An Exception Handling Service for the Contract Net Protocol Family

by
Lijin Aryananda

B.S. Electrical Engineering
Massachusetts Institute of Technology, 1998

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 21, 1998
[June 1999]

©Copyright 1999 Lijin Aryananda. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author_____
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by_____
Chrysanthos Dellarocas
Douglas Drane Career Development Assistant Professor of Management
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY
JUL 1 5 1999
LIBRARIES

ARCHIVES

An Exception Handling Service for the Contract Net Protocol Family

by
Lijin Aryananda

Submitted to the
Department of Electrical Engineering and Computer Science

May 21, 1999

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Autonomous agents are systems that inhabit complex and dynamic environments. Attaining robust behavior in such complex conditions is a key challenge for agent-based systems. This challenge is intensified in multi-agent settings, where a diverse set of agents communicate and coordinate with each other, resulting in non-deterministic system behaviors. The standard approach to this problem relies on the notion that each agent should be equipped as much as possible in order to survive on their own in its environment, leading to several serious limitations. Dellarocas and Klein (1999) propose an alternative approach, using a generic exception handling service for providing adaptability by detecting and handling domain independent exceptions.
This thesis presents an evaluation and prototype implementation of this approach using a contract net setting. The feasibility of the approach is assessed, involving identification of contract net related domain independent exceptions and their handling strategies as well as an investigation of the minimum base level interfaces required from each agent. Experimental results showing a scenario in which agents relies on the exception handling service for dealing with agent death exception are demonstrated. Lastly, a thorough analysis and comparative study of the shared service approach versus other related works are presented.

Thesis Supervisor: Chrysanthos Dellarocas
Title: Douglas Drane Career Development Assistant Professor of Management

Acknowledgement

This thesis was made possible by the support of many people. To them, I would like to express my deepest gratitude.

Firstly, I would like to thank my advisors, Prof. Chris Dellarocas and Dr. Mark Klein for their kind support throughout the work involved in this thesis. I thoroughly enjoy working on this project and have learned a tremendous amount because of the remarkable insights and continuous guidance they have given me in the past year. I also want to thank Dave Shue for giving up lots of his time to help me and patiently answering my never-ending questions and Dr. Jeffrey Bradshaw for all his help and useful suggestions in developing the test bed. Special thanks go to Ji for supporting and encouraging me without fail; John, Livia, and Anto for always being there for me; and Carol for listening to my complaints and lending me her laptop for a whole month. Finally, I would like to dedicate this thesis to both my father and mother and thank them for being the most wonderful parents and believing in me all these years.

Table of Content

1. INTRODUCTION **7**

1.1 THE DYNAMIC ENVIRONMENT OF OPEN MULTI-AGENT SYSTEMS 7
1.2 THE SURVIVALIST APPROACH 9
1.2.1 Overview and Limitations *9*
1.2.2 Contract Net: an Agent Coordination Protocol *10*
1.3 THE GOOD CITIZEN APPROACH 15
1.4 THESIS GOAL 17
1.5 OUTLINE 17

2. OVERVIEW OF THE EXCEPTION HANDLING SERVICE **19**

2.1 REGISTRATION 21
2.2 FAILURE MODES MAPPING 22
2.3 SENTINELS FOR DETECTING AND ANTICIPATING EXCEPTION 24
2.4 DIAGNOSING EXCEPTION 24
2.5 RESOLUTION AND AVOIDANCE PLAN SELECTION 26

3. IMPLEMENTATION AND EVALUATION **29**

3.1 EVALUATING FEASIBILITY 29
3.1.1 Domain Independent Exceptions *30*
3.1.2 Detection, Resolution, Anticipation, and Avoidance Strategies *31*
3.1.4 Discussion *36*
3.2 PROTOTYPE IMPLEMENTATION AND SAMPLE EXPERIMENT 37
3.2.1 Test Bed: SimHazard + simple contract net agents *37*
3.2.2 Exception Handler *39*
3.2.3 Sample Experiment *42*
3.3 SUMMARY: EVALUATION OF ACHIEVEMENT OF GOALS 44

4. CONTRIBUTION TO RELATED WORK **45**

5. FUTURE WORK **48**

APPENDIX A **50**

APPENDIX B **53**

APPENDIX C **56**

C.1 AGENT DESIGN 56
C.2 SIMPLE CONTRACT NET PROTOCOL 57
C.3 AUGMENTED (SURVIVALIST) CONTRACT NET PROTOCOL 58
C.4 MESSAGE TYPES AND FIELDS 59

C.5 TASK STRUCTURE 60
REFERENCES **61**

List of Figures & Tables

FIGURE 1. STRUCTURE OF MULTI-AGENT SYSTEMS 8
FIGURE 2. THE CONTRACT NET PROTOCOL 11
FIGURE 3. THE CONTRACT NET PROTOCOL + IMMEDIATE RESPONSE BIDS 13
FIGURE 4. CONTRACT NET PROTOCOL + TASK EXECUTION MONITORING 15
FIGURE 5. FUNCTIONAL ARCHITECTURE 20
FIGURE 6. REGISTRATION PROCESS 21
FIGURE 7. AN EXAMPLE OF A CONTRACT NET VARIANT 22
FIGURE 8. FAILURE MODE IDENTIFICATION AND SENTINEL CREATION 23
FIGURE 9. DIAGNOSIS PROCESS 25
FIGURE 10. CAUSAL-EFFECT RELATIONSHIP OF AGENT DEATH EXCEPTION 26
FIGURE 11. RESOLUTION PROCESS 27
FIGURE 12. TEST BED: SIMHAZARD + AGENTS 38
FIGURE 13. EXCEPTION HANDLER DESIGN 41
FIGURE 14. EFFECT OF A "SOCIAL MONITORING" INSTITUTION ON THE COMPLETION DELAY OF SUPPLY CHAINS WHERE AT LEAST ONE SUBCONTRACTOR AGENT UNEXPECTEDLY FAILS 43

TABLE 1. DETECTION, RESOLUTION, ANTICIPATION, AND AVOIDANCE STRATEGIES 32
TABLE 2. HYPOTHESIZED ADVANTAGES OF THE DOMAIN INDEPENDENT EXCEPTION HANDLING STRATEGIES 34
TABLE 3. QUERY AND ACTION INTERFACES 36

Chapter 1

Introduction

1.1 The Dynamic Environment of Open Multi-Agent Systems

Despite the absence of a single universally accepted definition of software agent, many agree on the notion that autonomous agents inhabit complex and dynamic environments. (Maes 1995) defines autonomous agents as systems that inhabit a dynamic, unpredictable environment in which they try to satisfy a set of time-dependent goals or motivations. (Shoham 1990) adds by pointing out that the agents' environments is often inhabited by other agents. (Jennings, Sycara and Wooldridge 1998) describes agents are software systems, which are capable of interacting with other agents in a flexible and autonomous way, in order to meet the design objectives of their creators.

Attaining robust behavior in such unpredictable environments and bounded resources is a key challenge for agent-based systems. This challenge is intensified in an open multi-agent setting, in which a diverse set of agents communicate and coordinate with each other, resulting in non-deterministic system behaviors. Figure 1 portrays the structure of a multi-agent system, consisting of three layers: infrastructure, agent, and coordination. In the infrastructure layer, communication channels may fail; causing lost and garbled messages. The individual agent layer may also be effected by the environment, resulting

in failures related to performing domain specific tasks. For example, a sudden change in weather conditions may create hazardous situations if it is undetected by the air-traffic controller agent. Agents may also break down, be defective, or simply malicious. The coordination layer is an abstract entity encapsulating the interactions and interdependencies among the agents in the system and just as fragile as the other two layers. Agents may enter and exit without notice. Unanticipated agent inter-dependencies leading to conflicts and circular deadlocks are yet another concern. Clearly, exceptions in any of these layers may propagate and generate other exceptions in other layers. Failure to handle exceptions in any of these layers will raise systemic problems, such as clogged networks, inefficient resource allocation, poor performance, system shutdowns, and security vulnerabilities (Dellarocas and Klein 1999).

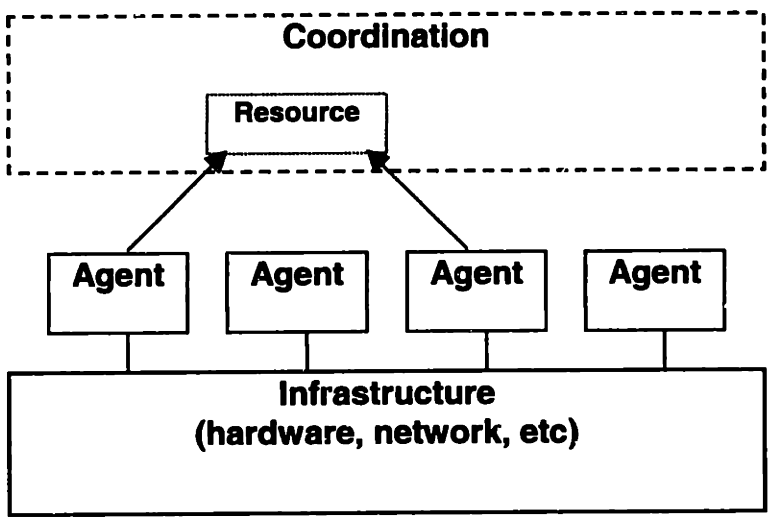


Figure 1. Structure of Multi-Agent Systems

It is important to note that these exceptions can be distinguished into two categories: domain dependent and independent. Domain dependent exceptions correspond to exceptions related to the performance of agent's domain specific tasks, whereas domain

independent exceptions are the ones that can be detected and resolved without any domain specific knowledge. Dealing with a sudden change in weather conditions or an incoming aircraft with abnormal speed capabilities clearly requires the air traffic controller agent's domain knowledge, thus these exceptions belong to the domain dependent class. On the other hand, lost messages between the air traffic controller and pilot agents are considered domain independent because they can be detected and handled without any domain expertise.

1.2 The Survivalist Approach

1.2.1 Overview and Limitations

Since the mid 1980s, several efforts have been made to design adaptive and robust agent architecture, each focusing on different aspects. Some researchers chose to focus on developing single agent architectures and utilized various kinds of individual planning (Georgeff and Lansky 1987, Laird et.al. 1987, Doyle 1996) to deal with exceptions in the agent layer. Others have been concentrating on the coordination layer and developing sophisticated negotiation protocols (Sierra et.al. 1997; Sandholm and Lesser 1995; Conry et.al. 1986; Davis and Smith 1983), multi-agent decision making algorithms (Stirling 1992; Tenney and Sandell 1981), or collaborative planning techniques (Lesser 1998; Decker and Lesser 1992; Lansky 1991; Georgeff 1987) with the goal of ensuring coherent interactions among agents.

Despite differences in methodology, all of these approaches rely on the common notion that each agent should be equipped as much as possible in order to survive *on their own*

in its environment, requiring sophisticated reasoning capabilities and heavyweight 'paranoid' coordination protocols. Consequently, this leads to several serious limitations. Firstly, these approaches tend to diverge toward complex agent designs and implementations, which reintroduces many of the problems that drove the evolution of increasing software localization in the first place. These agents are difficult to understand and maintain because the potentially large body of exception handling code often obscures the relatively simple domain specific behaviors. Moreover, integrating carefully coordinated exception handling behavior into the agent is fundamentally problematic. The inherent distributed nature of the multi-agent systems lacks a global view necessary to detect and resolve many systemic errors, which are results of an emergent dysfunction rather than local to any particular agent. Lastly, the explosive state space complexity in the dynamic environment simply makes it impossible for any designers to specify the correct behavior in each possible state in advance. Thus, there will always be some situations unmanageable by the agents.

1.2.2 Contract Net: an Agent Coordination Protocol

Let us consider, as a concrete example, the survivalist approach as applied to a well-known multi-agent coordination mechanism, namely the contract net protocol.

1.2.2.1 Overview

The contract net protocol (Davis and Smith 1983) for decentralized task allocation is one of the most important paradigms developed in distributed artificial intelligence (DAI) as it is the foundation of many other coordination protocols. Contract net is essentially a high-level protocol for communication among nodes in a distributed problem solver. It facilitates task sharing, where each node in the net take on one of the two roles related to

the execution of an individual task: contractor or subcontractor. A contractor is responsible for monitoring the execution of a task and processing the results of its execution whereas a subcontractor is responsible for the actual execution of the task. A contract is established by a process of local mutual selection based on a two-way transfer of information.

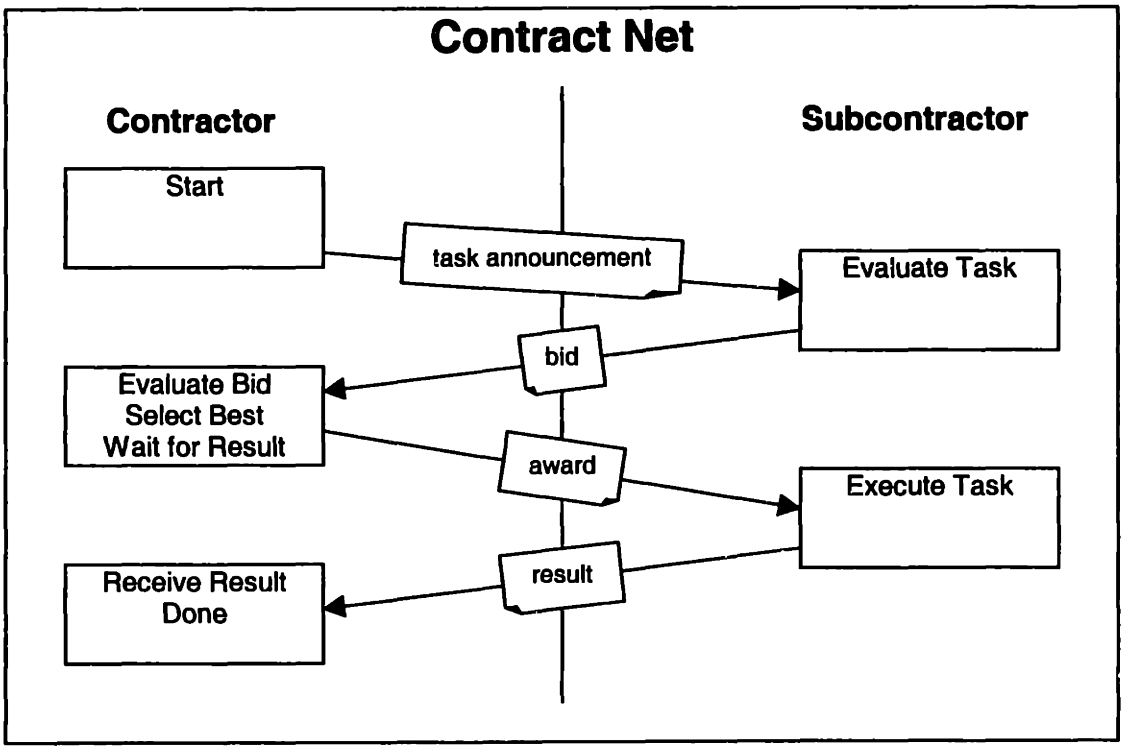


Figure 2. The Contract Net Protocol

In brief (see Figure 2), contractors send task announcement messages, which are evaluated by available subcontractors. Subcontractors then submit bids on tasks they are suitable for. Finally, the contractors evaluate the bids and award contracts to the nodes they determine to be most appropriate. The negotiation process may recur; subcontractors may further partition a task and award contract to other nodes, thus becoming the contractors for those contracts. This leads to a hierarchical control structure, where

control is distributed because processing and communication are not focused at particular nodes, but instead, every node is capable of accepting and assigning tasks.

1.2.2.2 Survivalist Approach: Heavyweight Contract Net Protocol

It is easy to imagine that the contract net protocol described above will have to be 'augmented' before it can be used in real-life applications. Exceptions such as the absence of bids due to lack of available resources must be resolved. The protocol should also be able to handle situations where the subcontractor is unreliable, thereby delivering faulty results or simply crashing. Indeed, many survivalist approaches have been applied to deal with these exceptions by devising various heavyweight contract net protocols. The following are some examples of these approaches.

(Smith, 1980) proposes an extension to the contract net protocol to handle cases when no bids are received by a task announcement. Two assumptions are made: 1) agents can submit bids only when idle and ready for new tasks and 2) no cancellation of an existing reservation is allowed. There are two possible causes for absence of bids: lack of available agents, due to either no idle agents, no eligible agents, or low task ranking and busy network load. A special kind of bid, namely the 'immediate response' bid was proposed to manage the first cause. Task managers can request for an immediate response bid such that each recipient must respond immediately with either a standard bid or a special one indicating whether it is ineligible, eligible but busy, or gives the task a low ranking (see Figure 3). This approach is problematic because it clearly generates a number of extra messages, increasing the network load, which is essentially the other

target issue. Since there are no exact ways of determining how long the contractor should wait for; random re-announcements may worsen the situation even more.

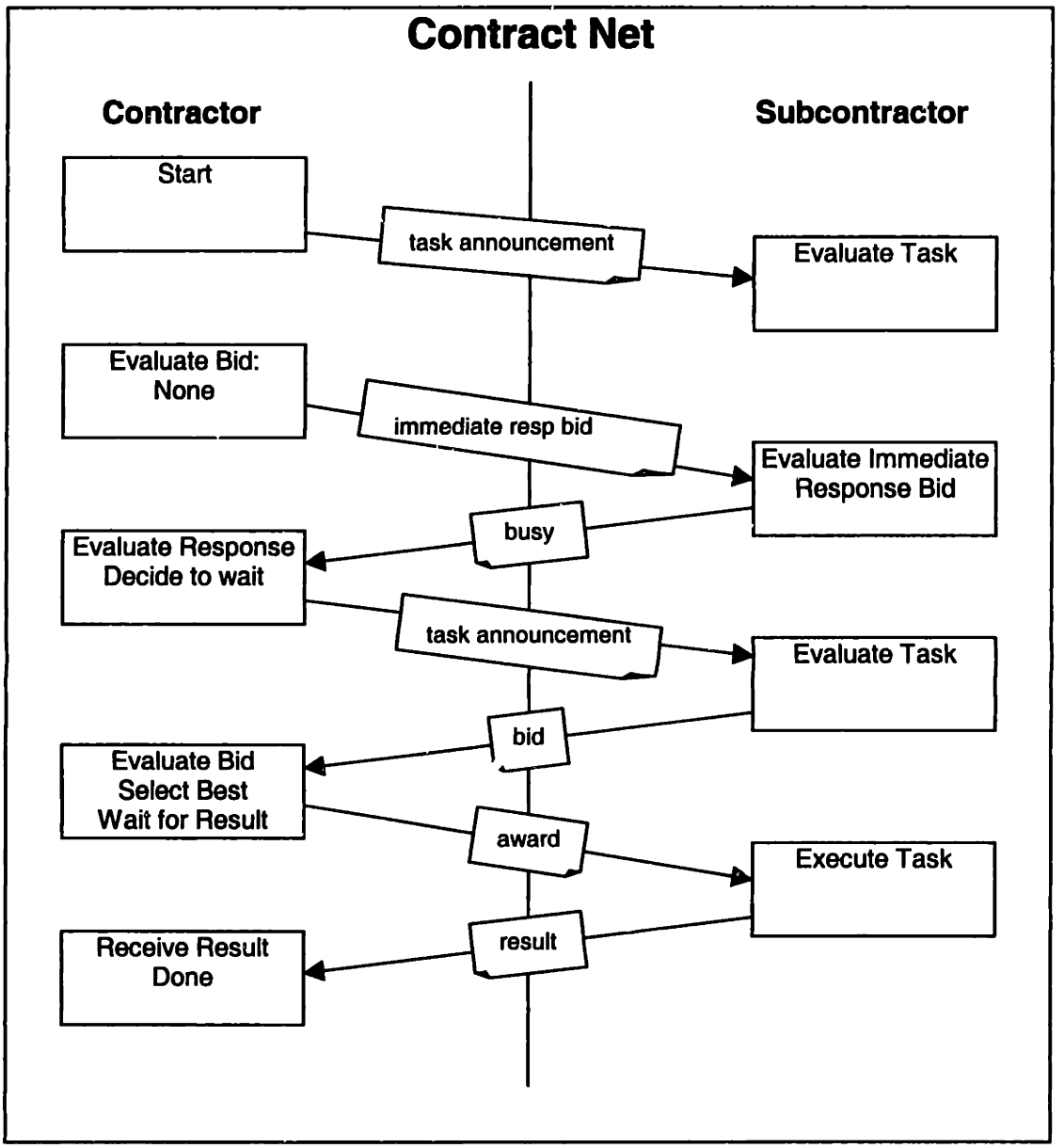


Figure 3. The Contract Net Protocol + Immediate Response Bids

Furthermore, the concept of 'node available' messages was proposed to alleviate problems generated by busy network load. These messages will allow idle agents to

transmit a node available message to search for a task to execute (so that a search for a task can be made). Task managers maintain a list of non-expired node available messages and utilize it to find a suitable volunteer before announcing a new task thereby reducing the network load. Both the immediate bid response and node available approaches increase the complexity of the agents by requiring them to perform additional functionality, such as handling more message types, keeping track of reasons for rejecting bids, and maintaining non-expired node available messages.

Agent death during task execution is another common exception in contract net systems and can be very hazardous especially in real-time domains. If a subcontractor crashes while executing a task, the contractor will not find out until it is time to collect the incomplete task results. The reverse situation is also undesirable. If contractor crashes after awarding a task to a subcontractor, the task will still be executed, resulting in wasted output. A survivalist solution to this is to include task monitoring into the protocol. Upon awarding a task, each contractor periodically monitors its subcontractor to ensure that it is still alive and properly executing its tasks. Similarly, upon receiving an award, each subcontractor monitors the contractor's status.

This approach poses several limitations. Firstly, it introduces additional complexity to the contract net protocol, as shown in Figure 4. Moreover, agents have to create and respond to large numbers of monitoring messages, reducing the agent's performance. The overall system's performance may also be affected due to the large number of extra messages generated in this augmented protocol.

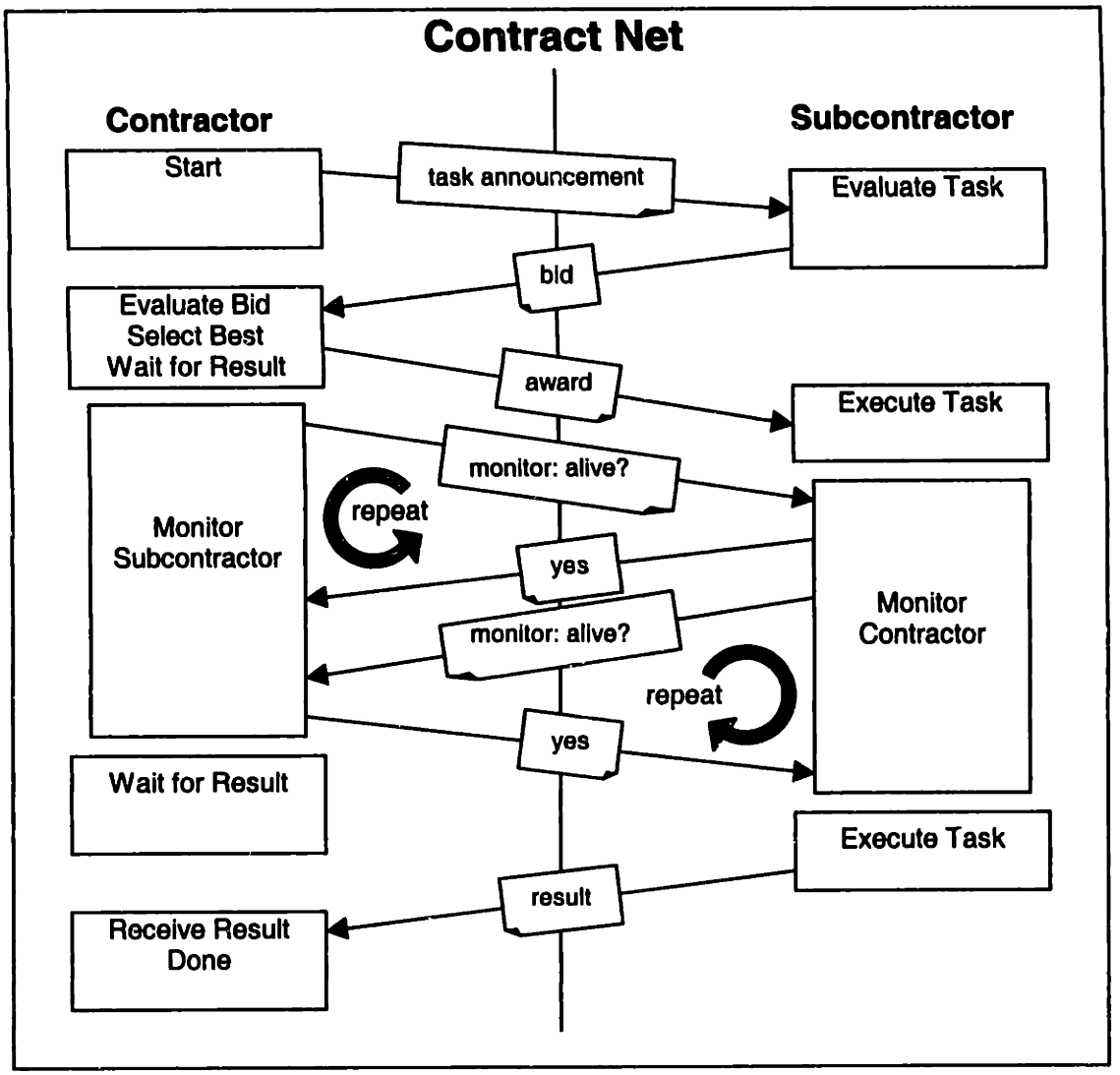


Figure 4. Contract Net Protocol + Task Execution Monitoring

1.3 The Good Citizen Approach

(Dellarocas and Klein 1999) propose an alternative approach using a generic exception handling service which can be 'plugged' into existing agent-based systems, allowing the agents to offload their exception handling needs. The shared service utilizes a reusable knowledge base to detect, diagnose, and resolve exceptions due to the changing environment. In essence, the exception handler provides an infrastructure similar to a

social institution, which ensures that the 'citizen' agents will survive in their hazardous environments as long as they adhere to a basic set of interfaces for communicating diagnostic information as well as receiving instructions.

Consequently, agents can function using relatively simple 'lightweight' protocols despite the occurrence of faults, unanticipated interactions, or malicious intrusions in the system. The exception handling service allows simpler agent designs because agents need to implement only their normative behavior without having to anticipate all possible scenarios in advance. Furthermore, since the exception handler has global view of the system, it can potentially access the information needed to detect and effectively resolve systemic errors produced by unanticipated interdependencies among the agents. This includes undesirable situations caused by agents' heterogeneity, a prominent issue in multi-agent research.

This approach relies on the powerful notion that generic and reusable exception handling expertise can be identified separately from the domain specific knowledge necessary for the agent to perform their tasks. Early work on expert systems validates this claim by revealing that it is useful to separate domain-specific problem solving and generic control knowledge (Barnett 1984; Gruber 1989). Klein's previous work in the domain of collaborative design conflict management (Klein 1991) and preliminary work on process exception management (Klein 1997; Dellarocas and Klein 1999) also confirmed this claim, pointing out that generic exception handling strategies are easy to find (Klein 1989). For example, if an agent's plan has failed, backtrack to a different plan for

achieving the same goal; if an agent may be late in producing a time-critical output, check whether the consumer agent will accept a less accurate output in exchange for a quicker result.

1.4 Thesis Goal

This thesis evaluates the potential of the generic shared service approach in providing adaptability and handling domain independent exceptions in agent-based systems. The goal of this thesis is twofold. The first one is to evaluate the feasibility of the proposed approach in practice, using a contract net setting. Evidently, this involves investigating two key questions:

- Is it possible to identify a useful set of domain independent exceptions relevant to the contract net protocol and strategies for detecting and handling them?
- Is it possible to define a set of query and action interfaces that is both easy for the agents to implement but also powerful enough to enable useful exception handling?

The second goal is to provide an environment for running experiments involving the exception handling approach. This thesis presents a prototype implementation of the exception handler as well as a sample experiment demonstrating a portion of the shared service's domain independent exception handling strategies.

1.5 Outline

The next chapter presents a functional architecture of the proposed approach. A step by step scenario portraying the activity of exception handler is included, using examples in a contract net setting. Chapter 3 describes an evaluation of key issues involving the feasibility of the exception handling service approach. A set of compiled contract net

related exceptions and handling strategies is presented along with the minimum base level query and action interfaces. The implementation of the exception handler prototype and a sample experiment demonstrating the exception handling service when used by lightweight contract net agents in handling agent death exception are described. In Chapter 4, a thorough analysis and comparative study of the proposed exception handling versus other related efforts is presented. Chapter 5 concludes this thesis and enumerates several possible avenues for future research.

Chapter 2

Overview of the Exception Handling Service

The proposed architecture consists of problem solving agents sitting on top of the agent systems infrastructure, the exception handler, and sentinel agents. Both the problem-solving agents and infrastructure must support at the minimum the base level query and action interfaces. It is envisioned that the exception handler will need to support multiple levels of these interfaces, which also correspond to multiple levels of exception handling service. When a new agent is created, it registers to the exception handler and presents specifications of its normative behavior. The exception handler's registrar utilizes the failure mode knowledge base to create one or more sentinels to detect a set of symptoms derived from the agent's specifications. The sentinel can accomplish this assignment by monitoring communications and querying the agent for further information if necessary. If the sentinel detects any of the symptoms, it will report to the exception handler. The exception knowledge base contains information regarding cause-effect relationships between exceptions and symptoms. Based on this, a set of candidate diagnoses is identified and passed on to the resolution knowledge base, producing a set of resolution or avoidance actions. During the diagnoses and resolution selection process, the exception handler may need to query additional information from both the agent and

infrastructure. Lastly, the agent and infrastructure are instructed to execute these resolutions and avoidance plans through their action interfaces.

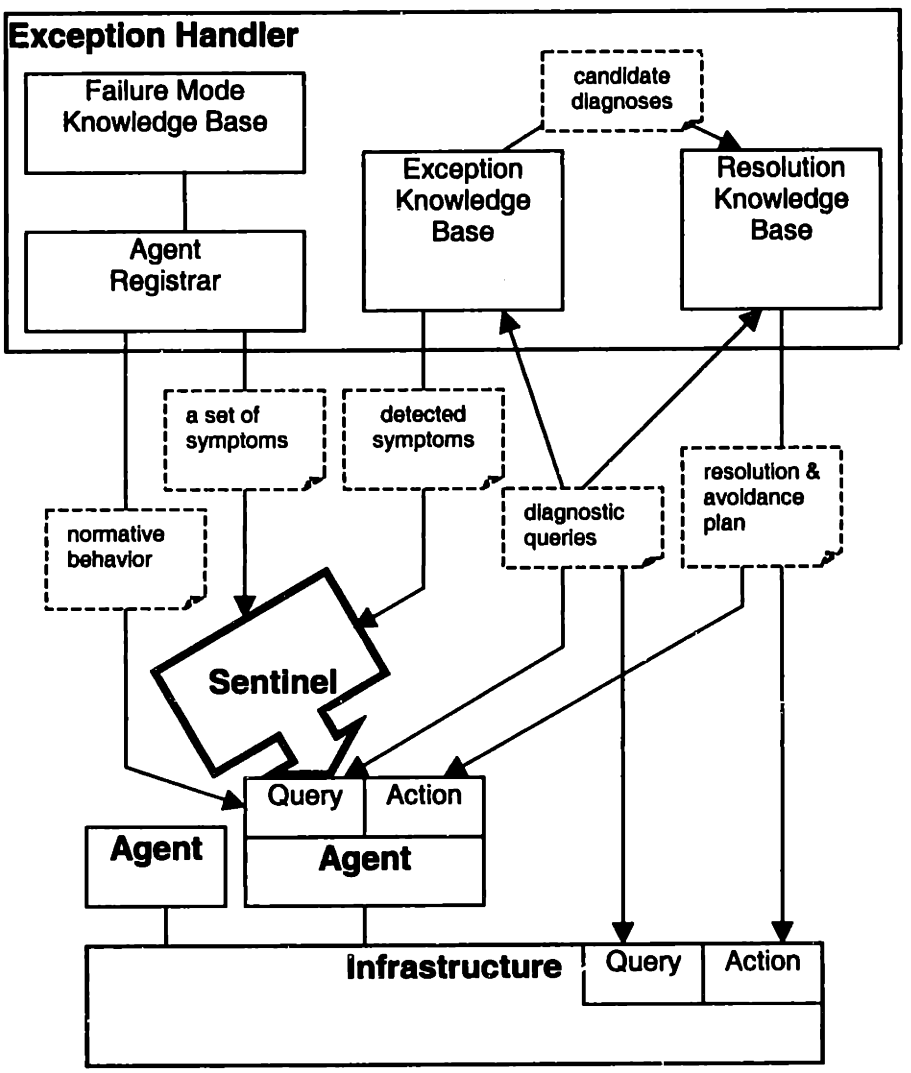


Figure 5. Functional Architecture

The following is a step by step scenario of the exception handling activity when used in a multi-agent system employing the contract net protocol.

2.1 Registration

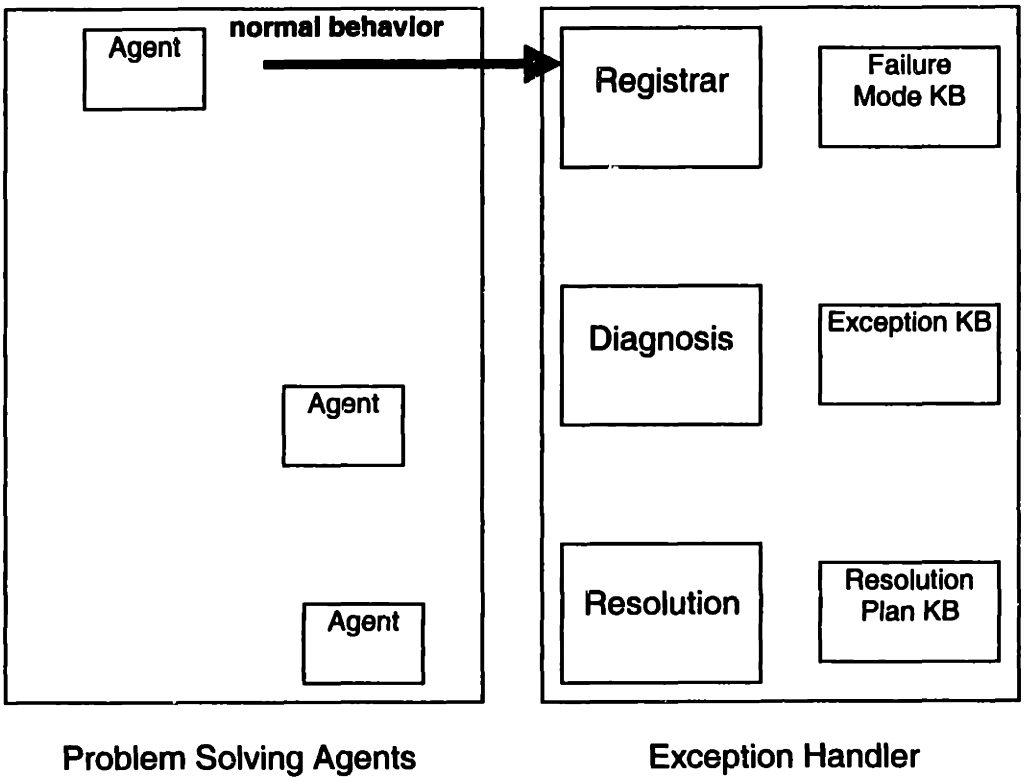


Figure 6. Registration Process

As shown in Figure 6, each agent initially registers to the exception handler and submits information regarding the type of coordination protocol it employs, which in this case is the contract net protocol. There are typically quite a few variations within a single coordination protocol, largely due to the various ways the protocol can be implemented and the specific needs of each domain application, and in many cases to incorporate

specific exception handling capabilities[1]. Thus, the agent must include detailed properties of the protocol into its specifications, explaining what happens if a contractor receives no bids, whether or not timeout is used in bid collection and cancellation is allowed, how the agent performs task execution monitoring if any, etc. Figure 7 shows an example of a contract net version.

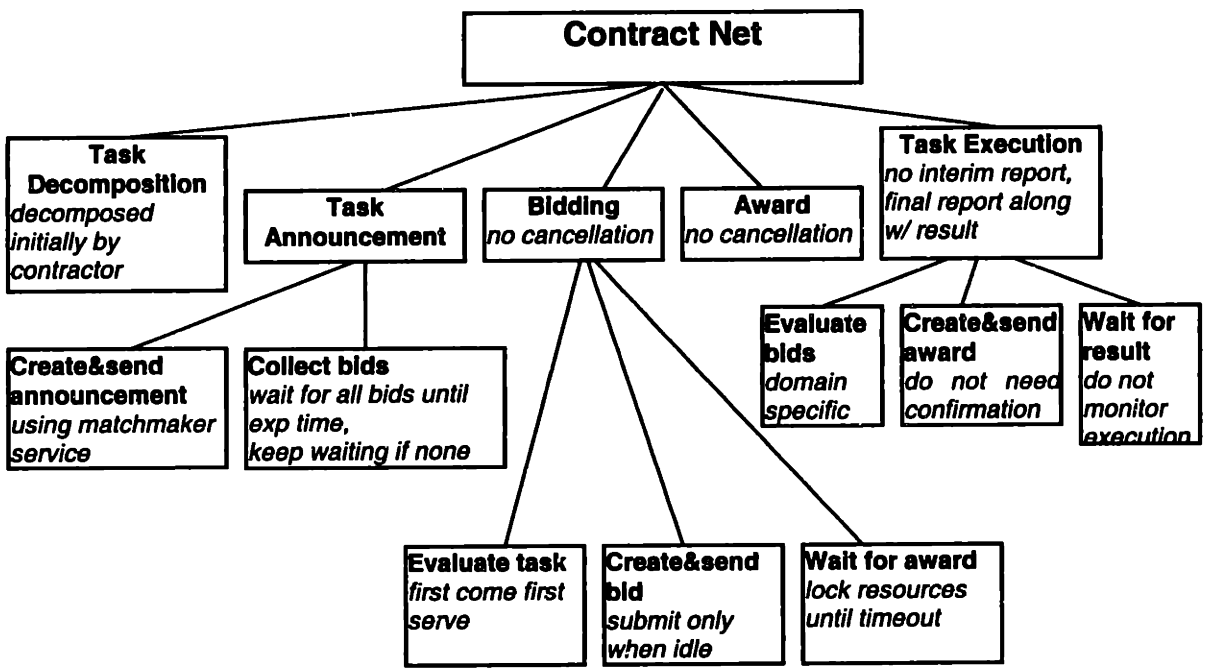


Figure 7. An Example of a Contract Net Variant

2.2 Failure modes mapping

In this scenario, the agent specifies that it is not equipped to do any task execution monitoring. In particular, upon awarding a task to a subcontractor, the agent does not monitor the execution process and simply waits for the end result. The registrar records and relays this description to the failure mode knowledge base. Utilizing information

[1] One can imagine that a contract net version may have some exception handling capabilities, in which case the exception handler only has to deal with the remaining exceptions. However, this case is not covered in this thesis.

derived from a taxonomy of contract net protocol variations (see Appendix A); the knowledge base identifies the agent death exception among several others as a common failure mode for the specified protocol variant (see Figure 8). Essentially, since the agent does not monitor its subcontractor while executing task, it may run into problems if the subcontractor crashes without notice. The reverse situation in which the contractor dies may occur as well, resulting in wasted task results.

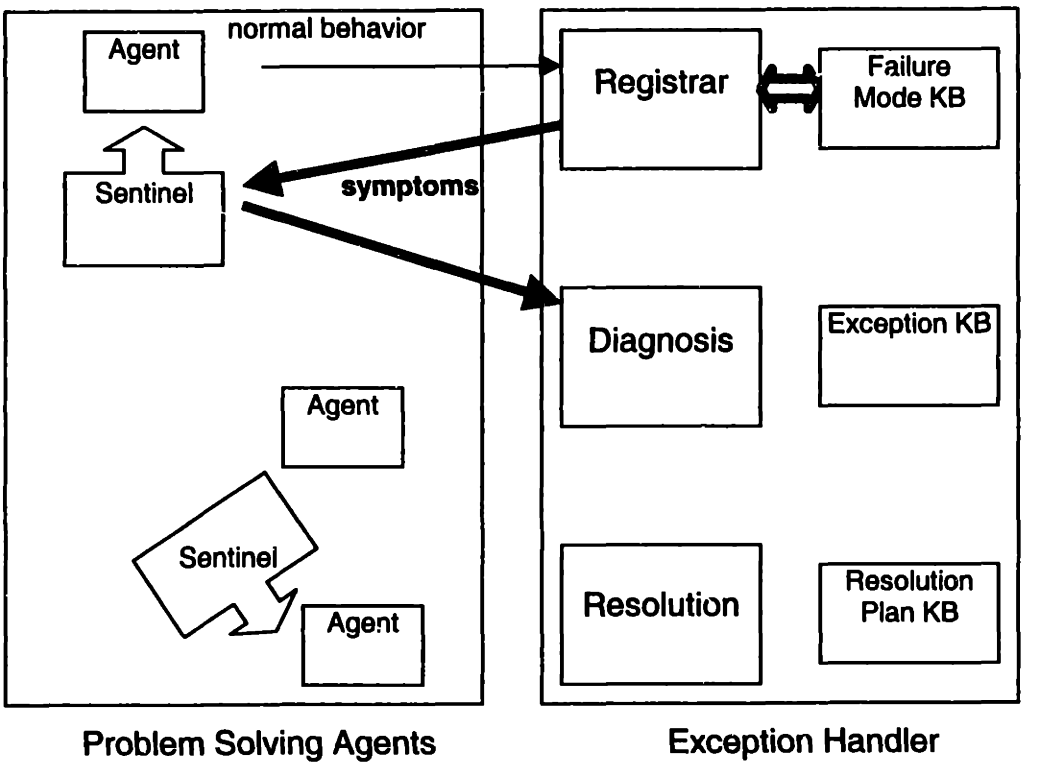


Figure 8. Failure Mode Identification and Sentinel Creation

2.3 Sentinels for detecting and anticipating exception

Although it is clearly advantageous to anticipate an exception in advance and avoid it, in some cases it may be necessary to detect an exception and attempt to resolve it before it generates further problems. Thus, sentinels must be created for both detection and anticipation purposes.

A set of strategies for detecting and anticipating various exceptions pertaining to the contract net protocol have been collected (see Appendix B). The agent death exception can be detected by identifying silent and unresponsive agents. The sentinel is thus assigned to periodically query each agent who is currently involved in a contract, either as a contractor or subcontractor and has not sent any messages for a certain period of time. Anticipating an agent crash can be accomplished by keeping track of the death frequency during the history of each agent. If the frequency exceeds a certain threshold value, the sentinel can conclude that it is likely that the agent will crash in the future.

2.4 Diagnosing exception

Upon identification of any symptoms, the exception handler is notified. Since a symptom may have multiple causes, the exception knowledge base is utilized to heuristically diagnose the source of the observed symptoms (see Figure 9). For example, silent and unresponsive agents may be caused by either agent death or network congestion (see Figure 10). The possibility of network congestion can be ruled out by monitoring other agents residing on the same host as the unresponsive agent. If they are not silent, the

hypothesis can be discarded since network congestion should affect all agents in a particular host indiscriminately.

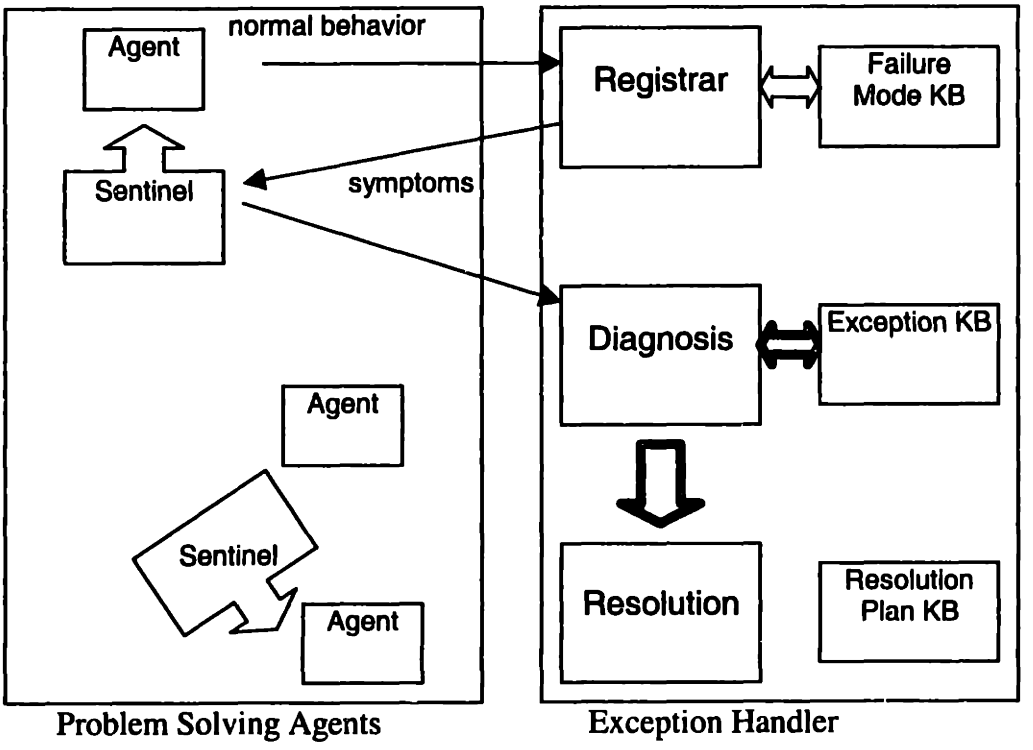


Figure 9. Diagnosis Process

Evidently, the diagnosis process may end up with multiple exception candidates, in which case we are forced to apply resolution plans for each candidate until one works. One can imagine assigning a numerical probability of likelihood, as a way of ordering which resolutions should be tried first. This heuristic classification approach is appropriate,

because similar to the medical diagnosis domain, complete and consistent first-principle-based behavioral models do not exist in this case (Dellarocas and Klein 1999).

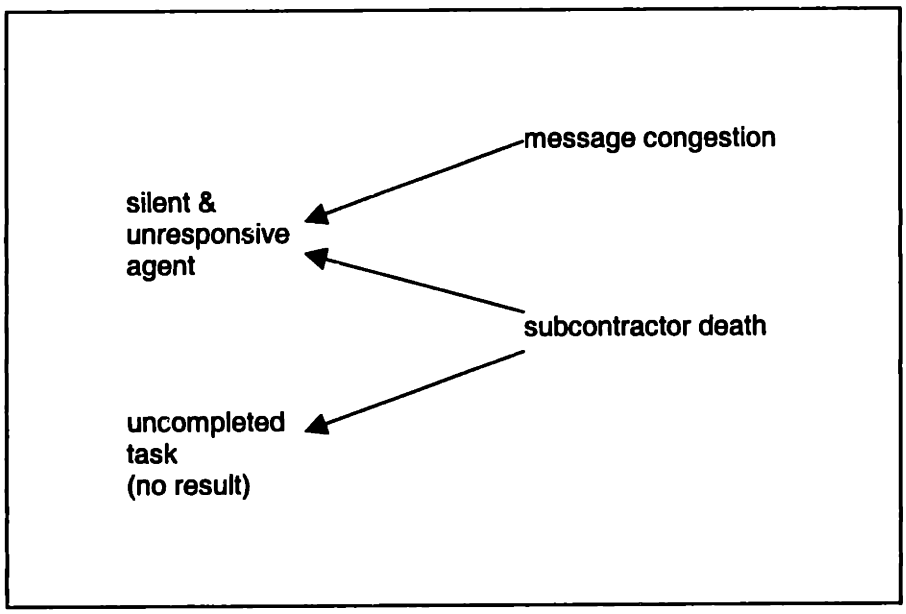


Figure 10. Causal-Effect Relationship of Agent Death Exception

2.5 Resolution and avoidance plan selection

Once diagnoses candidates are identified, they are entered as input to the resolution plans knowledge base, producing a series of resolutions and avoidance actions as shown in Figure 11. As soon as agent death is diagnosed as the liable exception, the resolution plans knowledge base decides that the exception handler should instruct the agent's contractor to re-announce task if the agent is a subcontractor and instruct the agent's subcontractor to abort task execution if the agent is a contractor. If the agent's subcontractor has subcontracted the task to other agents, the exception handler will have to apply this resolution recursively to the subtasks as well. If it is possible to recover any

completed subtask results, the exception handler will collect these results, send an illusory bid, and forward the completed subtask results to the replacement contractor.

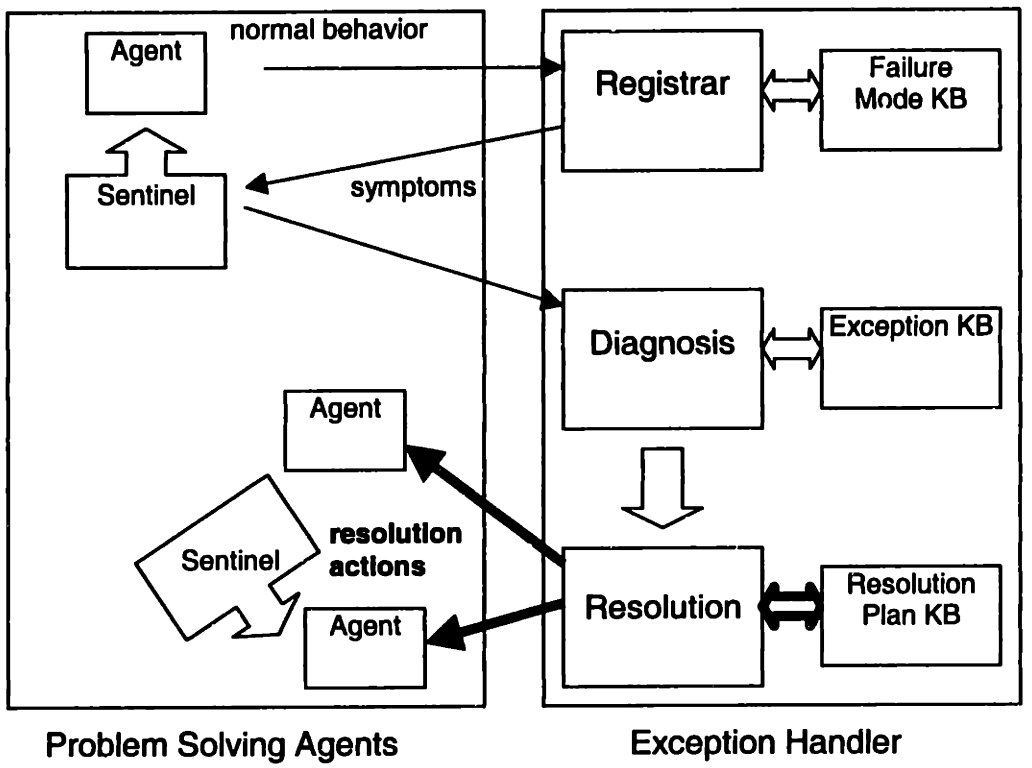


Figure 11. Resolution Process

In some cases, the exception handler may need to acquire further information from both the agent and infrastructure in order to determine the most appropriate resolutions or avoidance plans. For example, if the diagnosis process concludes that it has anticipated an agent death, the resolution plans knowledge base may decide that the agent should be killed permanently such that it will no longer reappear in the system. In addition, the knowledge base determines whether or not the unreliable agent provides the action interfaces for instructing it to kill itself permanently. If not, the exception handler will have to instruct the infrastructure to ostracize the agent.

There are two ways to apply the resolutions and avoidance plans: by instructing the agent to execute them through action interfaces and by modifying the state of the environment such that the exception is solved transparently. Instructing the agent to kill itself is an example of the first category. Resorting to the infrastructure for ostracizing the agent is an example of the second one. A set of resolution and avoidance strategies containing both of these categories have been identified and arranged into a taxonomy (see Appendix B).

Chapter 3

Implementation and Evaluation

As mentioned above, the goals of this thesis are to evaluate the feasibility of the proposed generic exception handler in practice and provide an environment for running tests and experiments involving the exception handling service. Clearly, several issues need to be investigated. How much information is needed from the agent? Is it possible to detect and resolve exceptions without any domain knowledge? How much does the agent have to implement to be able to execute the prescribed resolutions? This chapter addresses these questions and presents a prototype implementation as well as a summary of insights acquired from evaluation of achievement of thesis goals.

3.1 Evaluating feasibility

There are two essential steps in evaluating the feasibility of the shared exception handling service. Firstly, we need to validate the claim that domain independent exception handling expertise can be usefully separated from domain specific knowledge. This involves identifying domain independent exceptions relevant to contract net as well as their detection and handling strategies. The second step entails investigating the size of the minimum base level query and action interfaces and whether or not these requirements are reasonable. Note that the simple contract net variant described in Figure

7 is used as a model for deriving these exceptions, handling strategies, and base level interfaces. The following sections describe results and findings acquired from both of these steps.

3.1.1 Domain Independent Exceptions

A set of domain independent exceptions relevant to the contract net protocol have been identified by reviewing the literature for survivalist heavyweight contract net approaches (Sandholm and Lesser 1995; Chia et.al. 1998; Ramos 1995; Fatima et.al 1998; Fischer et.al. 1995; Ohko et.al 1996; Saad et.al. 1995) and studying their target exceptions. Below are detailed descriptions of these exceptions along with their associated symptoms:

- Resource poaching

Resource poaching occurs when agents with a more critical goal fail to secure certain resources because other agents have reserved it at an earlier time. Thus, the symptom of resource poaching is when a task with higher priority than other awarded tasks in the system is rejected.

- Message congestion

Message congestion is a typical exception in distributed systems due to busy network load, resulting in delayed and lost messages

- Agent death

Component death/crash is a significant issue in any non fault-tolerant software system. As mentioned above, a common symptom of agent death is silent and unresponsive agent.

- Fault in task execution

Fault in task execution may be caused by various reasons: buggy or malicious agents, lack of domain specific knowledge to handle environmental changes, etc. Since the result of task execution is highly domain dependent, this exception doesn't have any observable symptoms. Hence, its detection process involves requiring the agent to complain to the exception handler upon receiving faulty results.

- Delay in task execution

Delay in task execution may be caused by overloaded processors, delayed or lost messages, etc. Similarly, this exception doesn't have any observable symptoms. Hence, its detection process involves querying the agent for the percentage of task completion.

- Protocol mismatch

Protocol mismatch occurs when two agents involved in a contract employ variants of the contract net protocol which conflict with one another. In this thesis, an example involving agent's cancellation policy is used. If the contractor supports award cancellation and the subcontractor does not, the task will still be executed and inevitably wasted. On the other hand, if the subcontractor supports bid cancellation and the contractor does not, the task will be abandoned until detected by the contractor.

3.1.2 Detection, Resolution, Anticipation, and Avoidance Strategies

A collection of detection, resolution, anticipation, and avoidance strategies for each of the exceptions described above has also been identified, as shown in Table 1 (See Appendix B for more details).

Table 1. Detection, Resolution, Anticipation, and Avoidance Strategies

Exception	Symptoms	Detection and Resolution	Anticipation and Avoidance
		Q: query interface	*A: action interface*
Resource Poaching	Task w/ higher priority than other awarded tasks in the system is getting rejected	Detection: Detect when an announcement of a high priority task does not receive any bids	Anticipation: Detect when grid manager announces incoming high priority tasks
		Resolution: Abort or suspend lower priority tasks to accommodate higher priority ones	Avoidance: *1.* Switch between non-priority-sensitive and priority-sensitive scheduling algorithms 2. Batch task requests transparently
	Delayed messages	Not applicable because resolving message congestion involves increasing network capacity which is ultimately bounded	Anticipation: Q: query infrastructure for communication link load and detect when it is heavy
			Avoidance: Suspend lower priority tasks in the hope that it will reduce the number of generated messages
Agent death	Silent and unresponsive agent	Detection: Periodically poll agents upon receiving or granting an award every x time units since their last message and detect when unresponsive Diagnosis: Poll other agents residing in the same host; if they are not responsive, message congestion is more likely to be the cause Note that this is only relevant if we assume that messages between the agent and exception handler also suffer from network congestion	Anticipation: Record each agent death event and detect when failure frequency of the associated source code exceeds a threshold value
		Resolution: Instruct the corresponding contractor and subcontractor to abort or re-announce properly in order to avoid abandoned or wasted tasks If it is possible to recover subtask results, collect these results, send illusory bid, and forward them to the replacement contractor (Note that this feature is not implemented in the current prototype)	Avoidance: 1. Blacklist agent (provides Better Business Bureau-like services) A: instruct agent to kill itself

Exception	Symptoms	Detection and Resolution	Anticipation and Avoidance
		Q: query interface	*A: action interface*
Fault in task execution	Contractor's complaint	Detection: • Detect complaints from contractor upon receiving task results • Diagnose the agent causing faulty results by traversing the task chain	Anticipation: Record each fault in task execution and detect when frequency of faulty results caused by an agent exceeds a threshold value
		Resolution: Instruct the corresponding contractor to re-announce If it is possible to recover subtask results, collect these results, send illusory bid, and forward them to the replacement contractor	Avoidance: 1. Blacklist agent (provides Better Business Bureau-like services) 2. A: instruct agent to kill itself
Delay in task execution	Late result	Not applicable because delayed task result is irreversible	Anticipation: • Q: query infrastructure for each host processor load • If load is heavy, Q: query agents residing on the host for percent done and detect when any agents will not be able to meet task execution deadline
			Avoidance: Move the agent to another host if appropriate
Protocol mismatch	Cancel-savvy (fickle) agents are involved in a contract with non-cancel-savvy agents	Detection: Detect award and bid cancellations	Anticipation: Detect i) when a contractor who does not support bid cancellation receives bids from fickle subcontractors or ii) when a fickle contractor receives bids from non-fickle subcontractors
		Resolution: Instruct the corresponding contractor and subcontractor properly in order to avoid abandoned or wasted tasks	Avoidance: Avoid interactions between these agents or resort to detection & resolution if this is not possible

Analyses are performed to compare each of these exception-handling strategies against either an existing or potential survivalist approach for dealing with the same exception. Table 2 shows a list of hypothesized advantages of the exception handling strategies over the survivalist solutions based on this analysis.

Table 2. Hypothesized Advantages of the Domain Independent Exception Handling Strategies

Exception	Advantages	How? *EH = Exception Handler* *SA = Survivalist Agent*
Resource poaching	Simpler agents	• EH can transparently batch requests, allowing simpler agents to avoid resource poaching. • SA needs to augment its contract net protocol such that it can acquire some global information required to avoid resource poaching.
	Global perspective	• EH can determine overall system load and respond accordingly (tunability) • Again, SA needs to acquire some global information to be able to do this
	Support for emergent dysfunction	• EH can easily detect resource poaching • It is difficult for SA to detect and anticipate for emergent dysfunction
	Support for agent heterogeneity	• EH can potentially support agents employing various protocols (In this case, EH utilizes agent's ability to switch among various scheduling algorithms) • Agent interoperability is one of the most critical unsolved issues in current SA research
Message congestion	Global perspective	• EH can locate lower priority tasks and suspend them, thus effectively accommodating higher priority ones • Each SA can only attempt to reduce the number of generated messages
	Support for emergent dysfunction	• EH can detect resource poaching by communicating with the infrastructure • Again, it is difficult for SA to detect and anticipate for emergent dysfunction
Agent death	Simpler agents	• EH eliminates unreliable agents, thus protecting other agents transparently • SA has to augment its contract net protocol to include monitoring for agent death
	Greater efficiency	• EH can carry out consolidated monitoring • Each SA has to carry out its own monitoring
Fault in task execution	Simpler agents	• EH eliminates unreliable agents, thus protecting other agents transparently • SA has to augment its contract net protocol to include interim reports, fault penalty, or task monitoring
	Greater efficiency	• EH can function as Better Business Bureau (BBB), thus removing the need for each agent to monitor others • Each SA has to carry out its own monitoring
Delay in task execution	Simpler agents	• EH protects agents from delayed task results due to bounded system resources transparently • SA has to augment its contract net protocol to include interim reports, delay penalty, or task monitoring.

Exception	Advantages	How? *EH = Exception Handler* *SA = Survivalist Agent*
Protocol mismatch	Simpler agents	• EH protects non-cancel-savvy agents transparently • SA has to augment its contract net protocol to check for other agent's cancellation policy before engaging in a contract
	Greater efficiency	• EH can function as Better Business Bureau (BBB), thus removing the need for each agent to distinguish agent's employing different protocols • SA has to check for protocol consistencies before engaging in each contract, producing extra messages
	Support for agent heterogeneity	• EH supports agent employing various protocols • Agent interoperability is one of the most critical unsolved issues in current SA research

3.1.3 Base Level Query and Action Interfaces

Given the set of strategies for detecting and handling the target exceptions, a minimum set of query and action interfaces have been derived. As mentioned above, it is envisioned that the exception handler may need to support multiple levels of interfaces, which correspond to various levels of service. The compiled base level (level 0) query and action interfaces plus some examples of level 1 interfaces are shown in Table 2.

Table 3. Query and Action Interfaces

Category	Query	Action
Protocol related	*[level 0]* • Variant/version of protocol • Message types and fields	*[level 0]* • Abort/undo and switch to state x in the protocol, i.e. back-up to re-announce task, abort task execution • Contractor flags EH when result is faulty • Suspend and resume task execution • Switch between First Come First Serve and Priority Based + Batch Request
Fundamental properties of agent	*[level 0]* • Name/ID • Host/Node • Source code ID • Are you alive? • Subcontractor's status: idle or busy *[level 1]* • Percent done in task execution	*[level 0]* • Move to another node
Agent specific attributes	*[level 1]* • Skill/capabilities set, described in a common language. (alternative: Immediate response bid: are you eligible for this task?) • Task's quality of service: e.g. is late result acceptable?	
Infrastructure	*[level 0]* • Processor load per host • Communication link load	*[level 0]* • Kill/ostracize a particular agent

3.1.4 Discussion

The previous findings suggest that contract net related domain independent exceptions and their handling strategies are relatively easy to identify. In fact, this effort has managed to collect domain independent detection and handling strategies for solving exceptions commonly targeted by other survivalist works, such as resource poaching (Chia et.al. 1998), message congestion (Ohko et.al. 1996), fault and delay in task

execution (Ramos 1995; Fischer et.al. 1995), plus some additional exceptions, including agent death and protocol mismatch. Thus, it is evident that solving significant exceptions using generic exception handling expertise, without any access to domain knowledge is feasible. Moreover, in the course of devising these domain independent exception handling strategies, it has been observed that due to the exception handler's access to global perspective, the shared exception handling service is a more natural approach for dealing with emergent dysfunction, such as resource poaching, message congestion and agent heterogeneity. Lastly, the required set of query and interfaces for supporting these exception-handling strategies is small and consists of relatively simple additions to the agent's functionality. This is consistent with one of the hypothesized advantages of the exception handling approach: simpler and easier agent development.

3.2 Prototype Implementation and Sample Experiment

In order to further examine the effectiveness of the proposed exception handling approach in practice and perform relevant tests and experiments, a prototype of the exception handler is implemented. Using a simulated test bed, a sample experiment is carried out to demonstrate the performance of simple contract agents utilizing the exception handling service. The following sections describe the design of the prototype, experimental settings, as well as discussion of the acquired results.

3.2.1 Test Bed: SimHazard + simple contract net agents

The exception handler prototype operates in a simulated test bed consisting of contract net agents employing a lightweight version of the contract net protocol, as shown in Figure 7 (see Appendix C for details on agent's implementation). Both the exception

handler and agents are controlled by David Shue's SimHazard (Shue 1999). SimHazard is essentially an agent-world (network) simulator for injecting exceptions and instability into a multi-agent environment, thus providing an ideal and controlled infrastructure for running experiments and observing overall system behavior. The system has been designed to work with a set of heterogeneous agents. These agents connect to the system and interact with each other within the context of a simulation run, where events within the system affect agent behavior through fluctuations in the performance of network entities.

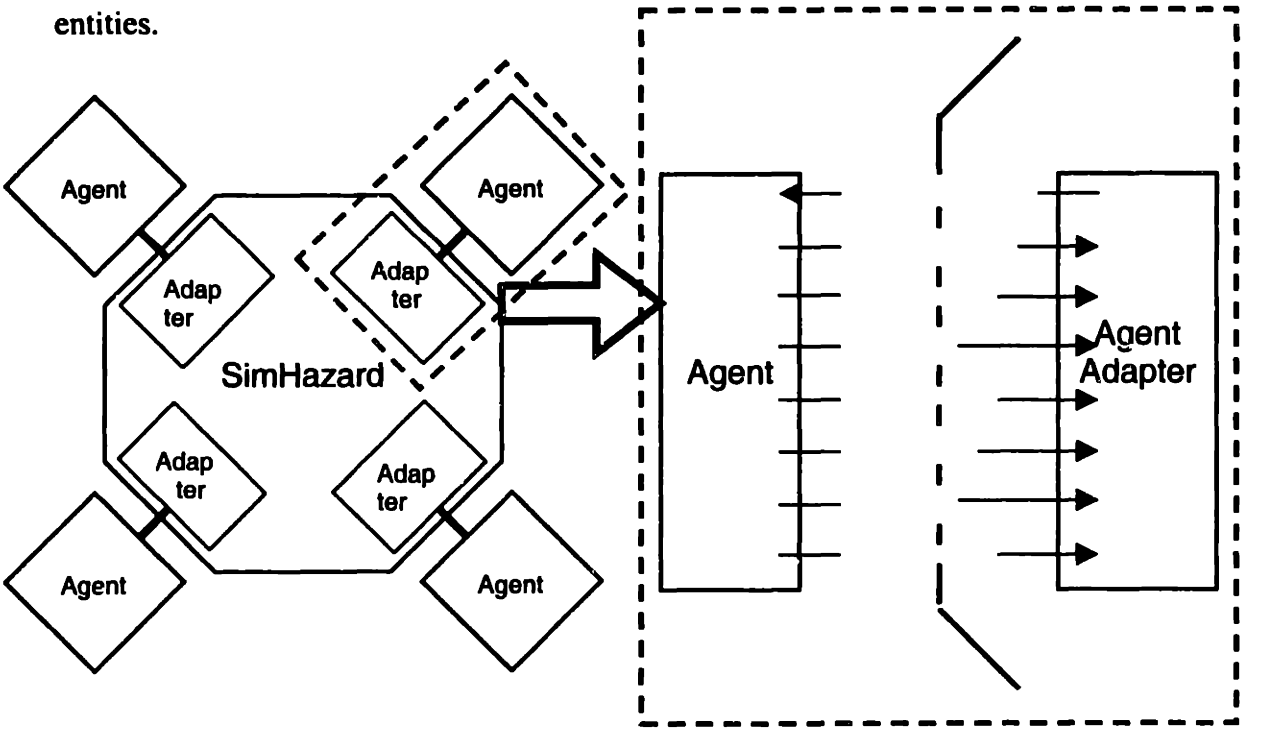


Figure 12. Test Bed: SimHazard + Agents

As shown in Figure 12, once the agents connected to the system, they interact with the simulator through an adapter interface that translates agent directives into simulation events and vice versa. Interaction between the agents and the simulator revolves around sending and receiving messages. The simulator handles message transmission while

agents perform the domain specific message handling, thus abstracting agent's domain specific behaviors from the simulation.

3.2.2 Exception Handler

3.2.2.1 Simplifying Assumptions

Since the goal of the prototype implementation is to assess the performance of the proposed approach, an attempt was made to simplify the exception handler's design as much as possible without altering its functional behavior. This effort is greatly facilitated by the event-based nature of the simulation, removing the needs for dealing with concurrent events. Thus, sentinels no longer need to be distributed. Instead, they can reside within the exception handler, which can simply be implemented as a single thread, receiving and processing messages one at a time. Moreover, many components of the exception handler, i.e. the knowledge bases, can be hard coded and installed within the sentinels.

Another assumption is made regarding the sentinel's message interception mechanisms. Ideally, the sentinel should be implemented as a wrapper to the agent, thereby allowing it to listen and filter messages as necessary. However, for simplifying purposes, the sentinels in the prototype system reside inside the exception handler, through which SimHazard routes all messages in the system. Hence, the exception handler may cut off any messages simply by not forwarding them to the proper receiver.

Several assumptions were made in order to reduce the problem's complexity, keeping in mind that these involve very important issues that should be investigated in future work. Specifically, the exception handler is assumed to generate very little overhead costs for both processing and communications. Unlike the agents, it is also presumed to be immune to the environment. Finally, an assumption is made that exceptions only occur one at a time, eliminating the possibility of complications caused by concurrent exceptions.

3.2.2.2 Design

The exception handler consists of four modules: message handler, data handler, sentinel action handler, and message actuator (see Figure 13). Upon receiving a message, the message handler translates domain independent contents of the message, such as sender, receiver, timestamp, etc. into an internal data structure. The data structure is then passed through the data handler, which maintains data regarding the existing contracts, agents and system infrastructure. For example, by monitoring the messages received by an agent, the data handler can infer and record the role of the agent (contractor or subcontractor), the state of the contract (collecting bid or task execution), etc. It is important to note that the version of the contract net employed by each agent is also communicated to the exception handler in this manner, replacing the registration process. The agent includes the contract net version number in its task announcements and advertisements to the matchmaker. The data handler simply extracts and records this information in the corresponding agent's folder containing other data about the agent.

Once the data update is completed, the sentinel action handler processes its set of active sentinels and actions accordingly. Essentially, each of these sentinels and actions is implemented as a finite state machine, receiving message (event) as input which result in transition to a new state and certain corresponding output events. These include activation or deactivation of other sentinels and actions.

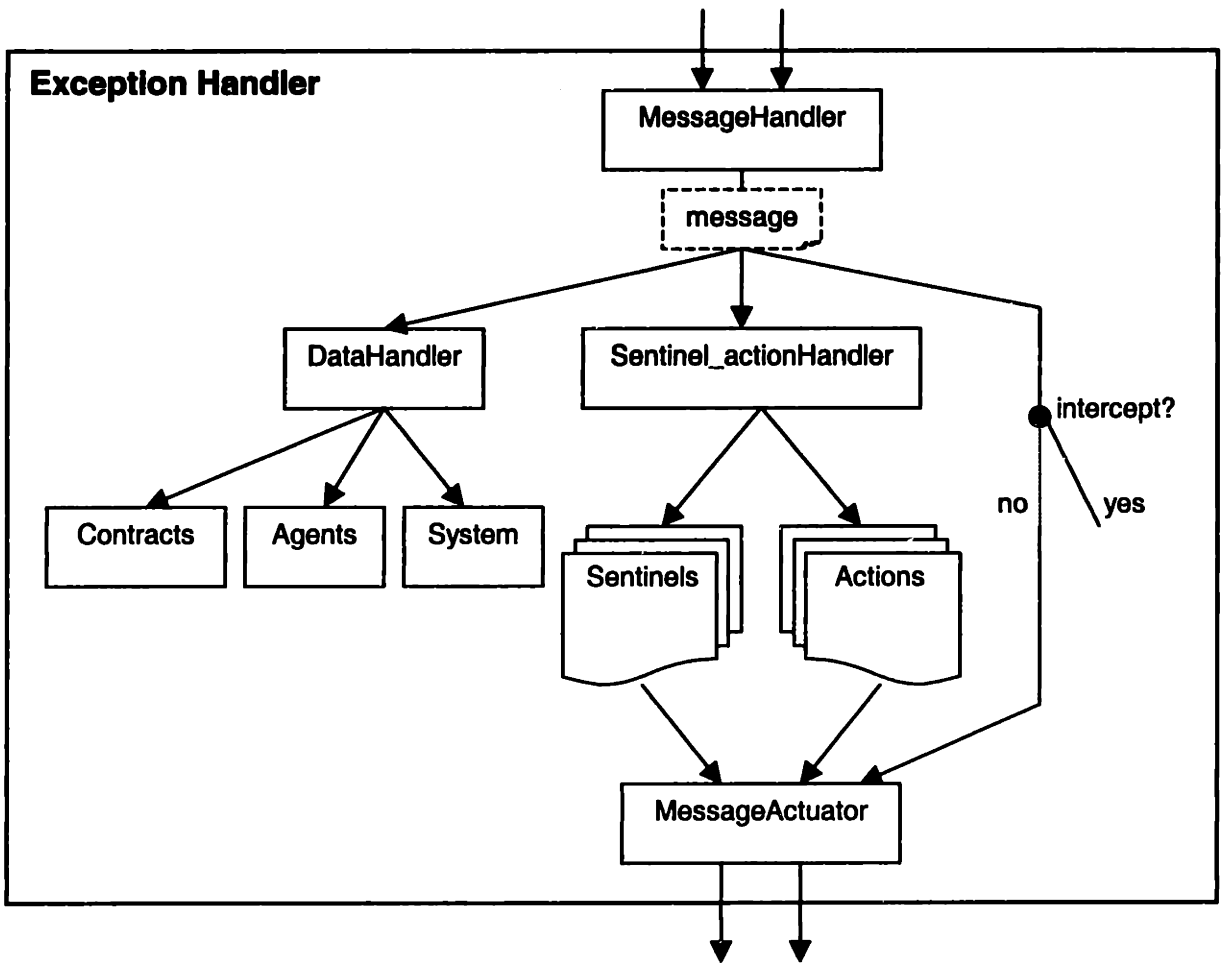


Figure 13. Exception Handler Design

As mentioned above, the failure mode, exception, and resolution plan knowledge bases are bypassed and instead, implemented as various sentinels and actions. In particular, instead of utilizing the failure mode knowledge base to identify common symptoms and

create the proper sentinels to detect them, all failure mode sentinels are activated upon initialization to detect for the existence of agents employing certain contract net version and automatically trigger other sentinels for detecting specific symptoms associated with it. Moreover, without utilizing the exception knowledge base, these symptom-detecting sentinels are equipped to perform the proper diagnosis and query for additional information for the set of symptoms assigned to them. Similarly, upon identifying the cause of the observed symptoms, these sentinels automatically activate the proper actions, which are also equipped to query for further information and dispatch resolution or avoidance instructions through the message actuator. Lastly, the message actuator forwards the received message to the proper recipient, unless the exception handler decides to intercept it.

3.2.3 Sample Experiment

Using the exception handler prototype operating on the simulated test bed, various issues involving the shared exception handling service approach can be evaluated. Figure 8 summarizes a typical simulation experiment. The goal of this experiment is to measure the usefulness of the shared exception handling approach in dealing with subcontractor agents that may crash unexpectedly after they have been awarded a task but before they have completed the work. It works by periodically monitoring the "health" of subcontractors and assisting in the immediate reassignment of tasks performed by failed subcontractors. In the absence of "social monitoring", the particular variant of the contract net protocol used in this experiment only checks for subcontractor death after a task result fails to arrive by the specified deadline. The experiment is carried out as follows:

- task structures (see Appendix C) with varying complexity (size of contractual chains) are assigned to the contractor in the system.
- for each task structure, an agent death exception is simulated at various points during the execution of the task and average task completion delay relative to schedule is measured.

The results of the experiment (see Figure 14) shows that the exception handling service manages to alleviate the negative performance consequences of unexpected subcontractor deaths. In particular, offloading the monitoring responsibility to the exception handler which is able to perform it concurrently during task execution significantly reduces the completion delay of supply chains where at least one of the subcontractors unexpectedly fails.

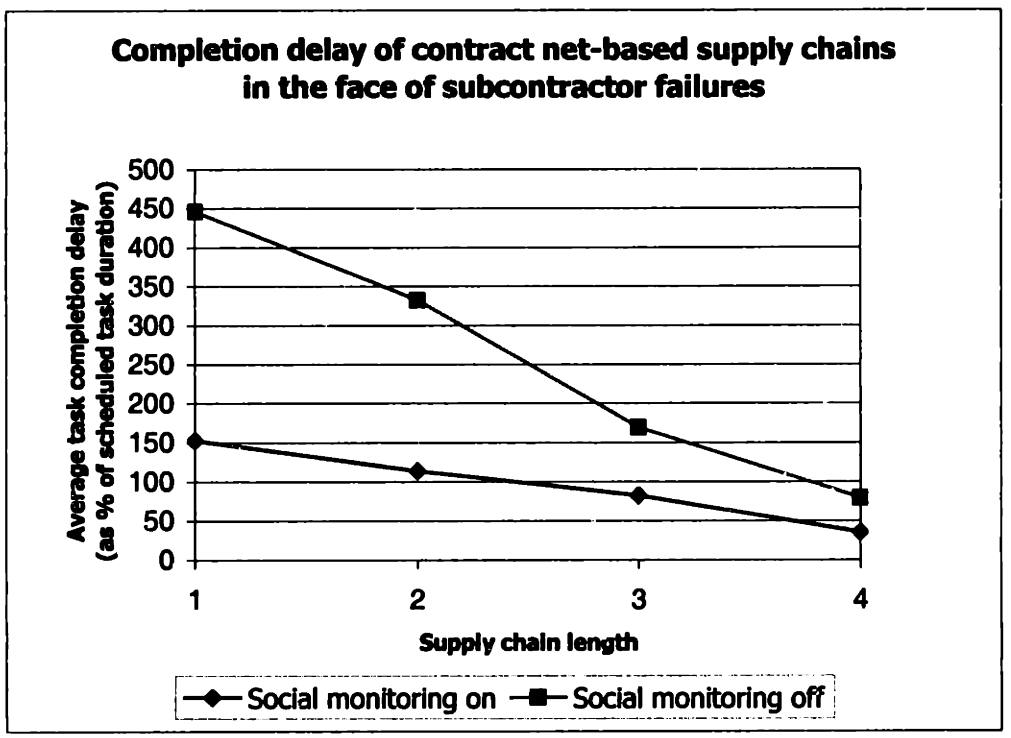


Figure 14. Effect of a "social monitoring" institution on the completion delay of supply chains where at least one subcontractor agent unexpectedly fails.

3.3 Summary: Evaluation of Achievement of Goals

This chapter evaluated accomplishments of this thesis with respect to its goals. As described above, the main objectives of this thesis are i) to evaluate the feasibility of the proposed approach in practice, using a contract net setting and ii) provide an environment for running various tests and experiments involving the exception handling service. The evaluation of the shared exception handling service approach has shown that domain independent exceptions and handling strategies are easy to identify, especially for dealing with emergent dysfunction. Furthermore, the minimum set of query and action interfaces required for supporting these detection, resolution, anticipation, and avoidance strategies is relatively small and easy to implement. These results support the hypothesis that it is in fact possible to acquire effective exception handling services without access to any domain knowledge, by simply relying on a simple set of query and action interfaces. Analyses of the compiled exception handling strategies provide evidence that the shared exception handling service provides the following advantages over the survivalist approach: simpler agents, greater efficiency, global perspective and support for emergent dysfunction and agent heterogeneity. Finally, an exception handler prototype was implemented and utilized to carry out a sample experiment for demonstrating a small portion of the compiled domain independent exception handling strategies. Together with SimHazard, the prototype allows various experiments to be performed for further analyses of the shared exception handling service approach.

Chapter 4

Contribution to Related Work

This thesis has presented the shared service approach for providing effective domain independent exception handling in multi-agent systems employing the contract net protocol. The following is an analysis comparing the shared exception handling service approach with other related efforts.

(Haag 1997) proposes the concept of sentinel agents. This work is the only other existing good citizen approach in current agent research. Similar to the shared service approach, these sentinel agents provides exception-handling service to the other agents in the system. They interact with all agents in the system, monitor communications, and intervene when necessary by selecting alternative problem solving methods, excluding faulty agents, or reporting to human operators. In addition to evaluating agents' behaviors, sentinels also build models of other agents by querying for information and utilize the compiled models to assess the internal state of the agents. Sentinels have a global view of the overall system and thus, are able to detect and handle systemic errors. This approach differs from the shared exception handling service in that it is not intended to be a generic service. Thus, each sentinel must be customized to each domain

application. On the contrary, the shared exception handling approach is generic and reusable, handling various exceptions without any domain knowledge. As a result, this offloads the need for additional implementations of sentinels in each domain application, which in some cases may be a significant amount of work.

(Kaminka and Tambe 1998) present Social Attentive Monitoring (SAM), an approach to monitoring and diagnosis for collaborative multi-agent systems inspired by social psychology. In this approach, agents use one another as information resources for detecting failures in itself as well as its teammates, for example, inconsistent sensing by different agents of environmental conditions. An explicit model of teamwork is utilized to perform social diagnosis and reason about team failures. SAM performs plan recognition and supports model sharing as an alternative method for modeling other agents instead of communicating all relevant information to each other, which can be very expensive and inefficient. Similar to the shared exception handling service approach, SAM supports the idea that monitoring and resolving exceptions as well as global perspective are integral in providing adaptability in agent-based systems. However, unlike the generic and reusable exception handling service, SAM clearly relies on domain knowledge for modeling and reasoning about each other's behavior. Hence, this approach is only relevant to multi-agent systems consisting of socially similar agents that operate within the same domain.

The use of domain independent diagnosis for achieving adaptability is also the foundation of Lesser's recent work (Horling et.al. 1998). Similar to the shared exception handling

service, this approach is based on the notion that being able to accurately detect and determine the cause of a fault or exception is the key to acquiring robustness. This work is mainly concerned with a subset of the domain independent exceptions considered by the generic exception handling service, namely exceptions related to performance issues surrounding the use of situation-specific coordination strategies. Essentially, the agents employ certain coordination strategies, which can be tailored to their current environments and specific situations. The diagnosis is then used to detect when assumptions about agent's behaviors and availability of resources which are the basis for effective situation specific coordination no longer hold, thus altering the current coordination strategy to a more appropriate one. Despite its similarities to the shared exception handling service, this approach essentially belongs to the survivalist category. Both the diagnosis and resolution processes are carried out by each individual agent, which lacks global perspective but has access to domain knowledge. A comparative study between this work and the shared exception handler approach to examine the trade-off in having access to global view of the overall system versus domain knowledge may generate useful insights.

Chapter 5

Future Work

There are clearly many possible venues for future research. The most important issue to consider is the amount of overhead costs generated by the exception handler and the exception handler's quality of service versus overhead costs trade-off. A deeper understanding of the amount of query/action interfaces versus the level of service trade-off is also essential.

An equally important issue relates to the effects of environmental hazards on the exception handler and how they can be handled. Moreover, it is imperative to relax the assumption made regarding the currently used message interception mechanism, thus requiring future work on implementing realistic message interception mechanisms.

Naturally, the work presented in this thesis can be expanded to verify all the hypothesized advantages of the exception handling strategies for dealing with contract net related exceptions as well as to cover exceptions related to other coordination protocols. Lastly, further efforts can be spent in implementing the knowledge bases, studying the causal-effect relationships among various exceptions, developing algorithms for exception detection, diagnosis, and resolution as well as defining a refined language for representing

normative behavior, coordination protocol plus its characteristics, queries and resolution actions.

Appendix A

Contract Net Variants

The following is a comprehensive list of contract net variants as mentioned in Chapter 2.2. The first column contains various subparts of the contract net protocol and the second column shows possible variations of each subpart.

Subpart	Variation
Task decomposition	Nature of task: • Task can be decomposed • Task cannot be decomposed
	Decomposition protocol • Agent has no decomposition protocol • Contractor can decompose and announce subtask • Subcontractor can dynamically bid for subtask
Create task announcement message	Message format: • Standard Contract net Task Announcement Message • Standard + market parameter (offer/bid price)
Send task announcement message	How: • general broadcast • limited broadcast • point-to-point
	To: • directly to agents • indirectly through others
Bid collection	Expiration time: • Wait until expiration time • No expiration time
	Wait until how many bids? • Wait for all bids • Wait until 1st satisfactory bids • Wait until xth satisfactory bids
	What if no satisfactory bids? • Fail/halt/keep waiting • Do nothing, leave task unsatisfied • Wait and re-announce forever until satisfactory bid • Wait and re-announce x times, report to user if none • Immediately report back to user
Bidding	Subcontractor's commitment level: • Support subcontractor's continuous commitment level • Do not support subcontractor's continuous commitment level
	Cancellation/suspension policy: • Support bid cancellation or suspension of task execution • Do not support bid cancellation or suspension of task execution

Subpart	Variation
Task announcement evaluation	Scheduling algorithm: • FCFS • Priority based + batch requests • Marginal cost calculation • Least laxity • Earliest deadline
	Preemption policy: • Allow preemption by subcontractor • Do not allow preemption by subcontractor
Create bid message	Message format: • Standard Contract Net Bid Message • Standard + market parameters (offer/bid price) • Others
Submit bid message	When: • only when idle • anytime
	To: • directly to contractor • indirectly through others
Wait for award	Resource lock: • Lock resources (expiration time) • Do not lock resources (no expiration time)
	Expiration time • Wait until expiration time • No expiration time
Award	Contractor's commitment level: • Support contractor's continuous commitment level • Do not support contractor's continuous commitment level
	Cancellation policy: • Support award cancellation • Do not support award cancellation
Bid evaluation	Preemption policy: • Allow preemption by contractor • Do not allow preemption by contractor
	How: (this is domain dependent, however we can distinguish between market vs non-market based) • Market based • Non-market based
Create award message	Message format: • Standard Contract Net Award Message • Standard + market parameters (offer/bid price) • Others
Send award message	To: • directly to subcontractor • indirectly through others

Subpart	Variation
Task execution	Report: • No report, contractor can only observe end result • Subcontractor sends interim report periodically during task exec • Subcontractor sends final report after task exec • Contractor polls subcontractor for report
	End result delivery: • End result is sent through a message • End result is not sent through a message
	If end result is late or faulty: • Fail/halt/keep waiting • Award other agents who submitted bid for the task • Report back to user • Re-announce task

Appendix B

Exceptions and Handling Strategies

The following table contains detailed descriptions of various exceptions, their associated symptoms, and handling strategies as summarized in Chapter 3.1.2. For each exception in the first column, associated symptoms are shown in the second column. The third and fourth columns consist of the corresponding detection, resolution, anticipation, and avoidance strategies for each exception.

Exception	**Symptom**	**Detection and Resolution**	**Anticipation and Avoidance**
		Q: query interface	*A: action interface*
Resource Poaching	Task w/ higher priority than other awarded tasks in the system is getting rejected	Detection: Detect when an announcement of a high priority task does not receive any bids	Anticipation: Detect when grid manager announces incoming high priority tasks
		Resolution: Abort or suspend lower priority tasks to accommodate higher priority ones • Locate an eligible subcontractor for the rejected task, who is currently executing a lower priority task (Q: immediate response bid) • A: instruct the eligible subcontractor to abort or suspend its task execution • A: instruct the contractor of the rejected high priority task to re-announce • If any tasks are aborted, A: instruct the contractor to re-announce immediately • If any tasks are suspended, A: instruct the subcontractor to resume after high priority tasks are completed	Avoidance: 3. Switch between non-priority-sensitive and priority-sensitive scheduling algorithms • Q: query infrastructure for average processor load. • If load is heavy and low priority tasks are long, A: instruct agent to use priority-sensitive scheduling algorithm • Otherwise, A: instruct agent to use non-priority-sensitive scheduling algorithm. 4. Batch task requests transparently • Intercept task announcement messages and delay each announcement as much as possible • A: instruct contractors to re-announce their tasks in the order of priority

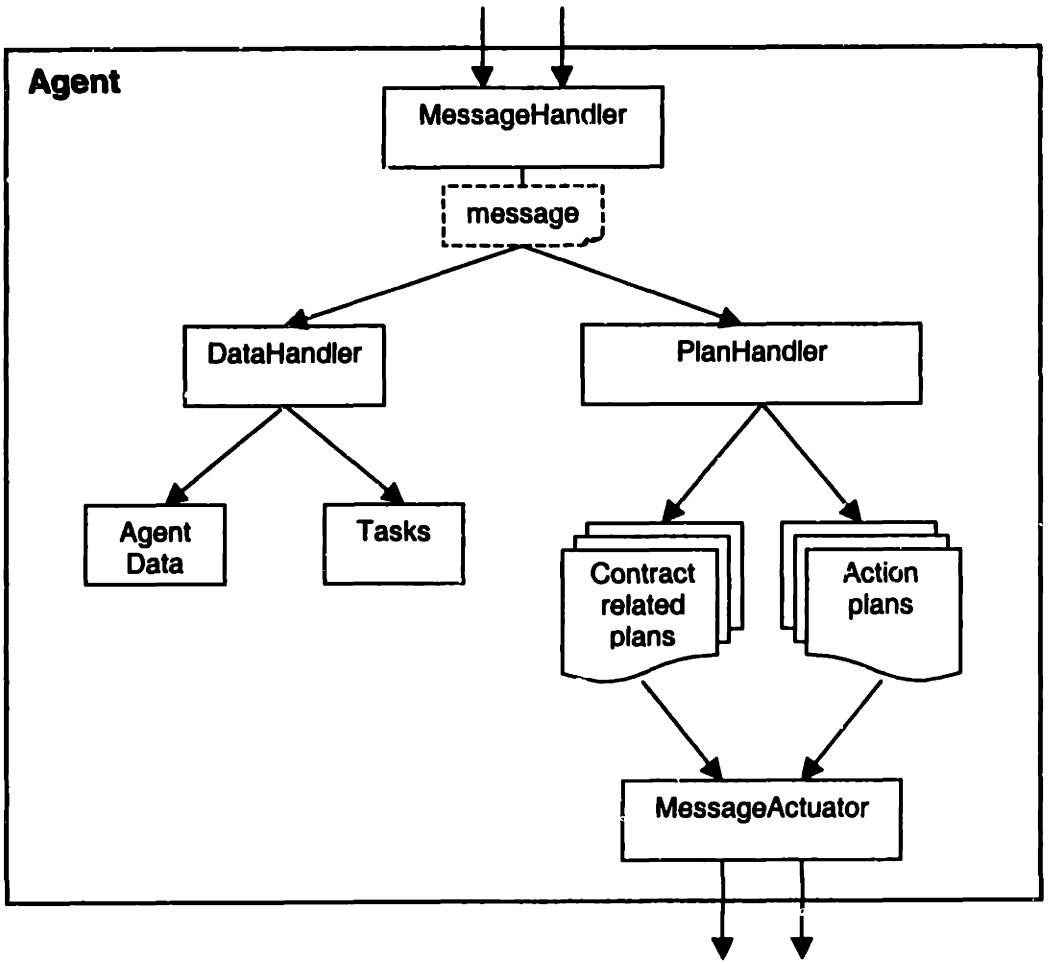
Exception	Symptom	Detection and Resolution	Anticipation and Avoidance
		Q: query interface	*A: action interface*
Message congestion	Delayed messages		Anticipation: Q: query infrastructure for communication link load and detect when it is heavy
			Avoidance: Suspend lower priority tasks in the hope that it will reduce the number of generated messages • A: instruct subcontractor of lower priority tasks to suspend task execution • After higher priority tasks are completed, A: instruct subcontractor of lower priority tasks to resume task execution
Agent death	Silent and unresponsive agent	Detection: Periodically poll agents upon receiving or granting an award every x time units since their last message and detect when unresponsive Diagnosis: Poll other agents residing in the same host; if they are not responsive, message congestion is more likely to be the cause Note that this is only relevant if we assume that messages between the agent and exception handler also suffer from network congestion	Anticipation: Record each agent death event and detect when failure frequency of the associated source code exceeds a threshold value
		Resolution: Instruct the corresponding contractor and subcontractor to abort or re-announce properly in order to avoid abandoned or wasted tasks • If agent is a contractor, A: instruct subcontractor and also subcontractors of all generated subtasks to abort task execution • If agent is a subcontractor, A: instruct subcontractors of all generated subtasks to abort task execution and instruct its contractor to re-announce • If it is possible to recover subtask results, collect these results, send illusory bid, and forward them to the replacement contractor	Avoidance: 1. Blacklist agent (provides Better Business Bureau-like services) 2. A: instruct agent to kill itself

Exception	Symptom	Detection and Resolution	Anticipation and Avoidance
		Q: query interface	*A: action interface*
Fault in task execution	Contractor's complaint	Detection: • Detect complaints from contractor upon receiving task results • Diagnose the agent causing faulty results by traversing the task chain	Anticipation: Record each fault in task execution and detect when frequency of faulty results caused by an agent exceeds a threshold value
		Resolution: Instruct the corresponding contractor to re-announce • If it is possible to recover subtask results, collect these results, send illusory bid, and forward them to the replacement contractor	Avoidance: 1. Blacklist agent (provides Better Business Bureau-like services) 2. A: instruct agent to kill itself
Delay in task execution	Late result		Anticipation: • Q: query infrastructure for each host processor load • If load is heavy, Q: query agents residing on the host for percent done and detect when any agents will not be able to meet task execution deadline
			Avoidance: Move the agent to another host if appropriate • Q: query task's contractor for QoS • If late result still has some values, A: move subcontractor to another host If not, A: instruct subcontractor to abort task execution
Protocol mismatch	Cancel-savvy (fickle) agents are involved in a contract with non-cancel-savvy agents	Detection: Detect award and bid cancellations	Anticipation: Detect i) when a contractor who does not support bid cancellation receives bids from fickle subcontractors or ii) when a fickle contractor receives bids from non-fickle subcontractors
		Resolution: Instruct the corresponding contractor and subcontractor properly in order to avoid abandoned or wasted tasks • Bid cancellation: instruct contractor to re-announce task • Award cancellation: instruct subcontractor and subcontractors of all generated subtasks to abort task execution	Avoidance: Avoid interactions between these agents or resort to detection & resolution if this is not possible • In the first case: intercept bid messages from fickle subcontractors if contractor has received other bids from non-fickle subcontractors • In the second case: intercept bid messages from non-fickle subcontractors if contractor has received other bids from fickle subcontractors

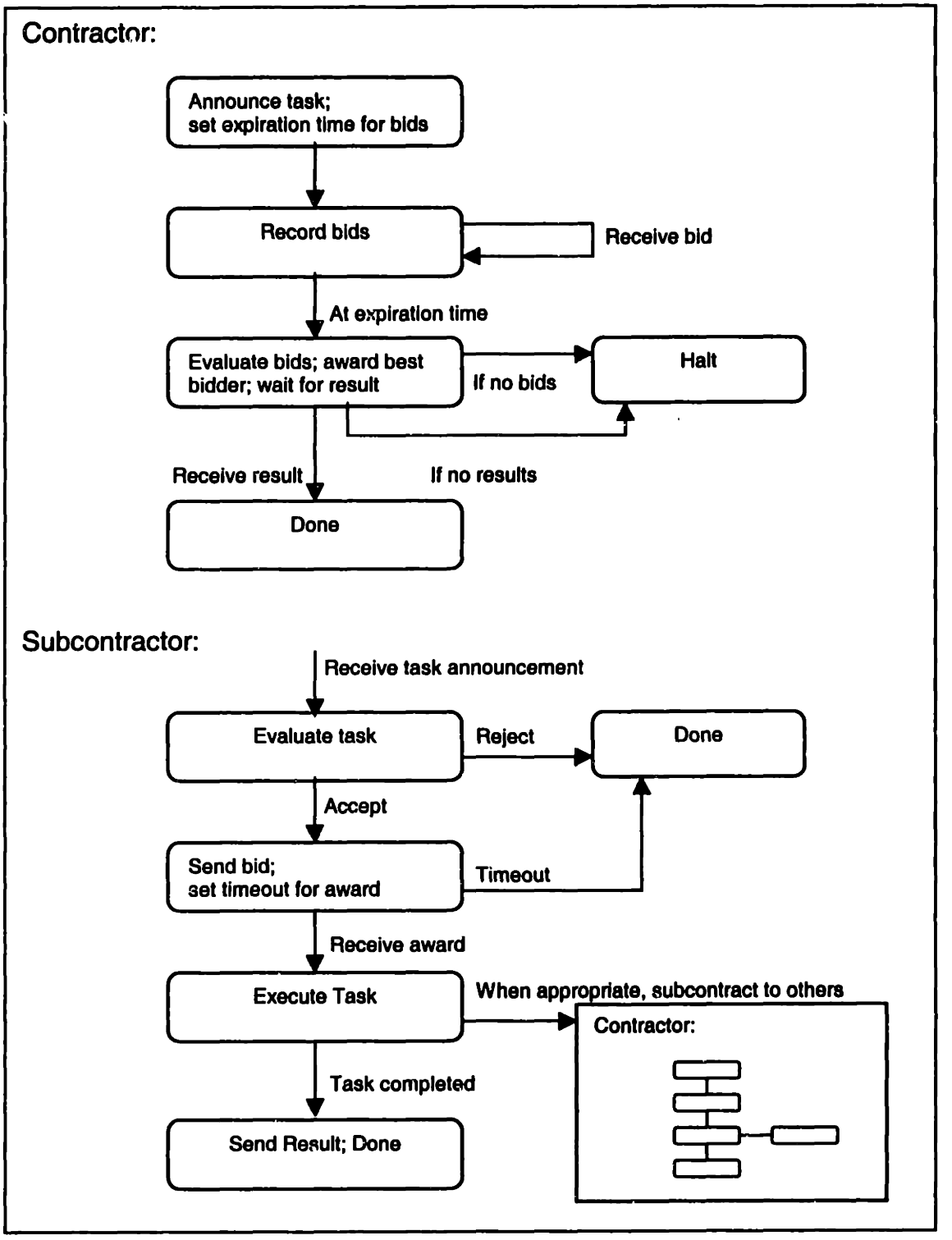
Appendix C

Agent Design and Code

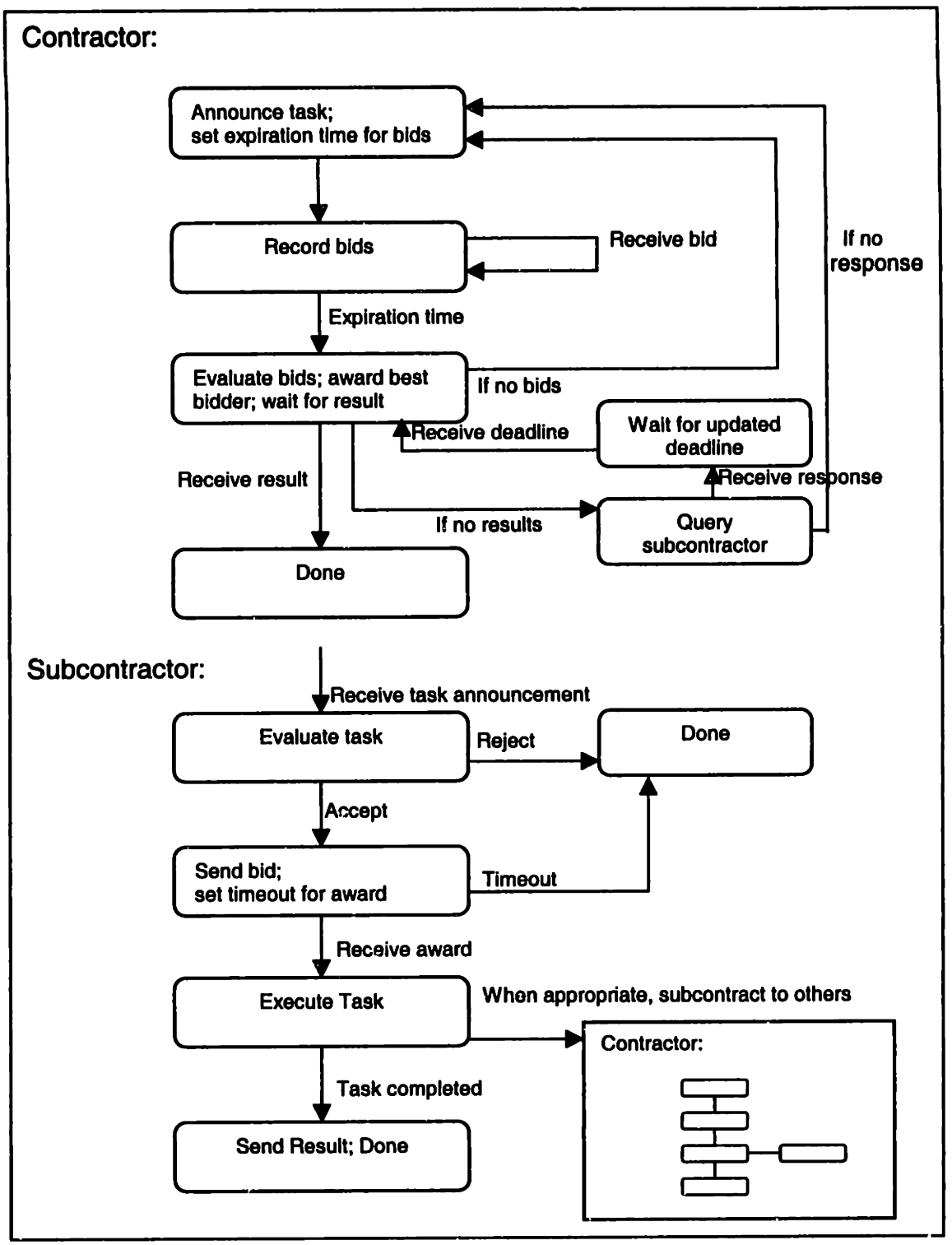
C.1 Agent Design



C.2 Simple Contract Net Protocol



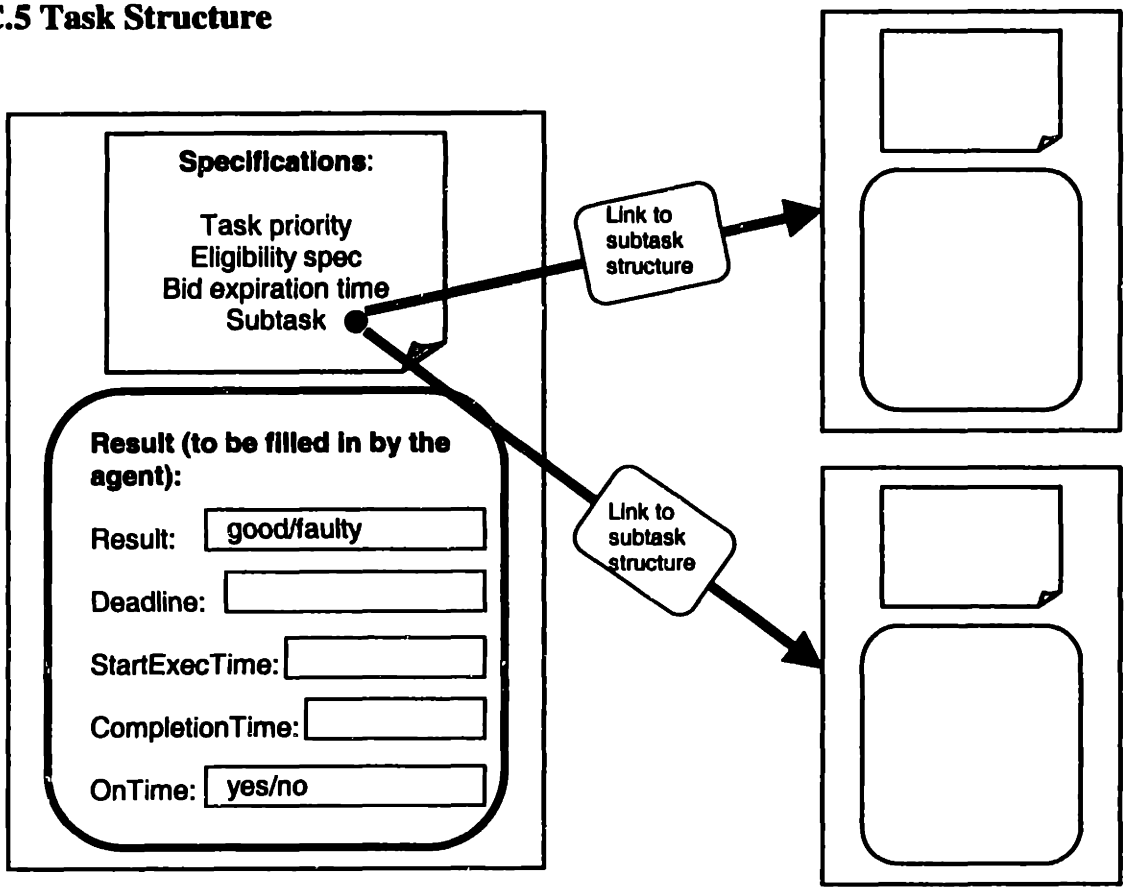
C.3 Augmented (Survivalist) Contract Net Protocol



C.4 Message Types and Fields

Message types	Fields
Default	String type String sender String receiver long timestamp
Advertise *Agent advertises its eligibility list and type to matchmaker*	Enumeration eliglist int sendertype
RequestAgents *Agent requests matchmaker for eligible agents w/ a certain eligibility specification*	String eligibilityspec ContractID contractid
EligibleAgents *Matchmaker's response to RequestAgents*	Enumeration eligibleagents ContractID contractid
TaskAnnouncement *Agent announces task to eligible agents*	ContractID contractid int sendertype long expirationtime int taskpriority long taskdeadline String eligibilityspec
Bid *In bid, agent indicates how long it'll take to complete task (nodeabstraction)*	ContractID contractid long nodeabstraction
Award *In award, agent includes task specification*	ContractID contractid Task taskspec
Result	ContractID contractid String resultquality (good or faulty)
Complaint *Agent sends a complaint to ExceptionHandler if it receives a faulty result from its subcontractor*	ContractID contractid
Query *Query message from Exception Handler to agent*	String querytype ("agentstatus") long queryid
QueryResponse *Query response from agent to Exception Handler*	String querytype long queryid
Action *Action message from Exception Handler to agent*	String actiontype +other parameters

C.5 Task Structure



References

Barnett J.A. (1984). "How Much Is Control Knowledge Worth? A Primitive Example." Artificial Intelligence 22 (1): 77-89.

Chia M.H., Neiman D.E., Lesser V.R. (1998). "Poaching and Distraction in Asynchronous Agent Activities." Proceedings International Conference on Multi Agent Systems. IEEE Comput. Soc., pp.88-95. Los Alamitos, CA.

Conry S., Meyer R., and Lesser V. (1986). "Multistage Negotiation in Distributed Planning." COINS Technical Report 86-67, University of Massachusetts. Also appeared in 'Readings in Distributed Artificial Intelligence', A.H. Bond, L.Gasser (eds.), Morgan Kaufmann Publishers, pp. 367-384.

Davis R., Smith R. (1983). "Negotiation as A Metaphor for Distributed Problem Solving." Artificial Intelligence, vol. 20, n.1, pp. 63-109.

Decker K. and Lesser V. (1992). "Generalized Partial Global Planning," International Journal on Intelligent Cooperative Information Systems, 1(2):319-346.

Doyle J. (1996). "Toward rational planning and replanning: rational reason maintenance, reasoning economies, and qualitative preferences." Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative (Austin Tate, editor), Menlo Park, California: AAAI Press, 1996, pp. 130-135.

Klein M., Dellarocas C. (1999). "Exception Handling in Agent Systems." Proceedings of the Third International Conference on AUTONOMOUS AGENTS, Seattle, Washington.

Fatima S.S., Uma G. (1998). "An adaptive organizational policy for multi agent systems-AASMAN. Proceedings International Conference on Multi Agent Systems." IEEE Comput. Soc., pp.120-7. Los Alamitos, CA.

Fischer K., Muller J.P., Pischel M., Schier D. (1995). "A model for cooperative transportation scheduling." ICMAS-95 Proceedings. First International Conference on Multi-Agent Systems. AAAI Press., pp.109-16. Menlo Park, CA.

Georgeff M.P. (1987). "Planning." Annual Reviews of Computer Science, pp. 359-400. Also published in 'Readings in Planning,' J. Allen, J.Hendler and A. Tate (eds.), Morgan Kaufman, 1990.

Georgeff M. P., Lansky A. L. (1987). "Reactive reasoning and planning: An experiment with a mobile robot." In Proceedings of the Sixth National Conference on Artificial ntelligence, pages 677-682, Seattle, Washington.

Gruber T.R. (1989). "A Method for Acquiring Strategic Knowledge." Knowledge Acquisition 1(3): 255-277.

Horling B., Lesser V., Vincent R., Bazzan A., Xuan P. (1999). "Diagnosis as an Integral Part of Multi-Agent Adaptability", UMass Computer Science Technical Report 99-03.

Jennings N.R., Sycara K., and Wooldridge M. (1998). "A Roadmap of Agent Research And Development", Autonomous Agents and Multi-Agent Systems (1:1), pp. 7-38.

Kaminka G.A., Tambe M. (1998). "What is Wrong With Us? Improving Robustness Through Social Diagnosis", Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98). AAAI.

Klein M. (1989). "Conflict Resolution in Cooperative Design." Computer Science. Urbana-Champaign, IL., University of Illinois.

Klein M. (1991). "Supporting Conflict Resolution in Cooperative Design Systems." IEEE Systems Man and Cybernetics 21(6).

Klein M. (1997). "An Exception Handling Approach to Enhancing Consistency, Completeness and Correctness in Collaborative Requirements Capture." Concurrent Engineering Research and Applications.

Laird J. E., Newell A., Rosenbloom P. S. (1987). "Soar: An architecture for general intelligence", Artificial Intelligence, 33:1-64.

Lansky A.L. (1991). "Local Search for Multi-Agent Planning." Proceedings of the International Joint Conference on Artificial Intelligence.

Lesser V. (1998). "Reflections on the Nature of Multi-Agent Coordination and Its Implications for an Agent Architecture", Autonomous Agents and Multi-Agent Systems (1:1), pp. 89-111.

Maes P. (1995). "Modeling Adaptive Autonomous Agents." In: Artificial Life, An Overview, edited by Christopher G. Langton, MIT Press. Cambridge, MA.

Ohko T., Hiraki K., Anzai Y. (1996). "Reducing communication load on contract net by case-based reasoning-extension with directed contract and forgetting". ICMAS-96 Proceedings. Second International Conference on Multi-Agent Systems. AAAI Press, pp.244-51. Menlo Park, CA.

Ramos C. (1995). "Task negotiation for distributed manufacturing systems." Proceedings. IEEE International Symposium on Assembly and Task Planning. IEEE Comput. Soc. Press, pp.259-64. Los Alamitos, CA.

Saad A., Kawamura K., Biswas G., Johnson M.E., Salama A. (1995). "Evaluating a contract net-based heterarchical scheduling approach for flexible manufacturing." Proceedings. IEEE International Symposium on Assembly and Task Planning. IEEE Comput. Soc. Press, pp.147-52. Los Alamitos, CA.

Sandholm T.W., Lesser V.R. (1995). "Issues in automated negotiation and electronic commerce: Extending the contract net framework." ICMAS-95 Proceedings. First International Conference on Multi-Agent Systems. AAAI Press. Menlo Park, CA.

Shoham Y. (1990). "Agent-oriented programming." Technical Report STAN-CS-1335-90, Computer Science Department, Stanford University, Stanford, CA.

Shue D. (1999). "SimHazard: agent-world exception simulator." Master's of Engineering Thesis, Massachusetts Institute of Technology, 1999.

Sierra C., Faratin P., Jennings N. (1997). "A Service-Oriented Negotiation Model between Autonomous Agent." Proc. 8th European Workshop on Modeling Autonomous Agents in a Multi-Agent World (MAAMAW-97), Ronneby, Sweden, 17-35.

Smith R.G. (1980). "The contract net protocol: high level communication and control in a distributed problem solver." IEEE Transactions on Computers, vol.C-29, no.12, pp.1104-13.

Stirling W. (1992). "Multi-Agent Coordinated Decision-Making Using Epistemic Utility Theory". Proceedings of the MAAMAW, National Research Council, Institute of Psychology, Rome, Italy.

Tenney R.R., Jr., Sandell N.R. (1981). "Strategies for Distributed Decisionmaking." IEEE Transactions on Systems, Man and Cybernetics, pp.527-538.