to group checkpoint C and $C(q)$.CID $=C$.CID. In the recovery process, these recovery agents will re-execute to their pre-failures states by re-exchanging their messages among themselves and replaying the events logged after the group checkpoint.

A recovering agent will load its corresponding checkpoint and restore its SI, SSN, RSN, CGS, and its application knowledge. Therefore, SSN_p^q of a message resent from p to q before the failure can be regenerated as the same SSN_p^q of the original message. In the receiver side, when agent p receives a message m with SSN_q^p from q, p can ignore m as duplicated message if SSN_q^p is less than or equal to RSN_p^q saved in the restored RSN. Otherwise, m should be delivered to agent p sooner or later.

Figure 3-6 shows the recovery algorithm. A three-phase protocol is used to guarantee that all the relevant agents will roll back. When agent p is detected as crash, p is recreated with its latest checkpoint. After the creation, p will send "rollback" messages to others agents in ICGS(p). A rollback agent, in the receipt of one "rollback" message, will reset its state and send "ready" message to p. After all the "ready" messages are received by p, p will send "commit" messages to members in ICGS(p). All the recovery agents, in the receipt of "commit" messages, begin to re-execute. The recovery agents include the crashed agent and the rollback agents.

```
Recreate crashed agent p {
        C ← the most recent checkpoint of agent p
        Restore C
        Log ← events logged after the checkpoint C

        CGS ← Φ
        For each un-pruned checkpoint c taken by agent p
                CGS ← CGS ∪ c.CG

        For each agent q: (q∈ICGS(p) and q ≠ p)
                Send rollback(C.CID) message to agent q

        For each agent q: (q∈ICGS(p) and q ≠ p)
                Block to receive ready(C.CID) message from q

        For each agent q: (q∈ICGS(p))
                Send commit(C.CID) message to q
        Block to receive commit(cid) message
}

Agent q receives rollback(cid) message from agent p{
        C ← the checkpoint of agent q where C.CID ==cid
        Log ← events logged after the checkpoint C
```

```
        If C is empty
                Skip the rollback message
                Send ready(cid) to agent p
                Continue as normal

        If SI > C.SI
                Restore C
                CGS ← C.CG
                For each un-pruned checkpoint c taken before C by agent q
                        CGS ← CGS ∪ c.CG

                Send ready(cid) to agent p
                Block to receive commit(cid) message
        Else
                Skip the rollback message
                Send ready(cid) to agent p
                Continue as normal
}

Agent q receives commit(cid) message from agent p {

        If agent q is waiting for commit(cid) message
          While Log is not empty
              e ← next event in Log

                If (SI+1 == e.si)
                    SI ← SI+1
                   If (e is of the form (source, ssn, dest, si, data))
                        RSN_dest^source ← RSN_dest^source + 1
                        Replay the message e
                    Else if (e is of the form (source, dest, si, data))
                        Replay the input event e
                    Else if (e is of the form (dest, si, data))
                        Replay the decision event e
                  Else
                     Block to receive message m from agent r: (r∈ICGS(q))

                     If RSN_q^r ≥ m.ssn then
                        Ignore the message due to duplication
                     Else if RSN_q^r == m.ssn -1 then
                        SI ← SI + 1
                        RSN_q^r ← RSN_q^r + 1
                        Deliver m.data
                     Else if RSN_q^r < m.ssn -1 then
                        Delay delivering m
                     Continue as normal
        Else
          Skip the commit message
          Continue as normal
}
```

Figure 3-6: Recovery algorithm

More than one agent may crash at the same time, and one agent may receive multiple rollback or crash requests. We use the "First come, first served" strategy to deal with the multiple requests. If a request for one agent comes after the completion (three-phase protocol) of the previous request, the two requests are separated; otherwise, they are interleaved and may cause the interleaved three-phase protocols.

In interleaved three-phase protocols, the action to the first request has been described in the single crash part, and the actions to the coming request are different according to the different scenarios. Four scenarios may happen. Scenario 1: the coming request of one agent is a crash request, which may happen if the agent is first requested to roll back, then is discovered as crash. In this situation, the agent is recreated with the first rollback checkpoint, and the crash request is ignored. Scenario 2: the coming request is a rollback request asking for the agent to roll back to the same checkpoint as the first request, which may happen if more than one agent in the same group crashes during the same period. In this situation, the agent just sends a "ready" message to the protocol initiator agent to inform that the agent has rolled back. Scenario 3: the coming request is a rollback request asking for the agent to roll back to a checkpoint later than the previous checkpoint. In this scenario, the request can be regarded as served, and a "ready" message is sent to the protocol initiator agent. Scenario 4: the coming request is a rollback request asking for the process to roll back to a checkpoint earlier than the previous checkpoint. In this situation, the agent sets its state to the new state, and sends a "ready" message to the protocol initiator agent.

In the four scenarios, scenario 1 and scenario 2 happen when crashed agents share the same latest group checkpoint. Scenario 3 and scenario 4 happen when crashed agents have the different latest group checkpoints.

3.5 Correctness Proof

The fundamental goal of one rollback-recovery protocol is to bring the system into a consistent state when inconsistencies occur because of failures, which should be consistent with the observable behavior of the system from the outside environment before the failures. Our rollback-recovery protocol guarantees the consistency in the way that when inconsistency occurs, the rollback-recovery protocol can bring the system into the consistent state before the failures occur. Let ρ be a run of the system before the failures, our recovery protocol guarantees the same events executed in ρ are again executed, and the system follows the same run ρ to the pre-failure global state during recovery. Once the recovery is complete, there are no orphan agents. That is, no surviving agents whose states are inconsistent with the recovered states of crashed agents and rollback agents. Before proving the correctness, we explain some terminologies used in the proof.

(1) Agent $q \in$ ICGS(p): agent q belongs to ICGS(p).

(2) Agent $q \notin$ ICGS(p): agent q does not belong to ICGS(p).

(3) ICGS(q) $\subset$ ICGS(p): ICGS(q) is the subset of ICGS(p).

Lemma 1: When agent p crashes, only agents belonging to ICGS(p) need to roll back. A crashed agent or a rollback agent will re-execute to its pre-failure state which is consistent with the survivor agents.

Proof: Suppose agent p crashes, and restarts from its latest group checkpoint C. All agents q: $q \in$ ICGS(p) have to roll back to checkpoint C together. Because $q \in$ ICGS(p), q participates in every group checkpoint that p participates. Thus, ICGS(q) $\subset$ ICGS(p), and q logs any message sent by agent outside of ICGS(p). Therefore, agents in ICGS(p) will re-execute by re-exchanging messages among them and relaying the messages sent from agents outside the group.

Furthermore, according to PWD assumption, if the state information and the afterward events of an agent are saved on the stable storage device during failure-free execution, the failed/rollback agent can be recovered with the saved state information, and repeat its execution to its pre-failure state by replaying the saved events. For a crashed/rollback agent in our system, all the events that happened after the checkpoint can either be replayed or re-generated because the events are either logged in stable storage or re-generated in the recovery process. Therefore, a crashed agent or a rollback agent will re-execute to its pre-failure state which is consistent with the survivor agents. □

Lemma 2: Multiple agent crashes can be tolerated by the recovery algorithm.

Proof: Each crash triggers a group of agents to recover together and its correctness is proved in lemma 1. Consider that two agent crashes occur concurrently. If they do not belong to the same group, the crash triggers two independent crashes and recovery actions. Hence the correctness is guaranteed by lemma 1. If they belong to the same group, the crash and recovery manager will accept the first and reject the second, and a single crash recovery will be triggered. This reasoning applies through induction when the number of concurrent crashes goes beyond 2. □

Lemma 3: As selective group checkpoint can be taken correctly in the presence of an agent crash.

Proof: In a failure-free run, the checkpoint protocol induces a checkpoint to be created in each agent of the group. However, the checkpoint protocol may be interfered with the crash or recovery of an agent in the group. Five separate cases have to be considered in the proof:

Case 1: An agent p crashes at the time of taking checkpoint C before it saves its state to stable storages, and p is the checkpoint initiator:

Agent p recreates itself with its latest checkpoint, and agents in ICGS(p) roll back with p. During the recovery, p will re-execute and re-start the checkpointing C. $\forall\ q$: $q \in C$.CG, if

$q \in$ ICGS(p), p will re-execute, re-receive the checkpoint message, and re-do the checkpoint. If $q \notin$ ICGS(p), then q will stop waiting for the release message, and the acknowledgement message sent by q to p is either logged by p before p crashes, or remains in the message queue of p. Agent p, after re-starting the checkpoint C, can replay/deliver the acknowledgement messages sent by agents outside of ICGS(p), and can re-receive the acknowledgement messages sent by agents inside ICGS(p) after their recovery. Therefore, agent p can send release messages to all the checkpoint participators and finish the checkpoint C.

Case 2: An agent p crashes at the time of taking checkpoint C before it saves its state to the stable storage, and p is not the checkpoint initiator:

Suppose agent q is the checkpoint initiator. If $q \in$ ICGS(p), q will roll back with p, and re-start the checkpoint. The situation is similar to case 1. If $q \notin$ ICGS(p), the checkpoint request message from p to q will be logged by q, and q will re-do the checkpoint by replaying the message.

Case 3: An agent p crashes at the time of taking checkpoint C after it saves its state to the stable storage but before sending the release messages and agent p is the checkpoint initiator:

Agent p recreates itself with C, and tries to roll back agents in ICGS(p) to C. Because ICGS(p) $\subset$ C.CG, all the rollback agents participate in C. Suppose agent $q \in$ ICGS(p). If q already saves its state, q will skip the rollback request, as current SI of q is the same as SI of its local rollback checkpoint. If q does not save its state, the rollback request will be skipped also as the rollback checkpoint does not exist. Thus, only p is recreated with C and it receives/replays the acknowledgment messages, and sends release messages to complete the checkpoint.

Case 4: An agent p crashes at the time of taking checkpoint C after it saves its state to the stable storage but before receiving the release messages, and agent p could be either the checkpoint initiator or not:

Proof: The situation is similar with case 3. Agent p is recreated. All the rollback agents do participated in C, and they will skip the rollback messages. Agent p will receive the release message after its creation.

Case 5: An agent p receives a rollback request in the process of taking checkpoint C, but before C was saved to the stable storage.

If the rollback request comes from an agent that participates in C, we have proved its correctness in the above-mentioned four cases. Consider the situation that the rollback request comes from a crashed agent q that does not participate in C, and asks p to roll back to a checkpoint that is taken before C. The situation happens when agent p participates in each checkpoint that agent q participates in, and p also participates in some other checkpoints such as C that do not include q as its member agent. In this situation, agent p will delay the service of the rollback request till the completion of the checkpoint C. □

Lemma 4: A state interval from which output is committed will be recovered.

Proof: This follows from lemma 1 and 2. The crashed and rollback agents will re-execute and follow the same run as they did before the failures to their pre-failure states. So, the state interval from which an output is committed will be recovered. □

3.6 Mixture of Independent Checkpoint with Group Checkpoint

An agent may participate in several common goals in its life. Hence it may appear in several milestone dependency graphs (one graph for each common goal), and each graph may have several milestones selected for triggering checkpoints. Therefore, dynamically

an agent may take several group checkpoints or independent checkpoints for different goals. The group checkpoints and independent checkpoints of an agent may interleave in its lifeline and create more complex situations. However, after further analysis, we find that an independent checkpoint can be viewed as a group checkpoint with only the agent itself in the group. With this interpretation, independent checkpoint becomes a special case of group checkpoint, and theoretical results and experimental support can be applied to both scenarios of use.

Chapter 4 Case Study: E-trading System

4.1 Introduction

We have described the role-milestone based methodology and the SCLR protocol. In this chapter, we will use an e-trading application to demonstrate how to use the methodology in a multi-agent system. In the following sections, we will describe an e-trading system, elaborate the role model and agent model for the system, and apply the role-milestone based methodology to get the milestone dependency graphs that help to locate the checkpoint places and identify the checkpoint groups.

4.2 An E-trading Multi-Agent System

An e-trading system is designed to help customers and merchants to carry out transactions online. From the perspective of customers, the system provides services for searching and purchasing products through bidding or negotiation. From the perspective of merchants, the system allows them to publicize and sell their products through negotiation or online auction. In the system, merchant agents reside in malls, and customer agents reside in customer sites and interact with merchant agents through message exchanges. Below, we will describe some typical scenarios in the system, build the role model to capture the social aspects of the system in a high level abstraction, and build the agent model by mapping roles to agents according to the mapping strategy proposed in our role-milestone based methodology.

4.2.1 Basic Scenarios

The objective in this exercise is to demonstrate how to build a fault tolerant multi-agent system using our methodology and rollback-recovery protocol. Three basic scenarios in the system as described below are incorporated in our design:

(1) Product information search

Before a customer decides on a purchase, she has to collect relevant product information. Therefore, merchants have the responsibility to provide accurate and complete product information, and customers can collect the information by querying the merchants.

(2) Price negotiation

The system provides a negotiation mechanism for customers and merchants. Negotiation is a flexible way to trade. During the negotiation, customers and merchants change their prices, and finally they may both accept a negotiated price or fail to reach an agreement. If an agreement is reached, customers can book the products, and do the actual payment later in the product purchase process.

The negotiation strategies the system provides include contract net and English auction. FIPA has specified the two negotiation strategies as FIPA iterated contract net interaction protocol [FIPAICN] and FIPA English auction interaction protocol [FIPAEAIP].

Contract net is a strategy for buying and selling goods. A contract net is started by an initiator who sends out a call for proposals to its participants. Each participant views the request and may make a proposal. The initiator may choose the best proposal and award a contract to that participant and reject others, or the initiator may reject all the proposals. In the original contract net, initiator calls for proposals only once. However, the contract net protocol can be extended to allow multiple iterations.

English auction is a single-item and ascending-bid auction, where a set of bidders compete by increasing the current bid. The product sold in the auction is displayed to the bidders. Each bidder is allowed to place a bid that must increase the current bid by more than a minimal increment. The product is sold to the bidder who placed the highest bid when the auction ends.

(3) Product purchase

After customers and merchants negotiate successfully, customers have to place orders and finalize the product purchases. The system should provide rapid and safe purchase services that can supply standardized receipts to customers.

4.2.2 Role Model

Many role models have been built in the multi-agent trading domain according to the role model theory of Kendall [Kendall00]. An example is the distributed marketplace and institutional auction role models proposed in the Zeus Agent Building Toolkit by Collis [Collis00]. The role model for our E-trading system follows the Zeus role models. However, instead of using an arrow to represent collaboration between roles, we extend the role model of Kendall in order to describe the relationships between roles more precisely. In our role model, relationships between roles are captured by edges and labels. There are three types of edges: a one-way directed edge, a bi-directed edge, and an undirected edge. The undirected edge models a static relationship that persists throughout, and the directed edge models an abstract action (dynamic relationship) that may be initiated by one role. The asymmetry of the edge is used to represent the initiator of the action, so that the difference between a one-way directed edge and a bi-directed edge is that in bi-directed edge the action can be initiated by either role. The label attached to an edge is used to show the name of the relationship that the edge represented. The role model for our E-trading system is shown in figure 4-1.

There are three parties in the system: customer, store, and mall. A customer involves five roles: customer manager, searcher, buyer, bidder and payer. The customer manager represents a customer and governs the other four roles with a one-to-one relationship. A store involves four roles: seller, auctioneer, doorman, and accountant. A mall includes the roles of mall manager and broker. A mall provides the place for stores to reside in and manages these stores. Each mall has only one agent playing the role of mall manger. The

same applies to the broker role. However, a mall may have multiple agents playing the roles of seller, auctioneer, door-man, and account. We describe each role with its responsibilities, collaborators, and goals. We describe a relationship between roles with its initiator, participators, and description. The roles and their relationships in the E-trading system are described in the following.

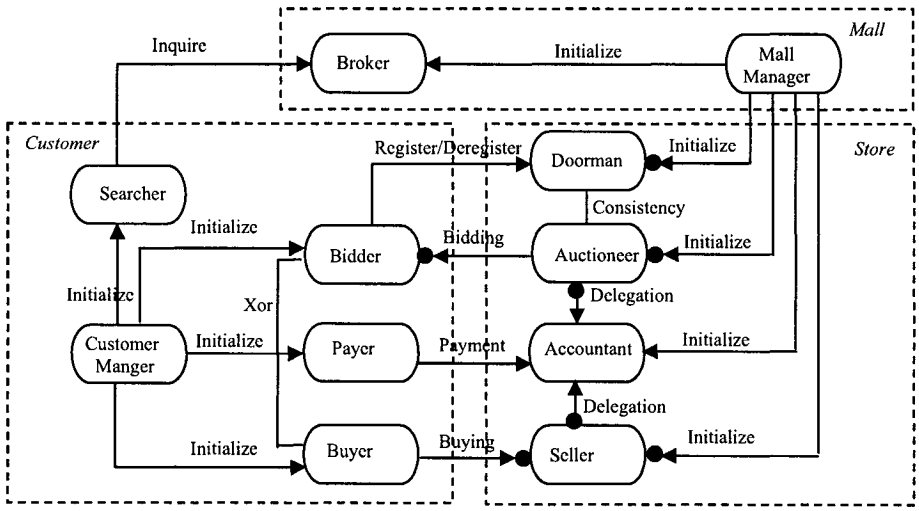


Figure 4-1: Role model for e-trading system

Roles:

(1) Customer Manager

- *Responsibilities*: A customer manager represents a customer. It receives commands from a user. Then it manages the purchasing process by assigning different tasks to an appropriate searcher, buyer, bidder, or payer. Finally it receives results and communicates them back to the user.
- *Collaborators*: Its collaborators include searcher, buyer, bidder, and payer.
- *Goals*: Its main goal is to manage the purchase request from a user.

(2) Searcher

- *Responsibilities*: It receives the product features that customer manager has interest in. Then it asks the broker for product and seller information. After getting the information, it informs customer manager of the result.
- *Collaborators*: Its collaborators include customer manager, and broker.
- *Goals*: Its main goal is to collect product and seller information from malls.

(3) Buyer

- *Responsibilities*: It receives the information about the product that a customer manager wants to buy, and the sellers that the customer manager wants to negotiate with. Then it negotiates with these sellers to get a good discount for the product using the iterated contract net protocol. Finally it informs the customer manager of the result.
- *Collaborators*: Its collaborators include customer manager and seller.
- *Goals*: Its main goal is to buy the product with the best price that is acceptable for the user.

(4) Bidder

- *Responsibilities*: It receives the information about the product that the customer manager wants to bid for, and the auctions in which the customer manager wants to participate. Then it bids for a product through the English auction protocol. Finally it informs the customer manager of the result.
- *Collaborators*: Its collaborators include customer manager, doorman, and auctioneer.
- *Goals*: Its main goal is to book the product with an acceptable price for the user.

(5) Payer

- *Responsibilities*: It receives the information about products that the customer manager wants to purchase, and the accountants that the customer manager has to interact with for the payment. Then it interacts with an accountant to pay for the products. Finally it informs the customer manager of the result.

- *Collaborators*: Its collaborators include customer manager and accountant.
- *Goals*: Its main goal is to make payments for products that the user wants to purchase.

(6) Mall Manager

- *Responsibilities*: It creates the merchant agents, and manages the merchant information.
- *Collaborators*: Its collaborators include broker, seller, auctioneer, and accountant.
- *Goals*: Its goals include creating the merchant agents in the mall and updating the merchant information.

(6) Broker

- *Responsibilities*: It receives queries from a searcher. Then it generates responses to be returned to the searcher.
- *Collaborators*: Its collaborator is the searcher.
- *Goals*: Its main goal is to provide correct information for searchers.

(8) Seller

- *Responsibilities*: It negotiates with buyers using the iterated contract net protocol. If the negotiation succeeds, it delegates the accountant role to do the payment.
- *Collaborators*: Its collaborators include buyer and accountant.
- *Goals*: Its main goal is to sell products to buyers.

(9) Doorman

- *Responsibilities*: It maintains the membership of the bidders participating in the auction. It also informs the auctioneer about the bidder membership. A bidder must register with the doorman before it can bid, and a bidder must de-register with the doorman before it can quit the auction.
- *Collaborators*: Its collaborators include bidder and auctioneer.
- *Goals*: Its main goal is to update the bidder list in an auction.

(10) Auctioneer

- *Responsibilities*: It conducts an English auction to sell a product. If the auction ends successfully, it delegates the accountant to finalize the settlement.
- *Collaborators*: Its collaborators include bidder and accountant.
- *Goals*: It main goal is to sell the product to one bidder.

(11) Accountant

- *Responsibilities*: It receives payment requests from payers. Then it generates receipts. Finally it informs payers about the receipt and delivery information.
- *Collaborators*: Its collaborator is payer.
- *Goals*: Its mail goal is to provide receipts and delivery information to payers.

Relationships:

(1) Initiate

- *Initiator*: Its initiator is customer manager
- *Participators*: Its participator is searcher
- *Description*: The customer manager creates and provides a product list to the searcher. The searcher sends the result to the customer manager.

Many other "initiate" relationships exist in the role model, such as "initiate" relationships between the customer manager and the buyer, the mall manager and the broker, etc. All these "initiate" relationships are similar in the sense that the initiator role creates the participator role, and provides the initial data. We ignore these similar parts to avoid repetition.

(2) Inquiry

- *Initiator*: Its initiator is searcher.
- *Participators*: Its participator is broker.

- *Description*: The searcher requests the broker for information about sellers and auctioneers who sell the searched products. The broker sends the answers to the searcher.

(3) Buying

- *Initiator*: Its initiator is buyer.
- *Participators*: Its participator is seller.
- *Description*: The buyer negotiates with sellers using the iterated contract net protocol.

(4) Register

- *Initiator*: Its initiator is bidder.
- *Participators*: Its participator is doorman.
- *Description*: The bidder requests for participation of an auction. The doorman confirms or rejects the bidder for the registration.

(5) De-register

- *Initiator*: Its initiator is bidder.
- *Participators*: Its participator is doorman.
- The bidder requests for permission to quit an auction. The doorman confirms or rejects the bidder for the quitting.

(6) Bidding

- *Initiator*: Its initiator is auctioneer.
- *Participators*: Its participator is bidder.
- *Description*: The auctioneer starts the auction using the English auction protocol.

(7) Delegation

- *Initiator*: Its initiator is buyer or auctioneer.
- *Participators*: Its participator is accountant.
- *Description*: The buyer or the auctioneer sends purchase requests to the accountant.

(8) Payment

- *Initiator*: Its initiator is payer.
- *Participators*: Its participator is accountant.
- *Description*: The payer requests the payment for purchasing of products. The accountant returns the receipt and delivery information to the payer.

(9) Consistency

- *Participators*: Its participators include doorman and auctioneer.
- *Description*: The doorman and auctioneer keep their bidder list consistent in their lifetimes.

(9) Xor

- *Participators*: Its participators include buyer and bidder.
- *Description*: The buyer and bidder will never buy the same product for a user, which avoids the duplication purchase of the same product.

4.2.3 Agent Model

Agents are the basic entities running in the system. The roles that an agent plays have to be identified during design. In chapter 2, we have proposed the mapping strategy between roles and agents, which includes two rules to guide the role assignment. Now we exercise those rules to identify agents in the system and the roles assigned to them.

According to the two rules, normally an agent should play a single role. However, there is one exception: two roles should be mapped to the same agent if they share a high volume of message traffic for information sharing. In our role model, auctioneer and doorman roles share high information and they can map to the same agent. All the other roles either share less information, or they could not be mapped to the same agent. Thus, there are ten types of agents in the system to play the eleven roles. The ten agents are: customer manager agent, searcher agent, buyer agent, bidder agent, payer agent, mall manager

agent, broker agent, seller agent, auctioneer agent, and accountant agent. Each agent plays its corresponding role, except that the auctioneer agent plays both the doorman and auctioneer roles.

4.3 Apply Role-Milestone Based Methodology

The essential part of the role-milestone based methodology is the milestone dependency graph, which helps to identify the social groups of the agent system and the suitable places for starting checkpoints. Five steps are recommended to get these graphs from the system role model and agent model. In this section, we demonstrate how to follow the five steps to get the milestone dependency graphs, and to analyze the graphs to get the social groups and the checkpoint placements.

4.3.1 Step 1: List Common Goals and Milestones

The first step is to list the common goals, milestones for roles. Common goals are the common interests of groups of roles. If a group of roles with a common interest must cooperate with each other to finalize the common interest (and therefore achieve their individual goals), we say the common interest is the common goal of the group of roles. Milestones of a role are the important points that a role must reach in its journey towards its goal. Milestones are defined by system designers with their own knowledge, and different designers may define different milestones. However, there are conventions for designers to follow. For example, a milestone may be an intermediate goal used to evaluate the progress towards the final set target, a scheduling event that signifies the completion of a major deliverable or a set of related deliverables, a flag indicating the completion of a part of project that may be needed by some specific time, a key event that defines the end of a phase or reaching a target or goal, or a scheduled event to measure progress, etc.

By analyzing the eleven roles and their relationships in our e-trading role model, we find some common goals such as exchanging information between searcher and broker, making deal prices between buyer and seller, or bidder and auctioneer, making payment transactions between payer and accountant, etc. Some relationships such as "delegation" between seller and account, "initiate" between customer manager and searcher, etc., are more like work assignment than cooperation. In order to conform to our methodology, groups of roles with such relationships are viewed to have simple common goals named as work assignments.

The milestones of each role are defined by the designer according to his knowledge. To avoid repetition of demonstration, we only describe the milestones of buyer, seller, auctioneer, and bidder. Buyer role has the individual goal of booking a product with the lowest acceptable price. To fulfill the goal, a buyer starts an iterated contract net protocol with several sellers selected in the information collection stage. Under this situation, defining a milestone as a scheduled event to measure progress seems a better choice. Therefore, a buyer can have several milestones to measure the progress of the price negotiation, such as "50% negotiation time has elapsed", "negotiation finishes", and "book a product successfully". If a buyer has a pre-defined deadline time for negotiation, "50% negotiation time has elapsed" means that half of the allowable time has passed. If buyer has a pre-defined number of rounds for requesting for proposals, "50% negotiation time has elapsed" means that the seller has requested for proposals for half of the round number. The seller role has the individual goal of selling products with the highest possible prices. Instead of starting a negotiation, a seller waits for the 'call for proposal' from buyers. Once a seller receives a request for a product, it sends its proposed price to the buyer. A seller ends a negotiation when it receives an "accept" or a "cancel" request. The selection of milestones for a buyer can be scheduling events signifying the completion of a major deliverable. Thus, the milestones of a seller are: "finish one

negotiation with a buyer" that indicates the end of one negotiation, and "send the booking record to a buyer" that indicates the successful sale of one product. In a similar way, the milestones for an auctioneer are: "reach the 75% of the minimum acceptable price for auctioneer", "reach the minimum acceptable price", "finish the auction", and "send the booking record to the winner bidder". The milestone for a doorman is "update bidder list successful". And the milestones for a bidder are: "register successfully", "negotiation ends", and "receives booking record".

4.3.2 Step 2: Identify the Loosely-Coupled or Closely-Coupled Agents

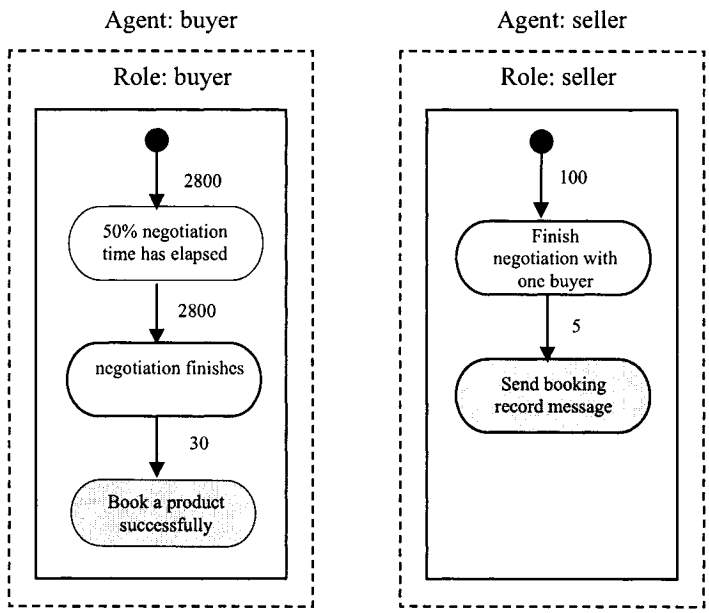
After selecting the goals and the milestones, the next step is to identify the loosely-coupled or closely-coupled agents of a common goal. As described in subsection 2.3.1, groups of agents collaborate to achieve their common goals. If many messages are exchanged among the group of agents towards their common goal, the group of agents is closely-coupled. On the contrary, if few messages are exchanged, the group of agents is loosely-coupled. The average number of messages that each agent in the group exchanges for achieving the common goal can be used to classify the two relations. For example, if a number of n agents exchange a number of m messages to achieve their common goal, m divides n (m/n) is the average number of messages that each agent exchanges to achieve the common goal. With the average number (represented as N) of messages exchanged among the group of agents, we can define a threshold. For a group of agents, if N is more than or equal to the threshold, the group of agents is closely-coupled; otherwise, they are loosely-coupled.

In our E-trading system, buyer and seller agents have the common goal of making a deal. To reach the goal, a buyer agent starts an iterated contract net protocol by sending request messages to seller agents, asking for the proposal price. A seller agent, upon receipt of the request, sends the proposal message with its proposal price. Upon receiving the

proposals, the buyer agent may choose the best price or reject all of them and start another round of requests. The elapsed time bound on iterations of the iterated contract net protocol can be specified by the customer before starting the buying process, and the maximum number of sellers that a buyer negotiates with is chosen as five in our system. With an average number of request round of five, the average number of messages that each agent exchanged during the collaboration process is around twelve. In the auction case, bidder and auctioneer agents collaborate to achieve the common goal of making a deal through the auction process. Each participating bidder agent has to send a register request to the auctioneer agent for registering into the auction first. When the auction starts, the auctioneer agent first informs the bidder agents of the start. Then it announces the current bid to all the bidder agents. Each bidder agent, in the receipt of the announce message, replies with a proposal message carrying its bid price. The auctioneer agent, upon the receipt of all the proposals, increases the highest bid it gets and announces the new bid again. The auctioneer repeats the announce-bid process until the end condition is reached. The auctioneer may sell the product to the bidder who places the highest bid or refuse all the bidders if its lowest acceptable price is not met. Normally in an auction, the bid price starts from a very low original price and end with tens of times of the original price. Therefore, the bidding round in an auction process is huge, which results in hundreds or thousands of messages exchanged among the auctioneer and bidder agents. So, in our system, we estimate that the average number of messages that each agent exchanges is more than forty. If we define the threshold as fifteen, then buyer agent and seller agent are loosely-coupled to reach their common goal; and bidder agent and auctioneer agent are closely-coupled while collaborating to achieve their common goal.

4.3.3 Step 3: Individual Milestone Dependency Graph for Loosely-Coupled Agents

As mentioned in subsection 2.3.3.1, log-based rollback recovery works better for loosely-coupled agents. Once an agent with log-based strategy fails, the agent will be recreated with its latest checkpoint and replay its logged messages. The computation lost is the individual agent effort from the failure point to the recent checkpoint of that agent. Therefore, we use individual milestone dependency graphs to capture the individual efforts of loosely-coupled agents to select milestones that trigger independent checkpoints. An example of the individual milestone dependency graph for buyer agent and seller agent are given in figure 4-2, and each agent should have a separate individual milestone dependency graph.



(a): buyer agent (b): seller agent

Figure 4-2: Individual milestone dependency graph for the common mission: making a deal price for a product

4.3.4 Step 4: Global Milestone Dependency Graph for Closely-Coupled Agents

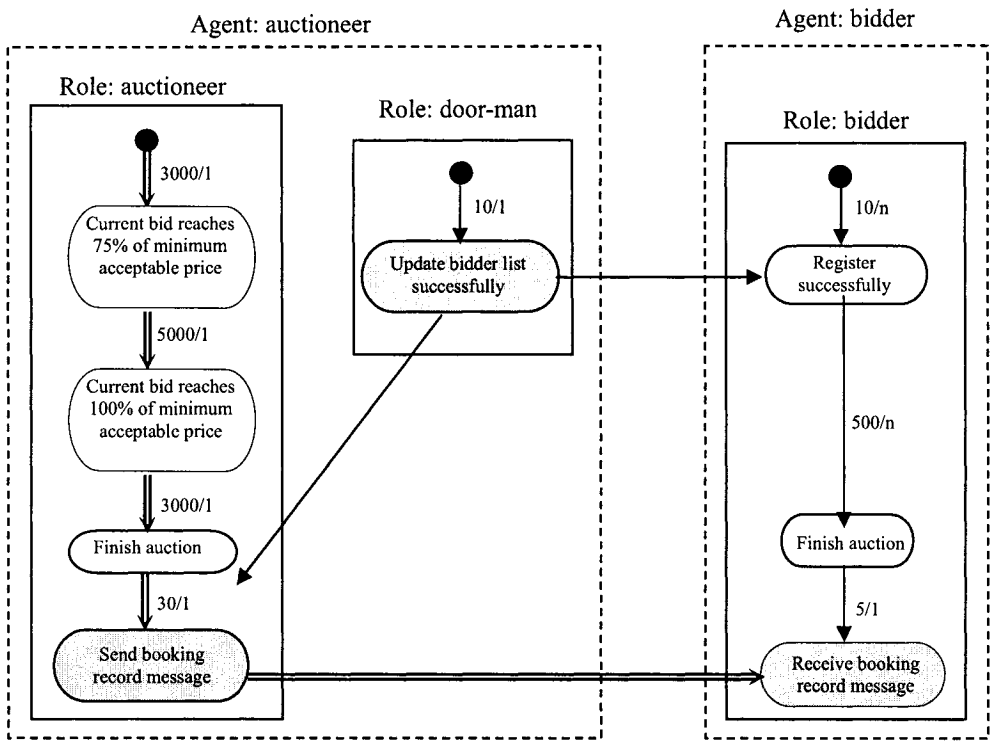


Figure 4-3: Global milestone dependency graph for the common mission: making a deal price for the bidding product

As mentioned in subsection 2.3.3.4, coordinated checkpointing works better for closely-coupled agents. In coordinated checkpointing, an event can trigger a set of checkpoints to be taken together among a set of agents to form a consistent state so that it can be used for recovery when a later failure occurs. Selection of milestones for triggering coordinated checkpoints should be based on global incremental effort because in the occurrence of a failure, the global incremental effort represents the maximum re-execution required during recovery. Therefore, we use global milestone dependency graphs, which capture the global effort for closely-coupled agents, to identify the social

groups and select the milestones for starting selective checkpointing. An example of the global milestone dependency graph for auctioneer agent and bidder agent are given in figure 4-3, and the global milestone dependency graph includes all the related agents.

4.3.5 Step 5: Estimate Incremental Effort

After the milestone dependency graphs are drawn, the next step is to estimate the incremental efforts of arrows inside the same role in the graphs. The incremental effort of two adjacent milestones of a role can be measured by running the agent who plays the role and recording the running time (milli-second) required from the statement corresponding to the previous milestone to the statement corresponding to the next milestone. Because different sample data may lead to different results, we run the system with typical sample data, and use the typical running time as the estimated incremental effort. For individual milestone dependency graph, the estimated incremental efforts can be labeled directly on their corresponding arrows since each individual milestone dependency graph represents an agent. However, global milestone dependency graph is for calculating global agent efforts towards milestones, and more than one agent is involved. Therefore, the number of agents contributing towards a milestone should also be labeled on the arrows. Details can be seen in subsection 2.3.2.3. Figure 2-3 has already labeled the estimated incremental efforts. Figure 2-4 labels both the incremental efforts and the number of agents that are involved. However, as the number of bidders involved in the milestones in figure 4-3 is variable and difficult to estimate, we use n to indicate that the number is unsure.

4.3.6 Milestone Selection and Group Identification

With one labeled individual milestone dependency graph, we can calculate the individual agent effort of each milestone in the graph. The individual agent effort of one milestone is the sum of all the incremental efforts attached to the arrows pointing towards the

milestone in the individual milestone dependency graph. After calculating the individual agent effort of each milestone, all the milestones in the graph can be ordered. Then, a threshold is defined so that every two milestones in the order with their incremental agent effort beyond the threshold can be selected to start independent checkpoints. The threshold is defined to be the tradeoff between failure-free running performance and recovery speed. Therefore, according to the individual milestone dependency graphs in the figure 4-2, the individual efforts for the milestones of the buyer agent are 2800, 5600, and 5630 respectively; and the efforts for the milestones of the seller agent are 100, 105 respectively. If we define the threshold as 2000, the milestones "50% negotiation time has elapsed" and "negotiation finishes" of the buyer agent are selected to trigger the independent checkpoints of the buyer agent. To the seller agent, it has no milestone in its graph that is beyond the threshold. However, one seller agent sells more than one product to more than one buyer agent in its lifetime. Therefore, once the seller agent sells every twenty products, the effort of its milestone "send booking record message" accumulates to be more than 2000, which can be selected to trigger an independent checkpoint of the seller agent.

With the labeled global milestone dependency graph, we can calculate the global agent effort of each milestone in the graph. The global agent effort of one milestone is the sum of the products of the incremental effort and the associated number of agents attached to the arrows pointing towards the milestone (refer to subsection 2.3.3.3). By ordering the milestones with their global agent efforts, we can define a threshold that every two milestones in the order with their incremental global agent effort beyond the threshold can be selected to start selective checkpoints. The threshold is defined to be the tradeoff between failure-free running performance and recovery speed. However, in the case that the global agent effort of a milestone cannot be estimated, the critical (longest effort) path in the global milestone dependency graph should be used for milestone selection. By

measuring incremental effort along this critical path, milestones can be selected to trigger new group coordinated checkpoint in the group of agents. Moreover, only agents playing the roles inside the global milestone dependency graph have to take part in the coordinated checkpoint and therefore form one checkpoint group. In figure 4-3, the number of auctioneer agents in the collaboration is one, and the number of bidder agents is variable and may vary significantly from an auction to another. Thus, due to the difficulty to estimate the global effort, we use the critical/longest path in the graph to perform the milestone selection. The longest path is shown by the double-lined arrows in the graph, and the milestones along this longest path have the incremental efforts of 3000, 8000, 11000, 11030. If we define the threshold as 3000, the milestones "Current bid reaches 75% of minimum acceptable price", "Current bid reaches 100% of minimum acceptable price" and "Finish auction" of the auctioneer agent are selected to trigger the group coordinated checkpoints. Auctioneer agent who plays the auctioneer and doorman roles should start the group checkpoints when the selected milestones are reached, and the corresponding bidder agents should participate in the group checkpoints. As register and de-register protocols are used between the doorman and the associated bidders, the auctioneer agent who plays the doorman role can maintain a bidder list that includes all the bidder agents to be involved in a group checkpoint.

4.4 Summary

We have described a sample e-trading system, built the role model, and demonstrated the steps to apply the role-milestone based methodology to the sample application. Readers are expected to be able to follow the step-by-step procedure provided in the methodology to refine their knowledge of the underlying application and apply this methodology to their applications.

Chapter 5 Implementation of Fault Tolerant E-trading System

5.1 Introduction

In this chapter, we will describe the implementation of our SCLR protocol on the e-trading multi-agent system. Because the log-based protocol and the mixture of the log-based protocol with the SCLR protocol are not the focus of this thesis, we will not describe their implementations. Our objective is to show how to implement a fault tolerate multi-agent system using the SCLR protocol in practice. Moreover, the implemented system can serve as a test bed for evaluating the SCLR protocol performance. In the following, we will introduce JADE (Java Agent Development Framework) to support the development of the multi-agent system, and describe the implementation details of the agent architecture proposed in chapter 2, and the four algorithms of the SCLR protocol.

5.2 JADE Platform

JADE (Java Agent Development Framework) is a software development framework aimed at developing multi-agent systems conforming to FIPA standards. In the system level, JADE is a platform (also a middleware) that provides runtime support for application agents. In the application level, JADE defines a framework to facilitate the development of multi-agent systems. It defines an agent model from which an agent developer can extend the abstract model to domain components by following a specific business logic. We now introduce JADE based on its version 2.61.

5.2.1 JADE Architecture

A JADE platform is composed of containers, which are distributed in machines. A container is actually a virtual host, which runs on one Java virtual machine and provides runtime support for agent execution. Main container is the core container of a JADE platform, and it coordinates all other containers and keeps together the whole platform. By using containers, agents running in containers are distributed in different machines, and they will not observe the existence of the underlying network as containers hide such complexity. Figure 5-1 shows the distributed JADE Architecture.

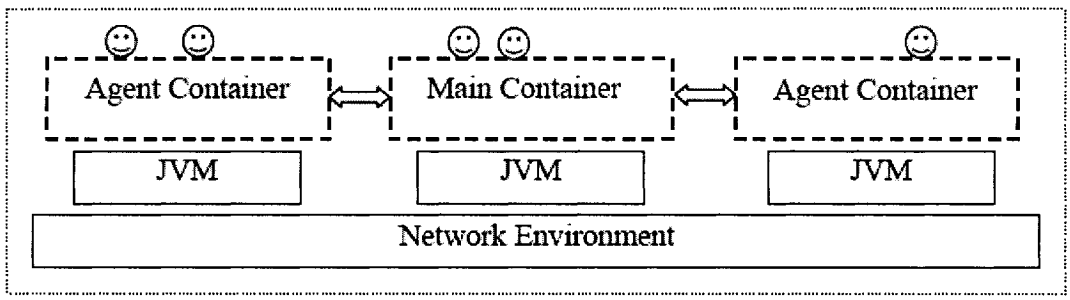


Figure 5-1 JADE architecture

5.2.2 JADE Agent Model and Behavior Model

A JADE agent is an active object, which adopts a thread-per-agent concurrency model. However, normally one agent has multiple concurrent activities (roles). Thus, to support the execution of an agent, JADE comes with a scheduler that schedules the behaviors of an agent in a non-preemptive way. Statically, a behavior is an abstract class that exposes an interface called action. A behavior of an agent is modeled as a sequence of actions taken from the behavior object. An agent naturally can have multiple behaviors. At runtime, the scheduler, which is embedded inside an agent and hidden to the agent developer, will take the various behavior objects available and execute them in a round-robin manner. However, the scheduler cannot save the stack frame of a behavior object. This means that once a behavior object is executed, it will not yield its execution resource

to another behavior object until it returns from its execution. By adopting the thread-per-agent strategy, JADE aims to limit the number of threads running in the agent platform and to minimize the thread overhead.

5.2.3 JADE Message Passing

JADE agents can communicate via asynchronous FIPA-compliant message passing. To send and receive messages, the JADE agent class provides a set of methods such as *Agent.send*() method and *Agent.receive*(). Each agent has a private message queue, which buffers the messages from the sender agents. The advantage of providing each agent with its own message queue is to reduce the synchronization among agents in receiving messages.

5.3 Implementation of Agent Architecture

The agents in the e-trading system are built with the agent structure proposed in chapter 2, and the agent architecture is a composition of roles that have their own structures. Based on the JADE agent model and behavior model, we can implement the proposed role structure and agent structure by extending the JADE agent class and behavior class.

A role can be defined as a class, which consists of the role knowledge and three behaviors: sensory behavior, reactive behavior, and proactive behavior. Sensory and reactive behaviors can be viewed as JADE cyclic behaviors which run cyclically in the lifetime of an agent. Proactive behavior is a behavior that has finite states that simulate all the status of the role while achieving its individual goal. Each state of the proactive behavior includes a set of programs that achieve a specific task. Proactive behavior finishes its work once the states corresponding to the end of the behavior are reached. A role has only one proactive behavior associated with it. However, as one agent may cyclically repeat one type of work, such as an auctioneer agent may cyclically auction

products one by one, proactive behavior, at the time of achieving the end of the behavior, may restart from the beginning and try to do another work.

With the role class to implement a role, the implementation of the agent structure is simple. An agent is defined as a class extending the JADE agent class, which consists of agent level knowledge and the pointers pointing to the roles it plays. Thus, according to the JADE behavior model, one agent has all the behaviors belonging to the roles it plays. Figure 5-2 shows a sample of an agent class and a role class played by the agent. In the figure, "Agent 1" maintains a "role 1" pointer that points to the "role 1" that "Agent 1" plays.

```
public class Agent1 extends JadeAgent {

    Role1 role1;                                    // a pointer linking to the role the agent plays
    Knowledge agentKnowledge;

    public Agent1() { //construction
        role1 = new Role1(this);
    }
} // end of class Agent1

public class Role1 {

    Agent myAgent;                                  // a pointer linking to the agent who plays the role
    Knowledge roleKnowledge;
    ProactiveBehavior PB;

    public Role1(Agent agent) {                     //construction
        myAgent = agent;
        myAgent.addBehaviour(new SensoryBehavior(myAgent));
        myAgent.addBehaviour(new ReactiveBehavior(myAgent));
        PB = new ProactiveBehavior(myAgent);
        myAgent.addBehaviour(PB);
    }

    public abstract class SensoryBehavior extends JadeCyclicBehaviour{
        SensoryBehavior (Agent agent);
        public void action() {                      // perceive action of the role
            .....
        }
    }

    public abstract class ReactiveBehavior extends JadeCyclicBehaviour{
        ReactiveBehavior (Agent agent);
        public void action() {                      // reactive action of the role
            .....
        }
```

```
}

public abstract class ProactiveBehavior extends JadeSimpleBehaviour {
    int state;
    ProactiveBehavior (Agent agent) {          //construction
        super(a);
        state = 0;                             // 0 indicates the initial state of the actions
    }
    public void action() {                     // proactive actions of the role
        switch (state) {
        case 0:                                //start the proactive behavior
            ...
            state = 1;
            break;
        case 1:
            .....
        case n:                                // n indicates the end state of the actions
            ...
            exit;
        } // end of swtich
    } // end of action

    public int onEnd() {                       //restart the proactive behavior once it ends
        state = 0;
        myAgent.addBehaviour(PB);
    }
} // end of ProactiveBehavior class
} // end of class Role1
```

Figure 5-2 Agent class and the role class the agent plays

We have implemented the proposed role-based agent structure; however, to make the agent fault tolerant, fault tolerant mechanisms have to be integrated into the above-mentioned classes to implement the SCLR protocol. Details are given in the next section.

5.4 Implementation of the SCLR Protocol

To make the e-trading system fault tolerant, we provide four servers to provide the services to facilitate the implementation of the rollback-recovery protocol. The servers can be implemented with replication techniques to guarantee their availability. The servers include: (i) a repository server to provide application agents the service of saving their checkpoints and logs; (ii) a failure detection server to provide the service of monitoring the liveliness of application agents; (iii) a recovery server to provide the

service of building the recovery line, recreating/restoring the rollback agents with their checkpoints and logs; (iv) a garbage collection server to remove the useless checkpoints and logs. With the four servers, we can implement the checkpointing algorithm, logging algorithm, recovery algorithm and pruning algorithm of the SCLR protocol easily.

In addition to the servers, fault tolerant mechanisms also have to be integrated into application agents to make them fault tolerant. In a fault tolerant application agent that conforms to our agent architecture, there is application program code for achieving the application specific operations, and fault tolerant program code for implementing the SCLR rollback-recovery protocol. Code for starting a checkpoint should be inserted after the code corresponding to a selected milestone; code for logging events should be inserted before events are handled; and code for restoring the state of a recovery agent and relaying its logged events should also be provided.

In the following, we will first introduce the terminologies, data structures and classes used in the system. After that we will elaborate the details about the four servers that run in the JADE platform as system agents. Then we will describe the fault tolerant code that is integrated into application agents. Finally we will elaborate the implementation of the algorithms of the SCLR protocol described in chapter 3.

5.4.1 Terminology, Data Structure and Classes

The terminologies used in the system include AID, CID, and SI. AID indicates the identifier of an application agent. CID indicates the identifier of a group checkpoint, and all individual checkpoints of a group checkpoint have the same CID. SI indicates the index of the current state interval of an agent.

The data structures used in the system include CHECKPOINT, CG, CGS, EVENT, SSN and RSN. The CHECKPOINT structure maintains the data of an individual checkpoint of an agent, which includes CID, AID, SI, SSN, RSN, CG, and DATA (refer to subsection

3.4.3). CID and AID form the key of the CHECKPOINT structure, which can uniquely locate an individual checkpoint. The CG structure is encapsulated in the CHECKPOINT structure, and it maintains CG (refer to subsection 3.4.1) of its corresponding checkpoint. The CG structure is a set of pairs "<AID, STATUS>", where AID indicates the identifier of the agent who cooperated to take the checkpoint and STATUS indicates whether the corresponding checkpoint of the cooperated agent is removable or not (refer to subsection 3.4.4). The CGS structure maintains CGS (refer to subsection 3.4.1) of an agent, which is a set of CG. The SSN structure maintains the SSN table described in subsection 3.4.2 of an agent, which is a set of pairs "<AID, NUM>", where AID is the identifier of the receiver agent with which the agent has communicated and NUM maintains the sequence number of the message sent to its corresponding receiver agent. We use SSN(AID) to indicate the corresponding sequence number of a receive agent. The RSN structure maintains the RSN table described in subsection 3.4.2 of an agent, which is also a set of pairs "<AID, NUM>", where AID is the identifier of the receiver agent with which the agent has communicated and NUM maintains the highest sequence number of the message received from its corresponding receiver agent. We use RSN(AID) to indicate the corresponding highest sequence number of a receive agent. Details about SSN and RSN can be found in subsection 3.4.2. The EVENT structure maintains the determinant of an event, which includes SOURCEAID, DESTAID, SI, SSN(DESTAID), DATA, and TYPE. SOURCEAID and DESTAID denote respectively the identifiers of the sender agent and the receiver agent. TYPE indicates that an event may be "message receipt event", "input event", or "decision event". SSN(DESTAID) indicates the sequence number attached to an event if its TYPE is "message receipt event" (refer to subsection 3.4.5).

Two classes are provided in the system to maintain checkpoints and logs saved by application agents. They are *CheckpointTable* class and *LogTable* class. *CheckpointTable*

class provides the methods such as *saveCheckpoint(c:CHECKPOINT)* to save a specific checkpoint to stable storage, *getCheckpoint(aid:AID, cid:CID)* to read a specific checkpoint of an agent from stable storage by AID and CID, *removeCheckpoint(cid:CID)* to remove a group of checkpoints for pruning, *getLatestCheckpoint(aid:AID)* to read the latest checkpoint of an agent from stable storage, and *getCGS(aid:AID)* to calculate CGS of an specific agent from its checkpoint history. *LogTable* class provides the methods such as *saveLog(e:EVENT)* to save a specific event of an agent to stable storage, *getLogs(c:CHECKPOINT)* to read events saved after an individual checkpoint of an agent from stable storage, and *removeLogs(c:CHECKPOINT)* to prune events saved before a individual checkpoint for pruning. These methods are used by the repository server, the recovery server, and the garbage collection server to maintain the data shared among them.

5.4.2 Implementation of Servers

The repository server maintains the checkpoints and messages/events saved by application agents that run in the system. It also informs application agents of their current CGS once they take new checkpoints. Application agents can save their checkpoints or events by sending "checkpointing" messages tagging the CHECKPOINT data or "logging" messages tagging the EVENT data to the repository server. The repository server, upon receipt of such messages, saves the CHECKPOINT/EVENT data on stable storage. After a checkpoint is saved, the repository server also informs the garbage collection server to do the pruning because pruning should be done after new checkpoints are created (refer to in subsection 3.4.4). Moreover, by using the centralized solution, the repository server can work as the coordinator of a group checkpoint in the way that it sends the "release" messages to the participator agents of a group checkpoint to complete the group checkpoint once it receives all the individual checkpoints from its

members. Before sending a "release" message to an agent, the repository server should calculate CGS of that agent according to its checkpoint history, so that CGS can be tagged to the "release" message to inform the destination agent about its current CGS.

The failure detector server is responsible for monitoring the liveliness of application agents. To do so, application agents have to register with the failure detector server at their creation time, and de-register before they vanish. The failure detector server sends "ping" messages to the registered application agents periodically. An application agent, upon receipt of a "ping" message, will reply with an "alive" message. An application agent will be regarded as in the 'crashed' state if the failure detector cannot receive its "alive" message within a bounded time. Once the failure detector server finds an application agent that has crashed, it informs the recovery server by sending a "failure" message tagging the AID of the crashed agent.

The recovery server is responsible for determining the subset of agents to be recovered, recreating the crashed agent with its latest checkpoint and logs, and resetting the states of the survivor agents involved in recovery with their corresponding checkpoints and logs. Upon the receipt of a "failure" message from the failure detector server, the recovery server will calculate CGS of the crashed agent according to its checkpoint history, recreate the crashed agent with its latest checkpoint and logs, load the checkpoints and logs of the survivor agents belonging to ICGS of the crashed agent (refer to subsection 3.4.1), roll them back with their corresponding checkpoints and logs, and finally restart all the involved agents. The recovery server works as the recovery coordinator that implements the three-phase recovery protocol described in subsection 3.4.6. Figure 5-3 shows the skeletal code of the recovery server and the *getCGS(a:AID)* method of the *CheckpointTable* class. The *getCGS()* method calculates CGS of an agent according to its checkpoint history, and the recovery server implements parts of the recovery algorithm. In the figure, we use *C* to indicate a CHECKPOINT data, use *C*.SI, *C*.SSN, and *C*.RSN to

indicate the data of *C*. To complete the work of recovery, some fault tolerant codes have to be inserted into application agents, which will be described later in subsection 5.4.3.4.

```
CheckpointTable:getCGS(agentid:AID) {
        CGS ← Φ
        For each checkpoint c taken by agent a with a.AID = = agentid
                CGS ← CGS ∪ c.CG
        Return CGS
}

Recovery server receives one failure(agentid) message from failure detector server {

        //Get CGS of the crashed agent
        CGS ← CheckpointTable.getCGS(agentid)
        ICGS ← Intersection of CGS

        //Load the most recent checkpoint of the crashed agent
        C ← CheckpointTable.getLatestCheckpoint(agentid)

        //Load logs saved after checkpoint C of the crashed agent
        LOG ← LogTable.getLogs(C)

        Recreate the crashed agent with the same agentid, C and LOG

        For each agent q (q.AID∈ICGS and q.AID ≠ agentid)
                C ← CheckpointTable.getCheckpointByCID(q.AID, C.CID)
                LOG ← LogTable.getLogs(C)
                Send "rollback" message to agent q tagging C and LOG

        For each agent q: (q.AID∈ICGS)
                Block to receive ready(C.CID) message from q

        For each agent q: (q.AID∈ICGS)
                Send commit(C.CID) message to q
}
```

Figure 5-3 Code for the recovery server

The garbage collection server implements the pruning algorithm described in subsection 3.4.4 that deletes those checkpoints and logs that are no longer needed in any agent crash. We have explained in subsection 3.4.4 that the pruning algorithm should be called once new checkpoints are created. Therefore, once a checkpoint is saved into stable storage successfully, the repository server should inform the garbage collection server to run the pruning algorithm. The garbage collection server and the repository server can be integrated in the same system agent to simplify the communication between them.

5.4.3 Integrate Fault Tolerate Mechanism into Application Agent

In addition to the servers, fault tolerant code has to be integrated into agent application code to achieve fault tolerant agents using the SCLR protocol. This fault-tolerant code inserted into agents cooperates with the four servers to implement the algorithms of the SCLR protocol. As the pruning algorithm is implemented totally in the garbage collection server, we will display how fault tolerant code is inserted into agent programs to implement the other three algorithms: selective checkpointing algorithm, logging algorithm, and recovery algorithm. In the following, we will first describe the *FTAgent* class, which extends the JADE agent class and encapsulates the data structure and method needed by a fault tolerant agent. Then we will elaborate the implementation of the algorithms by inserting fault tolerant code into the agent application code.

5.4.3.1 FTAgent Class

To implement the SCLR protocol, an agent has to maintain some protocol related data structures, which include SI, CGS, SSN, RSN (refer to subsection 3.4.2), and a list that stores the logged events during recovery. An agent also has to provide some methods such as *saveCheckpoint(cid:CID, cg:CG, data:DATA)* to save its checkpoint to the repository server, *saveEvent(e:EVENT)* to save its event to the repository server, and *restoreState(c:CHECKPOINT, events:List, cgs:CGS)* to restore its state and logged events for recovery. Moreover, to implement the logging and recovery algorithms, the methods for receiving and sending messages provided by the JADE agent have to be modified to be able to handle the message logging and replay. *FTAgent* class is defined to extend the JADE agent class and encapsulate the data structures and methods mentioned above. Therefore, an application agent in the e-trading system can extend the *FTAgent* class to inherit the data structures and methods needed for implementing the SCLR protocol. Figure 5-4 shows the *FTAgent* class. In the diagram, *Message* class is provided

by JADE, *send(m:Message, a:AID), receive(),* and *put(m:Message)* methods are provided by JADE agent class to send a message to the receiver agent, deliver a message to the agent from its message queue, and put a message to the end of the agent's message queue respectively. The *receiveMessage(m:Message)* and *sendMessage()* are user defined methods that implement the message logging and message replay recovery protocol and should be called by fault tolerant agents to deliver and send a message. An input event of an agent can be modeled as a message receipt event by properly designing the agent class; therefore, it works with the same mechanism of a message receipt event except that all input events should be logged. The fault tolerant code regarding a decision event should be inserted at the place where the event occurs, which are mixed with the application code and will be explained later in subsection 5.4.3.3.

```
public class FTAgent extends JadeAgent {
    int si;
    CGS cgs;
    SSN ssn,
    RSN rsn;
    List logs;
    AID repository_server, recovery_server;

    public void saveCheckpoint(cid:CID, cg:CG, data:DATA) {
        CHECKPOINT c = new CHECKPOINT(cid, this.AID, si, ssn, rsn, cg. data);
        send(c, repository_server);                    //send c to the repository server
    }

    public void saveEVENT(sourceaid:AID, type:TYPE, data:DATA) {
        switch (type) {
        case MESSAGE_RECEIPT:
            event = new EVENT (sourceaid, this.AID, si, ssn(destaid), data, type);
            break;
        case INPUT_EVENT:
            event = new EVENT (sourceaid, this.AID, si, null, data, type);
            break;
        case LOCAL_EVENT:
            event = new EVENT (null, thisAID, si, null, data, type);
            break;
        }
        send(event, repository_server);                //send event to the repository server
    }

    public void restoreState (c:CHECKPOINT, events:List, cgs_in:CGS) {
        si = c.SI;
        ssn = c.SSN;
```

```
        rsn = c.RSN;
        logs = events;
        cgs = cgs_in;
    }

    public Message sendMessage(m:Message) {          // Message class is provided by JADE
        ssn(m.receiverAID)++;
        send(m, m.receiverAID);
    }

    public Message receiveMessage() {
        if (logs.size() > 0)
            e = logs.getElementAt(0);
            if si = = e.si -1
                logs.removeElementAt(0);
                rsn(e.sourceAID)++;
                return e;                            // repay the logged message
            else
                m = receive();
                if rsn(m.senderAID) = = m.ssn -1
                    rsn(m.senderAID)++;
                    return m;                        // re-exchange the un-logged message
                else if rsn(m.senderAID) < m.ssn -1
                    putback(m);
        else
            m = receive();
            if rsn(m.senderAID) = = m.ssn -1
                rsn(senderAID)++;
                if (cgs ≠ null) and (m.senderAID ∉ Intersection of cgs)
                    saveEvent(m.senderAID, MESSAGE_RECEIPT, m) // log the message
                return m;
            else if rsn(m.senderAID) < m.ssn -1
                putback(m);
    }
}
```

Figure 5-4 Code for FTAgent

5.4.3.2 Implementation of Selective Checkpointing Algorithm

The JADE platform does not provide functions to save and reload the stack of an agent. Therefore, recovering from a checkpoint can only depend on the states of the logical control variables that can locate the actual source statement associated with a checkpoint. According to a thread-per-agent model and the behavior model of JADE, if the state variable of each behavior in an agent is saved in a checkpoint, the agent can be recreated and restarted from the exact execution place by restoring these state variables. In our

agent architecture, as sensory behavior and active behavior are simple behaviors that have no states, only the states of proactive behaviors of the roles that an agent plays have to be saved in a checkpoint. Thus, the application related data in a checkpoint include the agent-level knowledge, the role-level knowledge, and the state variables of proactive behaviors. During recovery, these data are restored and the proactive behaviors of roles can be restarted from their saved states.

As we have explained in chapter 2, checkpoints are started at the time when selected milestones are reached. With the agent structure we proposed, reaching a milestone could be designed as reaching a state of a proactive behavior. Therefore, with the recovery consideration we mentioned above, code for starting a checkpoint should be inserted in a proactive behavior at the beginning of the application code. Moreover, as a participator agent of a group checkpoint can only start its checkpoint at the time of receiving a marker message, the receiving message event should only occur at the first statement of each state of a proactive behavior. Figure 5-5 shows the code of a fault tolerant auctioneer agent, and the code shown from line 47 to 56 demonstrates the checkpoint part of the agent.

5.4.3.3 Implementation of the Logging Algorithm

Logging and relaying messages have been implemented in the *sendMessage()* and *receiveMessage()* methods of the *FTAgent* class as shown in the figure 5-4. Therefore, an application agent class inherits the ability of logging messages as it extends the *FTAgent* class. However, the code for logging the decision event is inserted at the place where the event occurs, which is mixed with the application code. Figure 5-5 shows a sample of logging a decision event (line 59, 64, and 65) in the proactive behavior of the auctioneer agent.

5.4.3.4 Implementation of Recovery Algorithm

The recovery server has implemented most parts of the recovery algorithm as shown in the figure 5-3. However, some fault tolerant code has to be inserted into the agent programs to cooperate with the recovery server. As indicated in figure 5-4, *FTAgent* class has implemented the message replay, which is inherited by application agents. The task left for an application agent is to provide a *construction(c:CHECKPOINT, logs:List, cgs:CGS)* method to recreate a crashed agent with its checkpoint, logs and CGS, provide a *rollback(c:CHECKPOINT, logs:List, cgs:CGS)* method to roll back a survivor agent with its checkpoint, logs and CGS, and insert the fault tolerant code at the place where a decision event might occur for relaying the logged decision event. The construction and rollback methods of an application agent should send READY messages to the recovery server to implement the three-phase recovery protocol described in subsection 3.4.6. Figure 5-5 shows the code of the construction (line 9-20) and rollback (line 21-31) methods of the auctioneer agent, and the code for the reply of a decision event (line 59-63) in the proactive behavior of the auctioneer agent.

```
1 public class AuctioneerAgent extends FTAgent {
2     RoleAuctioneer auctioneer;          // a pointer linking to the auctioneer role the agent plays
3     RoleDoorman doorman;                // a pointer linking to the bidder role the agent plays
4     Vector bidderlist;                  // agent-level knowledge

5  public AuctioneerAgent() {             //construction
6     auctioneer = RoleAuctioneer(this);
7     bidder = RoleBidder(this);
8  }

     //construction for recreation of the crashed agent
9  public AuctioneerAgent(c:CHECKPOINT, logs:List, cgs:CGS) {
10    auctioneer = RoleAuctioneer(this);
11    bidder = RoleBidder(this);
12    bidderlist = c.data.agentknowledge;                    // restore the agent-level knowledge
13    auctioneer.roleknowledge = c.data.auctioneer.kowledge; //restore the role-level knowledge
14    bidder.roleknowledge = c.data.bidder.kowledge;         // restore the role-level knowledge
15    auctioneer.state = c.data.auctioneer.state;      // restore the state of auctioneer proactive behavior
16    bidder.state = c.data.bidder.state;              // restore the state of the bidder proactive behavior
17    restoreState (c, logs, cgs);

18    Message ready = new Message(READY, c.CID);
```

```
19    sendMessage(ready, recovery_server); // send ready message
20  }

    // roll back the survivor agent during recovery
21  public viod rollback (c:CHECKPOINT, logs:List, cgs:CGS) {
22    bidderlist = c.data.agentknowledge;                    // restore the agent-level knowledge
23    auctioneer.knowledge = c.data.auctioneer.kowledge;     //restore the role-level knowledge
24    bidder.knowledge = c.data.bidder.kowledge;             // restore the role-level knowledge
25    auctioneer.state = c.data.auctioneer.state             // restore the state of auctioneer proactive behavior
26    bidder.state = c.data.bidder.state                     // restore the state of the bidder proactive behavior
27    restoreState (c, logs, cgs)                            // restore the protocol related data

28    Message ready = new Message(READY, c.CID);
29    sendMessage(ready, recovery_server);                   // send ready message
30  }
31 }


32 public class RoleAuctioneer {

33     Agent myAgent;                                        // a pointer linking to the agent who plays the role
34     Knowledge roleKnowledge;
35     ProactiveBehavior PB;

36  public RoleAuctioneer(Agent agent) {
       ...
37     PB = new ProactiveBehavior(myAgent);
38     myAgent.addBehaviour(PB);
39  }
    .....
40  public abstract class ProactiveBehavior extends JadeSimpleBehaviour {
41      public int state;
42      ProactiveBehavior (Agent agent) {
43          super(a);
44          state = 0;
45      }

46      public void action() {

            // initiator group checkpoint
47          if (state == selected_miletones_state)
48              for each bidder in myAgent.bidderlist
49                  String cid= myAgent.Name + SI;                          // create CID
50                  CG cg = new CG(bidderlist+auctioneer.AID);             // create CG
                    // create the marker message attaching CID and CG
51                  Message marker = new Message(marker, cid, cg);
52                  sendMessage(marker, bidder);                           // send marker message
53                  DATA data = new DATA(bidderlist, auctioneer.roleknowledge,
                                         bidder.roleknowledge, auctioneer.state, bidder.state);
54                  saveCheckpoint (cid, cg, data);     // store the checkpoint to the repository server
                    //wait for the release message from repository message
55                  Message release = receiveMessage(RELEASE);
56                  cgs = release.CGS;                                     // set the current CGS of the agent
            // end for group checkpoint

57          switch (state) {
```

```
        ...
58      case n:
            ...
            // At the place where a decision event may occur
59          if (logs,size() >0)
60              e=logs.getElementAt(0);                    // relay the logged event during recovery
61              if si = e.si -1 and e.TYPE = LOCAL_EVENT
62                  replay the event;
63                  logs.removeElementAt(0);
64          else
                //in the occurrence of a decision event e in a normal execution, log the event
65              saveEVENT(this:AID, LOCAL_EVENT, e.DATA);
            ...
66          exit;
67      } // end of switch
68  } // end of action
    ...
69 } // end of ProactiveBehavior class
70 } // end of RoleAuctioneer
```

Figure 5-5 Code for auctioneer agent

5.5 Summary

We have demonstrated the implementation of our SCLR protocol using a centralized solution on the JADE platform. Obviously, there are many possible different implementations. Readers are expected to understand better the underlying protocols and implications to their implementations and performance.

Chapter 6 Performance Test

6.1 Introduction

In the fault tolerant e-trading system, we have implemented the SCLR protocol and the log-based protocol. In this chapter, we use the e-trading system as the test case to evaluate the performance of these two protocols. The evaluation consists of three parts. The first part compares the percentage of messages logged in the two protocols. The second part evaluates the failure-free performance of the two protocols. The last part compares their recovery speeds. The first two tests are related to the cost/overhead of the protocols in a failure free run, and the last one is to evaluate the recovery performance when failures occur. Our results are obtained by averaging the results measured over ten runs of the e-trading system.

6.2 Experimental Setup

We conducted our experiment on 4 Pentium-based workstations connected by 100Mb/s Ethernet. Each workstation has 256 megabytes of memory and runs Windows 2000 server. In the experiment, each machine hosts one JADE container, which can simulate one mall site or one customer site. We use 3 machines to simulate 3 mall sites, and each mall includes one broker agent, three seller agents, one auctioneer agent, and one accountant agent. The fourth machine is used to simulate the customer site, which includes 4 customer manager agents, 4 searcher agents, 4 buyer agents and 30 bidder agents. Searcher and bidder agents move to the mall sites to complete their missions. Thus, in one auction scenario, we have 10 bidder agents working with 1 auctioneer agent. In an iterated contract net negotiation, we have 3 seller agents negotiating with each buyer agent.

6.3 Percentage of Messages Logged

Log-based rollback recovery protocol has the advantage of domino-free rollback and fast output commit. However, it has to log every message in the system, which introduces heavy runtime overhead and may impede the speed of the underlying application. The SCLR protocol avoids domino rollback without logging all messages. This can improve the failure-free performance of the rollback-recovery protocol. The percentage of messages logged reflects the improvement in overhead reduction when the SCLR protocol is used.

We use the auction scenario to get the percentage of messages logged in an auction. According to our system design, auctioneer and bidder agents participate in group checkpoints during their efforts to achieve their common goal of making a deal (refer to subsection 4.3.6). The auctioneer agent starts the group checkpoints at the three selected milestones: "Current bid reaches 75% of minimum acceptable price", "Current bid reaches 100% of minimum acceptable price" and "Finish auction". As ICGS of an agent (refer to subsection 3.4.1) determines whether a messages should be logged or not, the checkpoint history can significantly affect the percentage of messages logged. We use N_a to indicate the number of auctioneer agents involved in the achievement of the common goal, N_b to indicate the number of bidder agents involved, N_{cp1} to indicate the number of agents participating in the first group checkpoint, N_{cp2} to indicate the number of agents participating in the second group checkpoint, and N_{cp3} to indicate the number of agents participating in the third group checkpoint. Six test cases are simulated to see how checkpoint history influences the test result. Table 6-1 presents the percentage of messages logged against messages received by the auctioneer agent in the SCLR protocol in six different cases. In the six cases, N_a and N_b have fixed values: $N_a = 1$ and $N_b = 10$. N_{cp1}, N_{cp2}, and N_{cp3} are variable, which show the checkpoint history of the participating agents.

Logged message/exchanged message (%)						AVG
Case 1: N_{cp1}=11 N_{cp2}=11 N_{cp3}=11	*Case 2:* N_{cp1}=8 N_{cp2}=10 N_{cp3}=11	*Case 3:* N_{cp1}=7 N_{cp2}=9 N_{cp3}=11	*Case 4:* N_{cp1}=5 N_{cp2}=9 N_{cp3}=11	*Case 5:* N_{cp1}=4 N_{cp2}=8 N_{cp3}=11	*Case 6:* N_{cp1}=3 N_{cp2}=6 N_{cp3}=11	
15	27	35	47	61	78	44

Table 6-1 Percentage of messages logged against messages received

Analysis:

The logging strategy used in the SCLR protocol is that for an agent *A*, only those messages received by *A* and sent by agents not belonging to ICGS(*A*) have to be logged (refer to subsection 3.4.1). Therefore, the bigger the set of ICGS(*A*) is, the smaller the percentage of messages logged. Furthermore, group checkpoints are triggered at the achievement of the three selected milestones: "Current bid reaches 75% of minimum acceptable price", "Current bid reaches 100% of minimum acceptable price" and "Finish auction", which depend on how the bidding process goes on. As long as there are bidder agents offering competing prices, the selected milestones can be reached no matter how many bidder agents are involved in the bidding. Moreover, the time when a bidder agent registers with an auctioneer agent is unscheduled, which makes the groups of agents in a group checkpoint undeterminable. Due to the above mentioned reasons, the checkpoint groups of the three pre-built group checkpoints are variable, and ICGS of the participating agents are variable accordingly.

The six test cases show the different situations of the three checkpoint groups, and ICGS of participating agents varies in different cases. For example, in case 1 where $N_{cp1} = N_{cp2} = N_{cp3} = 11$, all 11 agents participate in each of the three group checkpoints. When a new group checkpoint is taken, the previous checkpoint is pruned. Therefore, ICGS of the auctioneer agent always contains 11 agents after the first checkpoint is taken, and no messages exchanged among the 11 agents have to be logged afterward. In case 2 where

$N_{cp1} = 8$, $N_{cp2} = 10$, $N_{cp3} = 11$, 8 agents are involved in the first group checkpoint. Thus, after the first checkpoint, ICGS of the auctioneer agent contains 8 agents and messages sent from the 3 bidder agents outside the ICGS have to be logged by the auctioneer agent. When the second group checkpoint is taken and the first group checkpoint is pruned, ICGS of the auctioneer agent comes to contain 10 agents and only messages from one agent outside the ICGS have to be logged. In the same way, after the third group checkpoint is taken, all 11 agents are contained in ICGS of the auctioneer agent and no messages received by the auctioneer agent from the 10 bidder agents have to be logged afterward. For case 3 to case 6, their situations are similar to what happens in case 2. Therefore, in a conclusion, ICGS of case 1 and case 2 are larger, so that in case 1 and case 2, the SCLR protocol logs fewer messages. In the other four cases, as ICGS becomes smaller and smaller, the protocol logs more and more messages.

The data in table 6-1 also shows that even in the best case, 15% of the messages received by the auctioneer agent have to be logged. The reason is that before a group checkpoint is done, all participating agents do not belong to any group and all messages have to be logged, which contribute to the 15% result.

6.4 Failure-Free Performance

Failure-free performance of a rollback-recovery protocol can be expressed by the execution overhead introduced to the application. The execution overhead can be measured by the percentage increase in runtime due to the additional services required to provide agent-crash fault tolerance. We can get the percentage increase in runtime by measuring the agent execution time with and without fault tolerance protocols. To compare the performance of the log-based protocol and the SCLR protocol, we implement the two protocols using the auction scenario, and run the group of auctioneer agent and bidder agents under the same condition, which means the number of auctioneer

and bidder agents participating in the auction is the same and each bidder agent joins the auction at the same time respectively in the two protocols. Table 6-2 presents the failure-free overhead for the SCLR protocol and log-based protocol. Under the SCLR protocol, we run the group of auctioneer agent and bidder agents with: $N_b = 10$, $N_a = 1$, $N_{cp1} = 5$, $N_{cp2} = 9$, and $N_{cp3} = 11$, and the percentage of messages logged against messages received is 41%.

Protocol	Execution time without Fault tolerance (ms)	Execution time with fault tolerance (ms)	Increase in runtime (%)
Group checkpoint	47966	50100	4.4
Log-based	47966	52136	8.7

Table 6-2 Failure-free overhead

Analysis:

From the data shown in table 6-2, the SCLR protocol increases only 4.4% of runtime to the application agent, while log-based protocol increases 8.7% of runtime. The difference 4.3% comes from the fact that 59% messages are reduced for recording due to the group checkpoint and group rollback recovery. The data shows that logging produces the most part of the overhead while checkpointing (group coordination checkpointing and independent checkpointing) produces less. The overhead produced by group coordination checkpointing is small because by using groups we can limit the number of agents participating in a group coordination checkpoint into a reasonable size. Moreover, the overhead for coordination checkpoint is further reduced as messages in channels during the group checkpoint are logged by the logging algorithm, which introduces this overhead to the logging algorithm not the checkpointiing algorithm. The above data: 4.3 (8.7 – 4.4) percent of performance improvement shows that the SCLR protocol improves the failure-free performance significantly compared with the log-based protocol. If the message traffic among the group of agents increases, the performance improvement

should be more dramatic. However, according to the test result, both the SCLR protocol and the log-based protocol introduce less overhead into the application. The reason is due to the fact that the Jade platform runs with many disk accesses even in ordinary applications. Hence the additional disk accesses caused by logging do not change the execution time profile as much as one might expect.

6.5 Recovery Speed

The recovery speed of a system can be expressed by the recovery time required to restore the system into a consistent global state from the time when a failure is detected. In the occurrence of an agent crash, the recovery time T_{rec} for the log-based protocol comprises three parts. (1) T_{chk} is the time to restore the state of the failed agent to its latest checkpoint. (2) T_{ack} is the time to retrieve determinants and messages logged of the failed agent during failure-free execution. (3) $T_{rollfwd}$ is the time to roll forward the execution of the failed agent to its pre-crash state. The recovery time T_{rec} for the SCLR protocol consists of five parts. (1) T_{chk} is the time to restore the states of the failed agent and the corresponding rollback agents. (2) T_{ack} is the time to retrieve determinants and messages logged of the failed agent and rollback agents during failure-free execution. (3) T_{3phase} is the time to incorporate the rollback recovery. (4) T_{replay} is the time to roll forward the execution of the failed agent to its pre-crash state by re-exchanging its un-logged events and replaying its logged events. (5) $T_{recover}$ is the time to roll forward the execution of other rollback agents to their pre-failure states by re-exchanging their un-logged events and replaying their logged events.

To compare the recovery speed of the log-based protocol and the SCLR protocol, we also use the auction scenario, and run the group of auctioneer agent and bidder agents under the same condition as shown in the failure-free performance study. We simulate an agent crash by crashing the auctioneer agent near the end of its mission, and test the time

needed to restore the groups of agents to their pre-failure states. Table 6-3 presents the recovery speed for the log-based protocol and the SCLR protocol.

Protocol	Total execution time (ms)	Recovery speed (ms)
Group checkpoint	50100	3983
Log-based	52136	815

Table 6-3 Recovery speed

Analysis:

From the data shown in table 6-3, the SCLR protocol recovers much slower than the log-based protocol. The difference comes from the fact that more agents have to roll back in the SCLR protocol than the log-based protocol. Therefore, the SCLR protocol needs more time in T_{chk} and T_{ack} than the log-based protocol. The SCLR protocol also requires extra time for T_{3phase} and $T_{recover}$. This result meets our expectation that the SCLR protocol uses more complex recovery algorithm than the log-based protocol to exchange for the benefit of better failure-free performance. However, the test result does not take into account the concurrency of the multi-agent rolling back. With the rollback concurrency, the recovery speed of the SCLR protocol can be further improved.

6.6 Summary

We have estimated the performance of the SCLR protocol by comparing it with the log-based protocol. The latency for the output commit is omitted here because no latency is required for the SCLR protocol as well as the log-based protocol. Our test results show that the SCLR protocol has better failure-free performance than the log-based protocol while maintaining the advantages of domino-free rollback, simple garbage collection, simple recovery protocol, and fast output commit. However, the first evaluation part: the percentage of messages logged against messages received shows that under the situation that agent groups in a system are stable, the SCLR protocol has better performance.

However, if agents join and leave groups frequently, the SCLR protocol could not produce the performance that we desired.

According to whether social groups of multi-agent applications are stable or not, we can divide multi-agent applications into strictly organized applications and loosely organized applications. With the test results, we can draw to the conclusion that for strictly organized applications with heavy communication, our SCLR protocol can significantly improve the performance compared with other rollback recovery protocols. As our sample E-trading system is not a typical strictly organized application, the test results we got here just show the average performance of the applicable multi-agent applications.

Chapter 7 Conclusion

7.1 Summary

Sociality is one of the most important features of a multi-agent system. Agents form societies within which they cross couple, depending on the roles they play. Using roles, different social groups can be identified. Interactions within social groups can be tracked based on a milestone model detailing the progress of a role in its mission. Using milestones as guides, a novel checkpoint with selective logging protocol (SCLR) is proposed and demonstrated to be effective. The methodology and protocol proposed give a new perspective to guide designers to build fault tolerant multi-agent systems.

The role-milestone based methodology can identify the checkpoint groups and suitable places for starting checkpoints. Milestone dependency graphs form the key basis in the methodology. A milestone dependency graph can be constructed from the system role model by following five simple steps. By analyzing the graph, decisions can be made as to how groups should be formed and where checkpoints should be inserted.

The methodology involves making decisions about groups and checkpoints at design time. This contrasts with decisions to be made at runtime. The latter, while possible, may incur more overhead as tracking of the progress of the communication will likely be required.

The SCLR protocol guarantees domino-free rollback and aims to optimize checkpointing with the designer's knowledge of the system design. Obviously both checkpoints and message logging incur runtime overhead and may delay the underlying application. The use of group checkpointing reduces the overhead by coordinating group checkpoints to reduce the message logging required among group members.

We designed and implemented an e-trading fault tolerant system on the JADE platform to demonstrate the use of the proposed methodology and its effectiveness. The test results show that the SCLR protocol performs well when agents in the multi-agent application form groups that do not change too often during their lifetime.

7.2 Contribution and Future Work

In this thesis, we report our efforts in considering multi-agent system features in the design of a rollback recovery protocol. We successfully designed an SCLR protocol, which makes use of the sociality of multi-agent systems to form groups and to take group checkpoints. We also proposed a role-milestone based methodology to facilitate the identification of social groups in a multi-agent system by providing a step-by-step procedure. The role-milestone based methodology and the SCLR protocol can help develop fault tolerant multi-agent systems with reasonable failure-free performance. However, further work is required to develop this methodology more fully by (i) identifying applications that may fit the role-milestone based checkpoint strategy better, (ii) incorporating the methodology on an agent platform to facilitate application design, and (iii) refining the methodology to cover a large spectrum of applications.

Bibliography

[Alvisi96] L. Alvisi "Understanding the Message Logging Paradigm for Masking Process Crashes", Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, New York, January 1996.

[Alvisi98] Lorenzo Alvisi and Keith Marzullo. "Message Logging: Pessimistic, Optimistic, Causal, and Optimal", *IEEE Transactions on Software Engineering*, 24(2):149-159, Feb. 1998.

[Bhargava88] B. Bhargava and S. R. Lian. "Independent Checkpointing and Concurrent Rollback for Recovery - An Optimistic Approach", *Proceedings of IEEE Symposium on Reliable Distributed Systems*, pp.2-12, 1988.

[Booch99] G. Booch, J.Rumbaugh, and I.Jacobson. "The Unified Modeling Language User Guide", Addison Wesley, 1999.

[Brooks86] R. Brooks. "A Robust Layered Control Systems for a Mobile Robot", *IEEE Journal of Robotics and Automation*, RA 2(1):14-23, 1986.

[Burmeister96] B. Burmeister. "Models and methodology for agent-oriented analysis and design", K. Fischer, editor, Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems, 1996.

[Cabri01] G. Cabri. "Role-based Infrastructures for Agents", *8th IEEE Workshop on Future Trends Distributed Computing System*, Bologna, Italy, Oct31-Nov 02, 2001.

[Cabri02] G. Cabri, L. Leonardi, and F. Zambonelli. "Modeling Role-based Interactions for Agents", Workshop: Agent-oriented methodologies, OOPSLA 2002, SEATTLE, WA, USA, pp.4-8, Nov. 2002.

[Chandy85] K. M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer Systems*, 3(1):63—75, Feb. 1985.

[Collis00] Jaron Collis, Divine Ndumu, "ZEUS ROLE MODELLING GUIDE", The Zeus Agent Building Toolkit, ZEUS Methodology Documentation Part I, 1999 British Telecommunications plc., Release 1.02, September 2000.

[Elnozahy02] E. N. Elnozahy, L. Alvisi, Yi-Min Wang, and D. B. Johnson. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems", *ACM computing Survey*, 34(3): 375-408, Sep. 2002.

[Elnozahy92] E. N. Elnozahy and W. Zwaenepoel. "Manetho: Transparent Roll back-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit", *IEEE Transactions on Computers*, 41(5) 526-531, May 1992.

[Elnozahy94] E.N. Elnozahy and W. Zwaenepoel. "On the Use and Implementation of Message Logging", *Proceedings of the Twenty Fourth International Symposium on Fault-Tolerant Computing (FTCS-24)*, pp.298—307, Jun. 1994.

[FIPA] Publicly Available Implementations of FIPA Specifications, http://www.fipa.org/

[FIPAEAIP] "FIPA English Auction Interaction Protocol Specification", Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00031/XC00031E.pdf

[FIPAICN] "FIPA Iterated Contract Net Interaction Protocol Specification", Foundation for Intelligent Physical Agents, http://www.fipa.org/specs/fipa00030/PC00030D.pdf

[Fischer97] K. Fischer and C. G. Jung. "A layered agent calculus with concurrent, continuous processes", In M. P. Singh, A. Rao, and M. J. Wooldridge (eds.), Intelligent Agents IV: Agent Theories, Architectures, and Languages, LNCS 1365, pp. 245-258, 1997.

[Giang02] N. T. Giang, D. T. Tung, "Agent Platform Evaluation and Comparison", Institute of Informatics of Slovax Academy of Science, June 2002.

[Gottlob96] G. Gottlob, M. Schrefl, and B. Rock. "Extending Object-Oriented Systems with Roles", *ACM Transactions on Information Systems*, 14(3):268-296, July, 1996.

[Grasshopper] http://www.grasshopper.de/

[Guessoum99] Zahia Guessoum and Jean-Pierre Briot. "From Active Objects to Autonomous Agents", *IEEE Concurrency*, 7(3):68-76, July-September 1999.

[Huang93] Y. Huang and C. Kintala. "Software Implemented Fault Tolerance: Technologies and Experience", *Digest of Papers (FTCS-23), the Twenty Third Annual International Symposium on Fault-Tolerant Computing*, pp.2-9, 1993

[Iglesias98] Carlos A. Iglesias, M. Garijo, and Jose C. Gonzalez. "A survey of Agent-Oriented Methodologies", *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, pp.317-330, 1998.

[Jennings01] Nicholas R. Jennings and Michael Wooldridge. "Agent-Oriented Software Engineering", J. Bradshaw, editor, Handbook of Agent Technology, AAAI/MIT Press, 2001.

[Johnson87] D.B. Johnson and W. Zwaenepoel. "Sender-Based Message Logging", *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing (FTCS-17)*, pp.14—19, Jun. 1987.

[Johnson91] D. B. Johnson and W. Zwaenepoel. "Transparent Optimistic Rollback Recovery", *Operating Systems Review*, pp.99—102, Apr. 1991.

[Kalaiselvi00] S. Kalaiselvi and V. Rajaraman. "A Survey of Checkpointing Algorithms for Parallel and Distributed Computers", *Sadhana*, 25(5):489-510, Oct. 2000.

[Kasbekar01] M. Kasbekar and Chita R. Das. "Selective Checkpointing and Rollbacks in Multithreaded Distributed Systems", *The 21st International Conference on Distributed Computing Systems*, Mesa, AZ, pp.39-46, April 2001.

[Kendall99-1] E. A. Kendall. "Role Model designs and Implementations with Aspect-Oriented Programming", *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications*, Denver, Colorado, United States, pp.353-369, 1999.

[Kendall99-2] E. A. Kendall. "Role Models – Patterns of Agent System Analysis and Design", *BT Technical Journal*, 17(4), October 1999.

[Kendall00] Elizabeth A. Kendall. "Role Modeling for Agent System Analysis, Design, and Implementation", *IEEE concurrency*, 8(2):34-41, April-June 2000

[Klein93] G. A. Klein, J. Orasanu, R. Calderwood and C.E. Zsambok (Eds.). "Decision Making in Action: Models and Methods", Ablex Publishing Corporation, Norwood, New Jersey, 1993.

[Koo87] Richard Koo and Sam Toueg. "Checkpointing and Rollback-Recovery for Distributed Systems", *IEEE Transactions on Software Engineering*, 13(1):23-31, Jan. 1987.

[Labrou97] Y. Labrou and T. Finin. "KQML as an Agent Communication Language", In Bradshaw. J., "Software agents", The MIT Press, 1997.

[Lesperance02] Yves Lesperance. "Introduction to Intelligent/Autonomous Agents and their Applications", COSC 6390A Knowledge Representation/ "Intelligent Agents", Fall 2002, http://www.cs.yorku.ca/course_archive/2002-03/F/6390A/slides/week1-2up.pdf

[Manivannan99] D. Manivannan and Mukesh Singhal. "Quasi-Synchronous Checkpointing: Models, Characterization, and Classification", *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703-713, Jul. 1999

[MASIF] MASIF-The Object Management Group Mobile Agent System Interoperability Facility. http://www.omg.org

[Miles01] Simon Miles, Mike Joy, and M. Luck. "Designing Agent-Oriented Systems by Analysing Agent Interactions", *First international workshop, AOSE 2000 on Agent-oriented software engineering*, pp.171 - 181, 2001

[Muller95] J. P. Muller and M. Pischel. "Modelling Reactive Behaviour in Vertically Layered Agent Architectures", *Intelligent Agent: Theories, Architectures, and Language (LNAY Volume 890)*, pp.261-276, Springer-Verlag: Berlin, Germany, 1995.

[Muller97] J. P. Muller. "The Design of Intelligent Agents: A Layered Approach", LNCS 1177, 1997.

[Nwana99] H. S. Nwana, D. T. Ndumu and L. C. Lee. "ZEUS: An Advanced Tool-Kit for Building Distributed Multi-Agent Systems", *Proceedings of the Third International Conference on Autonomous Agents*, 1999.

[Odell02] J. Odell, H. V. D. Parunak, M. Fleischer, and S. Brueckner. "Modeling Agents and Their Environment", *AOSE 2002*, pp.16-31. 2002

[Odell00] J.Odell, H. V. D. Parunak, and B. Bauer. "Extending UML for Agents", *Proceedings of the Agent-Oriented Information Systems (AOIS) Workshop at the 17th National conference on Artificial Intelligence (AAAI)*, 2000.

[Petrie01] Charles Petrie, "Agent-Based Software Engineering", Agent-Oriented Software Engineering, Lecture Notes in AI, Springer-Verlag 1957, 2001, pp.58-76. Stanford Networking Research Center, Stanford, CA 94305-2232.

[Pivk] Aleksander Pivk and Matjaž Gams. "Intelligent Agents in E-commerce", http://ai.ijs.si/Sandi/publications/IAinEC.pdf

[Plank97] James S. Plank. "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance", Technical Report(UT-CS-97-372), University of Tennessee, July 1997.

[Poggi00] A. Poggi and G. Rimassa. "Adding Extensible Synchronization Capabilities to the Agent Model of a FIPA Compliant Agent Platform", *Proceedings First International Workshop, AOSE 2000 on Agent-oriented software engineering*, pp.206 – 215, Limerick, Ireland, 2000.

[Poslad00] Stefan Poslad, Phil Buckle, and Rob Hadingham. "The FIPA-OS agent platform: Open Source for Open Standards", Nortel Networks. Manchester, UK. April 2000.

[Rana00] Omer F. Rana and Kate Stout. "What is Scalability in Multi-Agent Systems?", *International Conference on Autonomous Agents Proceedings of the fourth international conference on Autonomous agents*, pp.56 – 63, 2000.

[Rao91] Rao and M. Georgeff. "Modeling Rational Agents within a BDI Architecture", *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, MA, pp.473-484, 1991.

[Riehle98] D. Riehle and T. Gross. "Role Model Based Framework Design and Integration", *Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, ACM Press, pp.117-133, 1998.

[Shehory01] Onn Shehory, Arnon Sturm. "Evaluation of modeling techniques for agent-based systems", *Proceedings of the fifth international conference on Autonomous agents*, International Conference on Autonomous Agents, pp.624 – 631, 2001

[Shoham93] Y. Shoham. "Agent-Oriented Programming", *Artificial Intelligence*, 60(1):51-92, March 1993.

[Silva98] L. M. Silva and J. G. Silva. "System-Level versus User-Defined Checkpointing", *Proceedings of the Seventeenth Symposium on Reliable Distributed Systems*, pp.68—74, Oct. 1998.

[Storm85] R. Strom and S. Yemini. "Optimistic Recovery in Distributed Systems", *ACM Transactions on Computer Systems*, 3(3):204—226, Aug. 1985.

[Straßer97] M. Straßer, J. Baumann and F. Hohl. "Mole - A Java based Mobile Agent System", In M. Mühlhäuser: (ed.), Special Issues in Object Oriented Programming, dpunkt Verlag, pp.301-308, 1997.

[Sycara96] K. Sycara, A. Pannu, M. Williamson and D. Zeng. "Distributed Intelligent Agents". *IEEE Expert*, 11(6):36-46. 1996.

[Sycara98] Katia P. Sycara. "MultiAgent Systems", *AI Magazine*, 19(2):79-92, Summer 1998.

[Tveit01] Amund Tveit. "A survey of Agent-Oriented Software Engineering", The First NTNU Computer Science Graduate Student Conference, May 2001.

[Wang93] Y. M. Wang. "Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems", Ph.D. Thesis, University of Illinois Urbana-Champaign, August 1993.

[Wang95] Y. M. Wang, P. Y. Chung, I. J. Lin, and W. K. Fuchs. "Checkpoint Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems", *IEEE Transactions on Parallel and Distributed Systems*, 6(5):546—554, May 1995.

[Wood00] M. F. Wood and S. A. Deloach. "An Overview of the Multiagent Systems Engineering Methodology", *Proceedings of the first International Workshop on Agent-Oriented Software Engineering*, pp.207-220, June 2000

[Wooldridge00] M. Wooldridge, N. R. Jennings, and D. Kinny. "The Gaia Methodology for Agent-Oriented Analysis and Design", *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3):285-312, 2000.

[Wooldridge95] M. Wooldridge and N.R. Jennings. "Intelligent Agents: Theory and Practice", *The Knowledge Engineering Review*, 10(2):115-152, 1995.

[Wooldridge99] M.Wooldridge, N.R. Jennings, and D. Kinny. "A Methodology for Agent-Oriented Analysis and Design", *Proceedings Third International Conferance on Autonomous Agents (Agents 99)*, Seattle, WA, pp.69-76, May 1999.

[Xu03] Haiping Xu, "Multi-Agent System And Agent Based Software Engineering", CIS602: Advanced Software Engineering, December 11, 2003.

[Zambonelli98] F. Zambonelli. "On the Effectiveness of Distributed Checkpoint Algorithms for Domino-free Recovery", *7th IEEE Symposium on High-Performance Distributed Computing (HPDC-7)*, IEEE Computer Society Press, Chicago (IL), July 1998.

[Zeus] Zeus, http://more.btexact.com/projects/agents/zeus/