An Experimental Evaluation of Domain-Independent Fault Handling Services in Open Multi-Agent Systems

Chrysanthos Dellarocas and Mark Klein

CCS WP #212 SWP # 4116

May 2000

An Experimental Evaluation of Domain-Independent Fault Handling Services in Open Multi-Agent Systems[1]

CHRYSANTHOS DELLAROCAS, PHD
Sloan School of Management
Massachusetts Institute of Technology
dell@mit.edu
http://ccs.mit.edu/dell

MARK KLEIN, PHD
Center for Coordination Science
Massachusetts Institute of Technology
m_klein@mit.edu
http://ccs.mit.edu/klein/

Abstract. A critical challenge to creating effective open multi-agent systems is allowing them to operate effectively in the face of potential failures. In this paper we present an experimental evaluation of a set of domain-independent services designed to handle the failure modes ("exceptions") that can occur in such environments, applied to the well-known "Contract Net" multi-agent system coordination protocol. We show that these services can produce substantially more effective fault handling behavior than standard existing techniques, while allowing simpler agent implementations.

Keywords: contract net, failure detection, failure resolution, fault handling, multiagent

[1] Appears in the Proceedings of the 2000 International Conference on Multi-Agent Systems, Boston MA.

1. The Challenge

A critical challenge to creating effective agent-based systems is making them robust in the face of potential failures. Most work to date on multi-agent systems has focused, however, on supporting such basic functionality such as matchmaking (Decker, Sycara et al. 1997) and inter-agent communication (Finin, Labrou et al. 1997), and has typically assumed relatively simple closed environments where the infrastructure is reliable and agents can be trusted to work correctly (Hägg 1996). It is clear however that in many if not most important applications for multi-agent technology, these assumptions will not hold. We can expect, in contrast, to find:

- *Unreliable Infrastructures.* In large distributed systems like the Internet, unpredictable node and link failures may cause agents to die unexpectedly, messages to be delayed, garbled or lost, etc.

- *Non-compliant agents.* In open systems, agents are developed independently, come and go freely, and thus can not always be trusted to follow the rules properly due to bugs, bounded rationality, programmer malice and so on. This can be expected to be especially prevalent and important in contexts such as electronic commerce where there may be significant incentives for fraud.

- *Emergent dysfunctions.* Emerging multi-agent system applications are likely to involve complex and dynamic interactions that can lead to emergent dysfunctional behaviors with the relatively lightweight multi-agent coordination mechanisms that have proved most popular to date. This is especially true since agent societies operate in a realm where relative coordination, communication and computational costs and capabilities can be radically different from those in human society, leading to behaviors with which we have little previous experience. It has been argued, for example, that 1987's stock crash was due in part to the action of computer-based "program traders" that were able to execute trade decisions at unprecedented speed and volume, leading to unprecedented stock market volatility (Waldrop 1987).

All of these departures from "ideal" multi-agent system behavior can be called *exceptions*, and the results of inadequate exception handling include the potential for poor performance, system shutdowns, and security vulnerabilities.

In this paper we present an experimental evaluation of a set of domain-independent exception handling services we have developed to address these challenges, applied to the well-known "Contract Net" multi-agent coordination protocol. We show that these services produce more effective exception handling behavior than standard existing techniques, while allowing simpler agent implementations. The remainder of this paper will introduce the contract net protocol, outline our exception handling approach, describe the experiments used to evaluate it, consider the contributions of this work, and discuss directions for future research.

2. The Contract Net Protocol

The "Contract Net" (Smith and Davis 1978) (henceforth called CNET) is a protocol for matching up tasks with agents in multi-agent systems. CNET and its many variants is probably the most widely-used agent system protocol, presumably because of its intuitiveness, direct applicability to many common problems, simplicity and relative efficiency. CNET has been applied to many domains including manufacturing control (Baker 1988), tactical simulations (Boettcher, Perschbacher et al. 1987), transportation scheduling (Bouzid and Mouaddib 1998), and distributed sensing (Smith and Davis 1978).

The CNET protocol operates as follows (Figure 1):

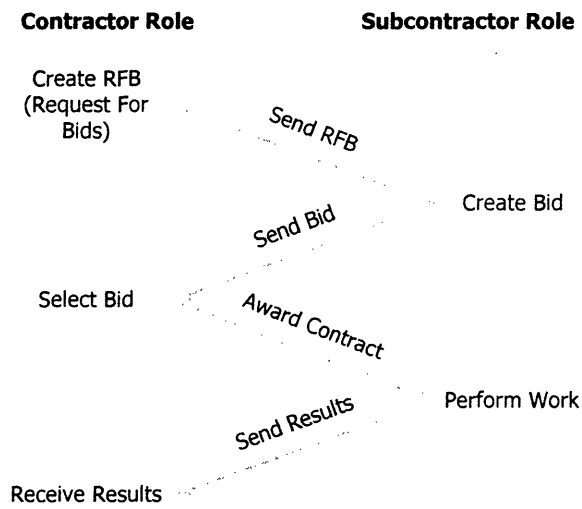


Figure 1. A simple version of the Contract Net protocol.

An agent (hereafter called the "contractor") identifies a task that it cannot or chooses not to do locally and attempts to find another agent (hereafter called the "subcontractor") to perform the task. It begins by creating a Request For Bids (RFB) which describes the desired work, and then sending it to potential subcontractors (typically identified using a matchmaker that indexes agents by the skills they claim to have). Interested subcontractors respond with bids (specifying such issues as the time needed to perform the task) from which the contractor selects a winner. The winning agent, once notified of the award, performs the work (potentially subcontracting out its own subtasks as needed) and submits the results to the contractor.

CNET is prone to a wide range of potential exceptions from all three of the categories (unreliable infrastructures, non-compliant agents, emergent dysfunctions) described above. A more exhaustive analysis of these failure modes will appear in a forthcoming paper; for now we will limit ourselves to three examples:

- *Agent death:* If a CNET agent dies there are several immediate consequences. If the agent is acting as a subcontractor, its customer clearly will not receive the results it is expecting. In addition, if the agent has subcontracted out one or more subtasks, these subtasks and all the sub-sub-... tasks created to achieve them become

"orphaned", in the sense that there is no longer any real purpose for them and they are uselessly tying up potentially scarce subcontractor resources. Finally, if the system uses a matchmaker, it will continue to offer the now dead subcontractor as a candidate (a "false positive"), resulting in wasted message traffic.

- *Fraudulent [sub]contractor:* A buggy or intentionally malicious CNET agent can wreak havoc through fraudulent advertising, bidding or subcontracting. Imagine, for example, an agent that falsely informs the matchmaker that it has a comprehensive set of skills, sends in highly attractive but fraudulent bids (e.g. specifying it can do any task almost instantaneously) for all the RFBs it receives, and once it wins the awards returns either no results, or simply incorrect ones. The result would be that many if not all of the system's tasks would be awarded to a non-performing agent.

- *Resource poaching:* It is typical for CNET systems to annotate tasks with priorities, so that when a subcontractor is considering several RFBs, it will bid (first) for the RFB with the greatest priority. One emergent dysfunction that can occur in such contexts is "resource poaching", wherein a slew of low-priority but long-duration tasks tie up the subcontractors, thereby freezing out resources needed for the higher-priority tasks that arrive later (Chia, Neiman et al. 1998). This does not represent an error per se, but rather an unexpected consequence of the protocol when applied in a complex environment.

The standard exception handling mechanism used in CNET, as in many distributed protocols, is timeout/retry: If no results are received by the deadline the subcontractor promised, for example, a contractor will re-start the subcontracting process for that task, sending a new RFB. This approach does handle the agent death exception, but rather inefficiently, since it does not eliminate orphaned tasks, does not remove false positives from the matchmaker, and is prone to an "unzippering" effect, wherein the death of an agent performing a subtask can cause cascading timeouts and retries for its customers, the customers of its customers, and so on, all the way up to the CNET agent at the top of the task decomposition tree. The timeout/retry approach will not, of course, prevent a contractor from repeatedly falling prey to a fraudulent CNET agent, nor will it help with resource poaching.

It is certainly imaginable that the CNET protocol could be elaborated to allow agents to handle a wider range of exceptions, and most agent system exception handling research has in fact taken this direction. Even the original CNET protocol (Smith and Davis 1978) included such augmentations as an "immediate response bid', which allowed a contractor to determine whether the lack of bids was due to all eligible subcontractors being busy (in which case a retry is appropriate) or due to the outright lack of subcontractors with the necessary skills (in which case presumably the system manager/user should be informed). This "survivalist" approach to multi-agent exception handling faces, however, a number of serious shortcomings:

First of all, it greatly increases the burden on agent developers. It is predicated upon "compiling in" potentially complicated and carefully coordinated exception handling behaviors into all problem-solving agents. Developers must anticipate and correctly prepare for all the exceptions the agent may encounter in all the environments in which it may operate. This is difficult at best, as the agent environment may be difficult to anticipate, and in any case no systematic methodology has been available to help identify relevant exception types and resolution strategies. Making changes in exception handling behavior is difficult because it potentially requires coordinated changes in multiple agents created by different developers. Agents become harder to maintain, understand and reuse because a potentially large body of exception handling code obscures the relatively simple normative behavior of an agent.

Perhaps more seriously, this approach can result in poor exception handling performance. Agents may not comply properly with these more sophisticated protocols, or violate some of their underlying assumptions. Some protocols, for example, are based on game-theoretic analyses (Sandholm, Sikka et al. 1999) and assume that all agents will be rational utility maximizers, which obviously may not always be the case. All agent interactions are slowed down by the overhead incurred by these heavyweight protocols. Some kinds of interventions (such as "killing" a broken agent that is uselessly monopolizing scarce resources) may be difficult to implement because the agents do not have the established legitimacy needed to apply such interventions to their peers. Finally, finding the appropriate responses to some kinds of exceptions (typically emergent exceptions such as resource poaching) requires that the agents achieve a more or less global view of the multi-agent system state, which is notoriously difficult to create without heavy bandwidth requirements.

3. Our Approach: Domain-Independent Exception Handling Services

It is for this reason that we have been creating a set of services that offload the exception handling burden from problem solving agents. We call this the "citizen" approach by analogy to the way exceptions are handled in human society. In such contexts, citizens adopt relatively simple and optimistic rules of behavior, and rely on a whole host of social institutions (the police, lawyers and law courts, disaster relief agencies, the Security and Exchange Commission, the Better Business Bureau, and so on) to handle most exceptions. This is generally a good tradeoff because such institutions are able, by virtue of specialized expertise, widely accepted legitimacy and economies of scale, to deal with exceptions more effectively and efficiently than individual citizens, while making relatively few demands of them (e.g. pay your taxes, obey police officers, report crimes).

The key insight that makes this approach workable in the multi-agent system context is the simple but powerful notion that highly reusable, *domain-independent* exception handling expertise can be usefully separated from the knowledge used by agents to do their "normal" work. There is substantial evidence for the validity of this claim. Early work on expert systems development revealed that it is useful to separate domain-specific problem solving and generic control knowledge (Barnett 1984; Gruber 1989). Analogous insights were also confirmed in the domains of collaborative design conflict management (Klein

1989; Klein 1991) and workflow exception management (Klein 1997). We have identified over 100 such strategies to date; in the CNET domain, some examples include:

- To detect agent death, periodically poll active subcontractors. Consolidate polling in order to minimize the number of "are you alive?" messages agents must respond to. If an agent dies, cancel any orphaned tasks that may result, clear the agent record from the matchmaker(s), and instruct the contractors for that agent to re-run the bidding process for the failed tasks. Keep track of agent MTBF (mean time between failure) statistics to help avoid relying on unreliable agents in the future.
- To uncover fraudulent agents, keep track of their performance over time and note which agents consistently fail to produce results with the contracted cost, quality and duration. To foil fraudulent agents, "kill" or "exile" them when they appear.
- To detect resource poaching, compare the average priority of pending tasks to the average priority of in-work tasks. To resolve it, instruct subcontractors to preempt current lower priority tasks and bid for the pending higher priority tasks.

This expertise is exploited, in our approach, by a knowledge-based exception handler (EH) service, whose functional architecture is depicted in Figure 2.

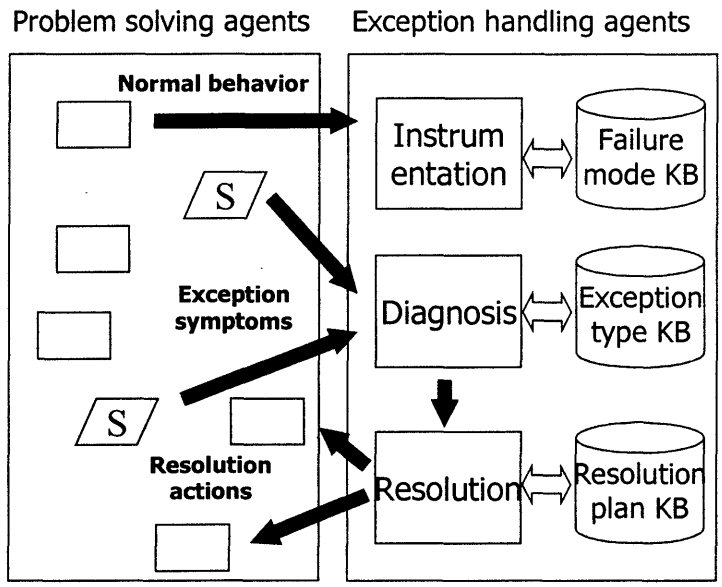


Figure 2. Functional architecture for the exception handling service.

When new agents register themselves they specify the coordination protocol they use (e.g. the CNET protocol). The EH service then consults its "failure mode" knowledge base to create the appropriate "sentinel agents" needed to detect the exceptions characteristic of that protocol. Sentinels work by monitoring communication between agents and by pro-actively querying agents about their status when necessary. Whenever they detect a likely exception symptom, they trigger the diagnostic component of the EH service, which uncovers the underlying cause and enacts the interventions, instantiated from the resolution plan knowledge base, needed to resolve the problem. The EH service requires that agents

support at least a minimal set of EH-related message types (e.g. an "are you alive?" query, a "cancel task" action, etc.), and relies upon several ancillary services such as an agent ID authority (which certifies unique IDs for each agent type), a reputation server (which tracks performance for each agent type), and a contract notary (that keeps track of which agents are performing which tasks for whom). See (Klein and Dellarocas 1999) and (Dellarocas and Klein 1999), as well as forthcoming papers from our group, for further details.

The potential advantages of this approach are manifold. "Citizen" agents can be implemented much more simply, relying on relatively lightweight and efficient coordination protocols. Exception handling performance can be improved by utilizing services that can efficiently gather any necessary global state information, rely on specialized knowledge bases to select appropriate interventions, and are uniquely possess the widely-accepted legitimacy needed to take such interventions as killing agents or canceling tasks.

4. The Experiments

We ran a series of experiments to test these claims in a multi-agent system running the CNET protocol. These experiments focused on the agent death exception, since it is clearly a potential problem in almost any real multi-agent system, and also because a well-accepted "survivalist" approach to this exception (i.e. timeout/retry) exists for comparison purposes.

The experiments all take place in a discrete event based multi-agent system simulator built on top of the Swarm Simulation System (Minar, Burkhart, Langton and Askenazi 1996). Our system allows one to emulate a world consisting of multiple host computers, each running one or more agents and connected by network links, all with controllable speed and failure frequency. The scenario consists of several dozens CNET agents, one per host, interacting over a reliable network. Contractor agents send out an RFB with a specified timeout period: potential subcontractors bid only if they become available during this period (i.e. subcontractors perform only one task at a time). Bids are binding, which means that subcontractors will bid on a new RFB only after the timeout for its pending bid expired without an award being received (presumably because some other subcontractor won the task). Contractors select the winning bids based solely on how quickly the bidders claimed they could perform the task. Contractors re-send RFBs if no bids have been received by the timeout period (presumably because no subcontractors with the needed skills were available at that time). This CNET protocol is modeled on the one described in (Smith and Davis 1978) and was chosen because it is simple and was shown by Smith et al. to represent a reasonable design tradeoff in several test domains.

Our experiments explored the effect of three experimental conditions. The key independent variable, of course, was whether the agents took a "survivalist" or "citizen" approach to handling agent death. Survivalist agents rely on the standard timeout/retry mechanism to handle agent death: If a subcontractor does not return results to its contractor by the agreed-upon deadline, the contractor issues a new RFB for the task. Citizen agents, by contrast, rely entirely on the EH services. Whenever a task has been awarded to a

subcontractor, the EH service begins periodic polling of the subcontractor to check whether it is still alive, which continues until the agent has died or returned the task results to its contractor. If an agent dies, the EH service takes a series of coordinated actions:

- It notifies the matchmaker that this agent is dead and should therefore be removed from the list of available subcontractors. This handles the "false matchmaker positive" problem.
- If the agent is subcontracting to someone else, it immediately informs the contractor that it should re-send the RFB for that task, thereby ensuring that the contractor does not waste time waiting for results from a dead agent. Note that this avoids the "unzippering" effect described above, because contractors will only re-send the RFB when the subcontractor has actually died.
- If the agent is a contractor for some pending subtasks, a proxy agent is created to try to find new customers for those "orphaned" subtask results. The proxy registers itself with the matchmaker, so that it becomes eligible to receive RFBs. It then waits for an RFB for the orphaned tasks, and submits a bid whose estimated completion time accounts for the amount of time that has already been spent processing those tasks, and is therefore likely to be highly competitive. This is a reasonable strategy in domains where there is a standardized task decomposition, so the replacement for the dead agent is apt to require the same subtask results that the dead agent did. If the proxy wins the anticipated RFB, it forwards the results as it receives them. Otherwise it keeps responding to RFBs until it wins or until the task results become obsolete. This strategy is this designed to minimize wasted work on orphaned tasks. In domains where results get obsolete very quickly, or there is no standard task decomposition, it may be more appropriate to do without the proxy-bidding agent and simply kill all orphaned tasks when the ultimate customer for them has died.

The exception handler is capable of knowing the current contractor-subcontractor relationships either by monitoring communication among agents, or by accessing information contained in a "notary" agent, who records all contracts formed among agents in the system. In either case, this information can be collected without the need to add any complexity to individual contractor and subcontractor agents.

Our central hypothesis is that the "citizen" exception handling approach will significantly reduce the average amount of time needed to complete tasks in which exceptions occur (due to quicker detection of agent death, and the avoidance of the unzippering effect), as well as reducing the overall system effort needed to perform tasks (by avoiding wasting resources on orphaned tasks). We also explored the related hypotheses that the impact of the EH services will depend on the nature of the task decompositions needed to perform a task. More specifically, tasks that have deep task decompositions should benefit more because in those cases the unzippering effect will be more severe with survivalist agents. Task decompositions with long duration tasks should benefit more because the quicker agent death detection offered by polling saves more time in absolute terms.

In order to test the above hypotheses, we tested the contract completion performance of four different agent configurations, whose parameters are summarized in Figure 3.

#	Description	# Top-Level Contractors	# Subcontractors	RFB timeout (cycles)	Task duration (cycles)
1	Short tasks, abundant subcontractors	3	50	100	1000
2	Short tasks, scarce subcontractors	3	16	100	10000
3	Long tasks, abundant subcontractors	3	50	100	1000
4	Long tasks, scarce subcontractors	3	16	100	10000

Figure 3. Summary of agent configurations tested.

In all configurations, top-level contractor agents execute a loop where they announce a new top-level task, wait for bids, award the contract to the best bidder, wait to receive the results and then stay idle a random amount of time before repeating the above steps.

In order to be completed, top-level tasks require the creation of task trees with depth 4 and branching factor 2 (Figure 4). In other words, in order to complete a top-level task, a top-level contractor has to seek two level-2 subcontractors, each of which has to seek two level-3 subcontractors, and so on. Therefore, a single top-level task may involve up to 15 agents working simultaneously. To simplify the experiment, it is assumed that any available subcontractor is capable of performing any task in a given task chain. The task duration mentioned in Figure 3 above is the number of cycles that a contractor has to spend *after* it has received the results of *both* its subcontracted tasks ("processing them"), before it returns *its* result to *its* contractor.

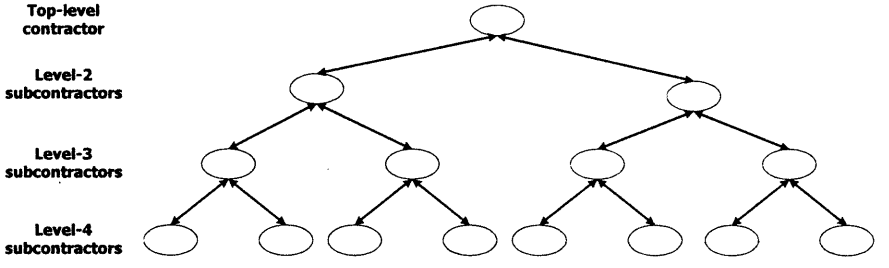


Figure 4. Top-level tasks require the creation of a 4-level task tree.

For each of the four above configurations, three distinct simulation runs were performed:

a. Failure-free environment (baseline case)
b. Failure-prone environment, "survivalist" agents
c. Failure-prone environment, "citizen" agents supported by EH services

In the failure-prone case, all subcontractor agents had a "lifespan" (time until death) that was selected for each agent randomly from a uniform distribution with probability $p=(100\text{xshort task duration})^{-1} = (10\text{xlong task duration})^{-1}$. When an agent dies, a new one is created with the same skills but with a different unique ID and is registered with the

matchmaker. This is done to keep the subcontractor population from shrinking over the course of the experiment, thereby emulating a large and dynamic agent pool where the population of subcontractors remains roughly constant.

Figure 5 summarizes the mean contract completion time relative to the failure-free (baseline) case for survivalist and citizen agents in each of the four agent configurations described above.

In configuration #1 (short tasks, abundant subcontractors), citizen agents with EH support clearly outperformed survivalist agents and managed to almost eliminate the effects of exceptions (mean completion time in the citizen case was less than 1% higher than the mean completion time in the failure-free case).

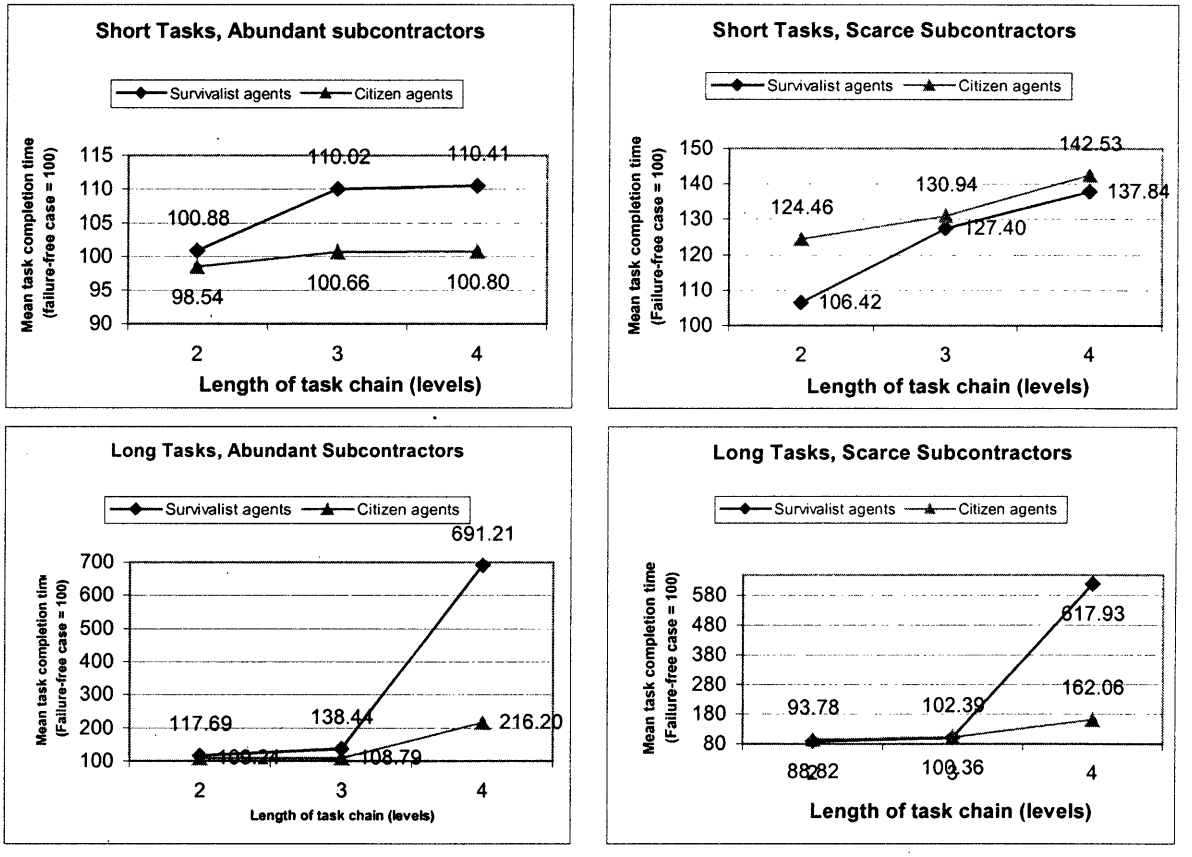


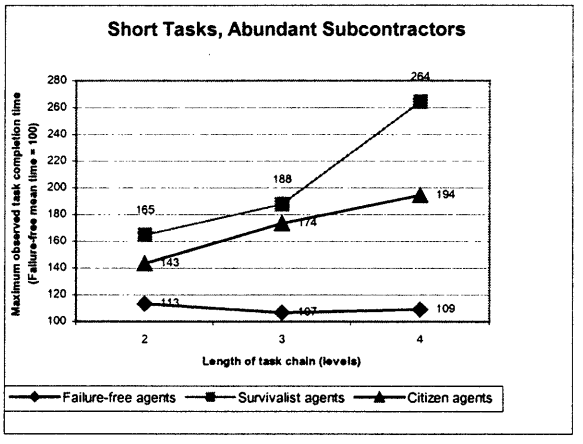
Figure 5. Mean contract completion times in the four agent configurations tested.

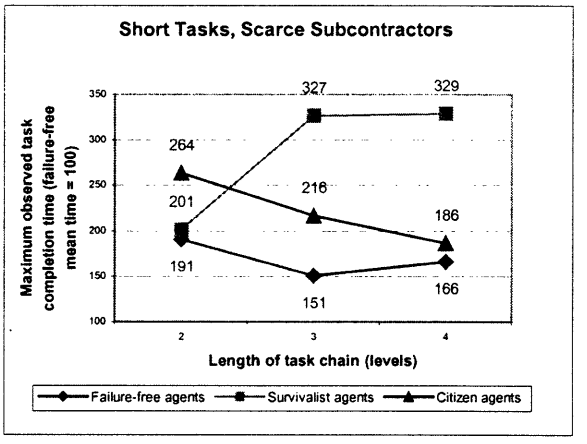
In configuration #2 (short tasks, scarce subcontractors) we were somewhat surprised to discover that survivalist agents outperformed citizen agents, especially for task chains of length 2 (e.g. a single level of contracting). This phenomenon can be explained by considering that, for short tasks (task length ~ 2 times the sentinel polling interval), the relative benefit that comes from timely detection of agent death is rather small. On the other hand, early detection causes more agents to re-announce their tasks and thus increases the number of agents who compete for the, already overloaded, scarce

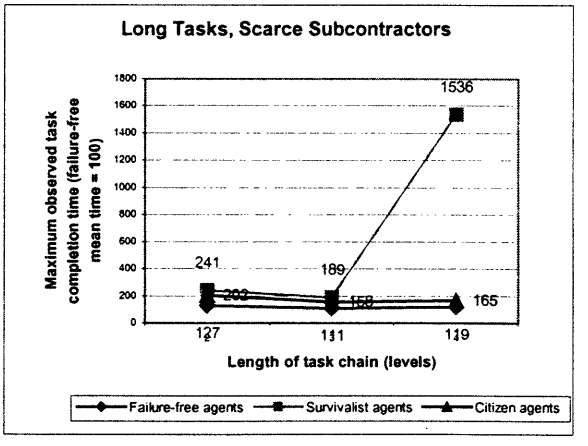
subcontractors, even more. The net effect is that the EH mechanism slightly worsens the bottleneck created by the scarce subcontractors. For longer task chains, the benefits of placing orphaned subtasks compensate for the previous effect and the performance of the system in citizen and survivalist cases seems to converge.

In configurations #3 and #4 (long tasks), citizens outperformed survivalists, as expected. The difference in performance was particularly dramatic for longer tasks chains. The explanation in this case is that, for longer tasks, the probability of multiple agent deaths in the same task tree is correspondingly higher. In the survivalist case, each death may trigger the "unzippering" effect described in Section 2, which would effectively double the task completion time. In the case of multiple deaths, the "unzippering" effect would be repeated, thus multiplying the mean completion time even more (to between 600% and 700% of the failure-free case). By avoiding the "unzippering" effect and attempting to find new contractors for orphaned tasks, the citizen case managed to keep the mean completion time at no more than 160% to 216% of the failure-free case.

In addition to the mean task completion time, we were also interested to compare the standard deviation and maximum observed task completion time in each of the above cases. Our rationale is that, in most environments, consistency is equally important to efficiency. A system with a low mean completion time, but where some task instances take a very long time to complete is bound to make some users extremely unhappy.







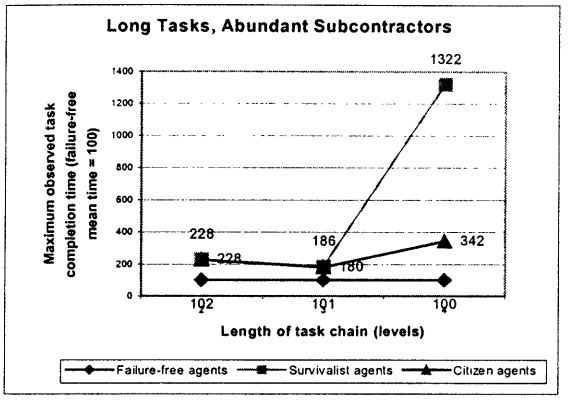


Figure 6. Maximum contract completion times in the four agent configurations tested.

Figure 6 summarizes the maximum observed task completion times in each of the tested configurations[2]. From a study of the charts it is clear that citizen agents had a lower maximum observed completion time in all four configurations.

In conclusion, citizen agents have proven to perform more efficiently than survivalist agents both in terms of lowering the mean as well as improving the consistency of contract completion time in the face of exceptions. In only one case (short tasks with scarce subcontractors) citizen agents performed slightly worse than survivalists. Even in that case, the difference seems to disappear as the task chain length increases.

The only implementation requirements from the part of citizen agents in order for them to participate in the exception handling scheme introduced in this paper is that they are capable of responding to periodic "are you alive?" messages sent by exception handling sentinels as well as to "cancel and reannounce task" messages sent when agent failures are detected. Both these message handlers are extremely simple to implement and well worth the added efficiency and consistency in the face of failures.

5. Contributions of the Work

The central contribution of the work presented here is to demonstrate empirically the potential value of domain-independent exception handling services for an important type of failure in multi-agent systems. As we have seen, this approach produces substantially superior performance without complicating agent development, and reduces our reliance on agents being correctly implemented.

These results make, we believe, a significant addition to the existing literature on reliability in multi-agent systems. As we have already noted, there has been relatively little previous work on multi-agent exception handling, and much of this has taken a "survivalist" approach, which has the important shortcomings identified above. Several lines of research

[2] Measurements of the standard deviation of the observed completion times are not included here due to the lack of space. However, they follow patterns very similar to those of the maximum observed completion times. The corresponding charts are available from the authors upon request.

have begun to explore concepts similar to those presented here, but none as far as we know have explored the combination of domain-independent exception handling implemented as distinct services. Hägg (Hägg 1996) presents the concept of sentinel agents; these are distinct services, which monitor the agent system and intervene when necessary by selecting alternative problem solving methods, excluding faulty agents, or reporting to human operators. This approach is not domain-independent, however: sentinels must be customized for each new application. Kaminka et al (Kaminka and Tambe 1998) present Social Attentive Monitoring (SAM), an exception handling approach wherein agents detect exceptions via uncovering violations of normative relationships with their teammates, and exploit a teamwork model to diagnose and fix these problems. This approach does have generic elements, but it is limited to teamwork protocols like TEAMCORE (Tambe M 1997) and requires domain-dependent customization of the exception detection procedures. Horling et al. (Horling, Lesser et al. 1999) have explored the use of domain-independent tools to detect and resolve the exception wherein the agents have a harmfully inaccurate picture of the inter-agent dependencies in their current context. This approach is limited to a single exception type, however, and like SAM applies to just one class of coordination protocol. Venkatraman et al (Venkatraman and Singh 1999) describe a generic approach to uncovering agents that do not comply with coordination protocols. This approach only addresses one subclass of exception types, however, and does not include a resolution component.

Related work can also be found if we go farther afield into such disciplines as planning, distributed systems, manufacturing process control, and the like. Distributed and real-time systems research has produced useful techniques such as checkpointing and rollbacks (Burns and Wellings 1996) (Mullender 1993), but these "one size fits all" techniques achieve generality at the cost of the efficiencies that can result from coordination-mechanism specific, albeit domain-independent, exception handling mechanisms. There has also been substantial work in the planning and robotics communities on dealing with unexpected world states (Traverso, Spalazzi et al. 1996) (Howe 1995) (Birnbaum, Collins et al. 1990) (Broverman and Croft 1987) (Firby 1987) (Hindriks, de Boer et al. 1998). This work focuses almost exclusively on exceptions (e.g. failed operations, unexpected events) in the world manipulated by the agents, and not on exceptions concerning the agents themselves. Finally, there has been substantial work on detecting and resolving exceptions in computer-supported cooperative work (Mi and Scacchi 1993) (Chiu, Karlapalem et al. 1997) (Klein 1998) (Auramaki and Leppanen 1989) (Finkelstein, Gabbay et al. 1994) and manufacturing control (Fletcher and Misbah 1999) (Adamides and Bonvin 1993) (Katz 1993) but this has been applied to a very limited range of domains (e.g. just software engineering or flexible manufacturing cell control) and exception types (e.g. just inappropriate task assignments).

In an important sense we can say that the approach presented in this paper attempts to subsume much of the previous work in this area, in that our goal is to provide a unifying framework to exploit exception handling techniques derived from multiple disparate disciplines, for the benefit of improved robustness in open multi-agent systems.

6. Future Work

We plan to pursue two concurrent lines of development in this work. One line will include empirically and analytically evaluating different "survivalist" and "citizen" exception handling approaches for a wider range of exception types and coordination protocols. It seems clear that there are situations where a survivalist approach will be superior. Consider for example the "refused award" exception, wherein a subcontractor declines to accept an award from a contractor. This can happen, for example, if the award from the contractor has been delayed for so long that the subcontractors' binding bid expired and it accepted some other award in the interim. The EH services can detect this exception using a timeout (i.e. there is no result by the result deadline) or by directly querying the subcontractor after the award to see if it accepted the task, but the quickest and least message-intensive approach is simply for the subcontractor to let the contractor know it refused it's award. We would like to understand these tradeoffs and incorporate the insights so gleaned into our growing knowledge base of domain-independent exception handling expertise.

A second line of work will be to increase the power and scope of our generic exception handling technologies. We plan to extend to knowledge base underlying our tools to cover a wider range of coordination mechanisms. This is a task of significant but reasonable scope because there are relatively few multi-agent coordination protocols in use, and much of the variation in them is due to differences in exception handling, as opposed to the "core" coordination capability. We have already performed preliminary analyses of the characteristic exceptions and domain-independent exception handling strategies for the multi-level (Durfee and Montgomery 1990) and team-based (Tambe 1997) coordination protocols. Other areas for development include the design of improved diagnostic algorithms, including potentially model-based approaches, dealing with exceptions in the EH service itself, and enabling effective interoperation between the EH services and agents with survivalist capabilities.

The long-term goal of these efforts is to integrate these lines of work, and thereby provide multi-agent system developers with a comprehensive knowledge base of well-founded design guidelines, along with a suite of domain-independent component technologies that enable them to much more easily develop more robust open agent systems.

Acknowledgements

This work was supported by NSF grant IIS-9803251 (Computation and Social Systems Program) and by DARPA grant F30602-98-2-0099 (Control of Agent Based Systems Program).

References

Adamides, E. and D. Bonvin (1993). "Failure recovery of flexible production systems through cooperation of distributed agents." Ifip Transactions B: Computer Applications in Technology **11**: 227-38.

Auramaki, E. and M. Leppanen (1989). Exceptions and office information systems. Proceedings of the IFIP WG 8.4 Working Conference on Office Information Systems: The Design Process., Linz, Austria.

Baker, A. (1988). "Complete manufacturing control using a contract net: a simulation study.".

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

Birnbaum, L., G. Collins, et al. (1990). Model-Based Diagnosis of Planning Failures. Proceedings of the National Conference on Artificial Intelligence (AAAI-90).

Boettcher, K., D. Perschbacher, et al. (1987). "Coordination of distributed agents in tactical situations." IEEE(87CH2450): 1421-6.

Bouzid, M. and A.-I. Mouaddib (1998). "Cooperative uncertain temporal reasoning for distributed transportation scheduling." Proceedings International Conference on Multi Agent Systems.

Broverman, C. A. and W. B. Croft (1987). Reasoning About Exceptions During Plan Execution Monitoring. Proceedings of the National Conference on Artificial Intelligence (AAAI-87).

Burns, A. and A. Wellings (1996). Real-Time Systems and Their Programming Languages, Addison-Wesley.

Chia, M. H., D. E. Neiman, et al. (1998). Poaching and distraction in asynchronous agent activities. Proceedings of the Third International Conference on Multi-Agent Systems, Paris, France.

Chiu, D. K. W., K. Karlapalem, et al. (1997). Exception Handling in ADOME Workflow System. Hong Kong, Hong Kong University of Science and Technology.

Decker, K., K. Sycara, et al. (1997). Middle-agents for the Internet. Proceedings of IJCAI-97, Nagoya, Japan.

Dellarocas, C. and M. Klein (1999). Towards civil agent societies: Creating robust, open electronic marketplaces of contract net agents. Proceedings of the International Conference on Information Systems (ICIS-99), Charlotte, North Carolina USA.

Durfee, E. H. and T. A. Montgomery (1990). A Hierarchical Protocol for Coordinating Multiagent Behaviors.

Finin, T., Y. Labrou, et al. (1997). KQML as an agent communication language. Software Agents. J. Bradshaw. Cambridge MA, MIT Press.

Finkelstein, A., D. Gabbay, et al. (1994). "Inconsistency Handling in Multi-perspective Systems." IEEE Transactions on Software Engineering **20**(8): 569-578.

Firby, R. J. (1987). An Investigation into Reactive Planning in Complex Domains. Proceedings of AAAI-87.

Fletcher, M. and D. S. Misbah (1999). "Task rescheduling in multi-agent manufacturing." Proceedings. Tenth International Workshop on Database and Expert Systems Applications. DEXA **99**: 689-94.

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

Hägg, S. (1996). A Sentinel Approach to Fault Handling in Multi-Agent Systems. Proceedings of the Second Australian Workshop on Distributed AI, in conjunction with Fourth Pacific Rim International Conference on Artificial Intelligence (PRICAI'96), Cairns, Australia.

Hindriks, K., F. de Boer, et al. (1998). "Failure, monitoring and recovery in the agent language 3APL." Cognitive Robotics. Papers from the.

Horling, B., V. Lesser, et al. (1999). Diagnosis as an Integral Part of Multi-Agent Adaptability. Amherst, Massachussets, University of Massachussets at Amherst Department of Computer Science.

Howe, A. E. (1995). "Improving the reliability of artificial intelligence planning systems by analyzing their failure recovery." IEEE Transactions on Knowledge and Data Engineering **7**(1): 14-25.

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

Katz, D. M., S. (1993). Exception management on a shop floor using online simulation. Proceedings of 1993 Winter Simulation Conference - (WSC '93), Los Angeles, CA, USA, IEEE; New York, NY, USA.

Klein, M. (1989). Conflict Resolution in Cooperative Design. PhD thesis. 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): 1379-1390.

Klein, M. (1997). Exception Handling in Process Enactment Systems. Cambridge MA, MIT Center for Coordination Science.

Klein, M. (1998). A Knowledge-Based Approach to Handling Exceptions in Workflow Systems. Cambridge MA USA, MIT Center for Coordination Science.

Klein, M. and C. Dellarocas (1999). Exception Handling in Agent Systems. Proceedings of the Third International Conference on AUTONOMOUS AGENTS (Agents '99), Seattle, Washington.

Mi, P. and W. Scacchi (1993). Articulation: An Integrated Approach to the Diagnosis, Replanning and Rescheduling of Software Process Failures. Proceedings of 8th Knowledge-Based Software Engineering Conference, Chicago, IL, USA, IEEE Comput. Soc. Press; Los Alamitos, CA, USA.

Minar, N., Burkhart, R., Langton, C., Askenazi, M., The Swarm Simulation System: A Toolkit for Building Multi-Agent Systems, Santa Fe Institute Working Paper 96-06-042, Santa Fe, NM, 1996.

Mullender, S. J. (1993). Distributed systems. New York

Sandholm, T., S. Sikka, et al. (1999). Algorithms for Optimizing Leveled Commitment Contracts. Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden.

Smith, R. G. and R. Davis (1978). "Applications Of The Contract Net Framework: Distributed Sensing." Distributed Sensor Nets: Proceedings of a Workshop.

Smith, R. G. and R. Davis (1978). "Distributed Problem Solving: The Contract Net Approach." Proceedings of the 2nd National Conference of the Canadian Society for Computational Studies of Intelligence.

Tambe, M. (1997). "Towards flexible teamwork." Journal of Artificial Intelligence Research **7**: 83-124.

Traverso, P., L. Spalazzi, et al. (1996). "Reasoning about acting, sensing and failure handling: a logic for agents embedded in the real world." Intelligent Agents II. Agent Theories, Architectures, and Languages. IJCAI'95 Workshop.

Venkatraman, M. and M. P. Singh (1999). "Verifying Compliance with Commitment Protocols: Enabling Open Web-Based Multiagent Systems." Autonomous Agents and Multi-Agent Systems **3**(3).

Waldrop, M. (1987). "Computers amplify Black Monday." Science **238**: 602-604.