

Mobile Agent Fault Tolerance for Information Retrieval Applications: An Exception Handling Approach

Simon Pears, Jie Xu and Cornelia Boldyreff
Computer Science, University of Durham, Durham, DH1 3LE, UK
s.p.pears, jie.xu, cornelia.boldyreff@durham.ac.uk

Abstract

Maintaining mobile agent availability in the presence of agent server crashes is a challenging issue since developers normally have no control over remote agent servers. A popular technique is that a mobile agent injects a replica into stable storage upon its arrival at each agent server. However, a server crash leaves the replica unavailable, for an unknown time period, until the agent server is back on-line. This paper uses exception handling to maintain the availability of mobile agents in the presence of agent server crash failures. Two exception handler designs are proposed. The first exists at the agent server that created the mobile agent. The second operates at the previous agent server visited by the mobile agent. Initial performance results demonstrate that although the second design is slower it offers the smaller trip time increase in the presence of agent server crashes.

Key Words: Exception handling, fault tolerance, mobile agents, performance evaluation, server crash failures

1. Introduction

Distributed systems and on a wider scale, the Internet, are inherently complex in the presence of asynchrony, concurrency and distribution. Mobile agents introduce new levels of complexity, operating within an environment that is autonomous, open to security attacks (directed by malicious agents and hosts), agent server crashes, and failure to locate resources [1]. There is significant attention within the mobile agent fault tolerance community concerning the loss of mobile agents at remote agent servers that fail by crashing [1][2][3][4][5][6][7]. A mobile agent is lost when:

- An agent server crashes during execution;
- A mobile agent spawns a child and subsequently migrates to a remote agent server that crashes. The child completes its execution but is unable to communicate its results to the parent;

- A mobile agent spawns a child and subsequently migrates to a remote agent server. The child is lost when its agent server crashes. In this case the parent blocks waiting for its child to report back.

Resilience against agent server crashes is vital since a mobile agent may represent a single point of failure, whereby all its internal state is lost. For example a user may dispatch a mobile agent to visit airlines to find the cheapest flight from London to Mexico. The mobile agent migrates to each airline's remote agent server and dynamically updates its own internal state to reflect the current cheapest price. Unfortunately if an agent server crashes during mobile agent execution all information relating to the cheapest flight is lost. Some applications introduce failure dependencies with agent servers, i.e. execution of a mobile agent modifies both its internal state and the state of the agent server. Consequently transaction processing must be provided at agent servers.

A solution for mobile agent information retrieval applications obviously does not require transaction processing since there are no state dependencies introduced between the mobile agent and remote agent servers. Consequently transaction-based solutions [1][2][5][6] introduce unnecessary performance overheads. Some techniques [7] modify the agent server platform to introduce fault tolerance, e.g. an agent server may replicate each mobile agent before execution. However this is a considerable problem for information retrieval applications where mobile agents consume information from agent servers at remote (often different) enterprises. Introducing fault tolerance into the agent server platform restricts information retrieval to those enterprises that allow the modified agent server platform. In [6], Vogler *et al.* propose that a mobile agent inject a replica into stable storage upon arriving at an agent server. However, in the event of an agent server crash, the replica remains unavailable for an unknown time period.

This paper uses exception handling to maintain mobile agent availability in the presence of agent server crashes and examines the performance of two exception handler designs using an experimental case study application.

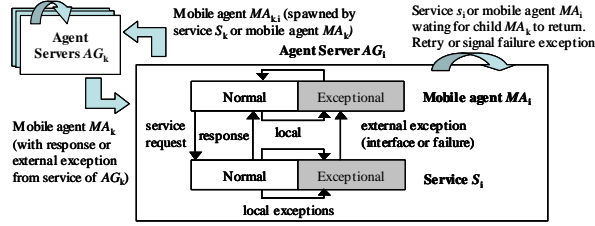


Figure 1 Mobile agent exception handling framework

Both exception handlers provide a solution that is agent server independent and offer continued service in the event of an agent server crash, using the following assumptions:

- mobile agents filter information, i.e. no state dependencies are introduced between the mobile agent and agent servers;
- no stable storage is provided at agent servers. (Stable storage and transaction processing will be addressed in a future paper.)

The remainder of the paper is structured as follows. Section 2 briefly introduces exception handling for mobile agents and outlines failure assumptions. Section 3 outlines the two exception handler designs. Section 4 describes the case study and section 5 then introduces the experiment. Section 6 analyses the performance results obtained from the case study experiment. Finally section 7 discusses related work and section 8 provides a conclusion.

2. Mobile agent fault tolerance framework

Exception handling with effective forward recovery can be perceived as a limited form of software fault tolerance [8], i.e. an operation that caused an error can be ignored, or a predefined degraded response is incorporated into the recovery handler.

This section highlights a model for mobile agent exception handling applicable to most mobile agent systems and application domains. Failure assumptions for handling crash failures are then presented.

2.1 Mobile agent exception model

A mobile agent is a computational entity that is capable of relocating its code and data to remote hosts to execute a task on behalf of its user. The sequence of hosts that a mobile agent visits is described by its *itinerary*. Weak mobility [9] is assumed, i.e. the mobile agent restarts its execution at each host. Each remote host runs an agent server platform that provides an execution environment for the mobile agent. The *home agent server* is where the mobile agent is created. All remote hosts visited by a mobile agent are assumed to execute the same

agent server platform. Figure 1 illustrates an adaptation of the exception model presented in [10] for mobile agents.

An agent server AG offers a set of services $S = \{s_1, s_2, \dots, s_n\}$. A service s_i is a software component that a mobile agent manipulates by issuing method calls (service requests). Both a service and mobile agent define its own set of internal or local exceptions $I = \{e_1, e_2, \dots, e_n\}$ and associated handlers $IH = \{h_1, h_2, \dots, h_n\}$ that serve to provide corrective action. An internal exception occurrence e_i triggers the exceptional activity h_i within the service or mobile agent. If the exception is successfully handled normal activity resumes and completes. A service completes its execution by providing a response to the mobile agent that made the service request. A mobile agent completes its activity by migrating to the next agent server in its itinerary.

A corrective action performed by a service or mobile agent in response to an internal exception is application specific and may involve dispatching a compensating mobile agent CM to interact with a service s_j at a remote agent server AG_j . For example a mobile agent may spawn a child to cancel a purchase made at agent server AG_{i-1} and locate a cheaper product because it exceeded its budget.

A service s_i signals a set of external exceptions $E = \{interface, failure\}$ to a mobile agent when it fails to satisfy the service request. There are two classifications of external exceptions:

- 1) *interface*: input values supplied by the mobile agent violate the pre-conditions of the service contract.
- 2) *failure*: the service is unable to provide a suitable response. Alternatively a service may be unable to authorise a request or there may be insufficient resources at the agent server to perform the service.

Upon receiving an external exception the mobile agent may retry service s_i , locate a service at an alternative agent server or report back to the home agent server or parent mobile agent.

Figure 1 illustrates that the exception handling framework is recursive. For example, service s_i at agent server AG_i may spawn mobile agent MA_k to visit agent server AG_k in reaction to a request made by mobile agent MA_i . Similarly mobile agent MA_i may spawn child MA_k to perform a delegated task such as information retrieval. Consequently the *owner* of a mobile agent is either a service or mobile agent.

A mobile agent is dispatched a second time if it crashed or reported back to its owner with a failure exception. In the event that the owner is a service a failure exception is signalled to the mobile agent that made the request, provided that the retry failed and no alternative service could be located. If the owner is a mobile agent, the failure exception is forwarded to its parent. The relationship between parent and child is normally asynchronous. However if the parent depends upon the results collected from its child a synchronous relationship is introduced, i.e. the parent must remain stationary until its child has returned. For example, a mobile agent is dispatched to determine a purchase plan for PC system components, e.g. motherboard, CPU etc. The mobile agent spawns a child to determine the best buy for a CPU. Due to hardware dependencies the parent can only consider a motherboard when the child returns.

The next section outlines the failure assumptions for agent server crash failures in information retrieval applications.

2.2 Failure assumptions

The following failure assumptions are used:

- A mobile agent crashes when its current local agent server halts execution, thus terminating all active mobile agents. Such an event is encountered when the host running the agent server platform crashes or a fault is encountered in the agent server process.
- No stable storage mechanism is provided at visited agent servers for the recovery of executing agents.
- Reliable communication links are assumed.
- All agent servers are correct and trustworthy.
- The home agent server is always available.
- At least once failure semantics are assumed, i.e. the agent performs its task at least once. If an agent server crashes the task is repeated at available agent servers. This assumption is applicable to applications where mobile agents only consume information at agent servers.
- A mobile agent ignores crashed agent servers.
- A mobile agent consumes information at agent servers. The state of agent servers is not modified.

3. Exception handler designs

In this research, two possible crash exception handler designs are considered. These are outlined below.

3.1 Mobile timeout design

In this scheme a crash exception handler at the home agent server is associated with a group of independent

```
while !dispatch.isEmpty() { //while agents to send
  handlerTimeOut(t) { // wait t seconds}
  handlerResend(dispatch) { //send replacements }
```

Figure 2 Timeout scheme

mobile agents dispatched to perform an information retrieval task. The handler (figure 2) waits for a timeout period and resends mobile agents that did not return.

The following meta operations and state are provided at the home agent server:

1. *task()*: create dispatch list and send a group of mobile agents to perform an information retrieval task.
2. *dispatch*: list of dispatched mobile agents.
3. *add(A, dispatch)*: add a mobile agent A to the dispatch list.
4. *remove(A, dispatch)*: remove mobile agent A from the dispatch list. A mobile agent is removed from the dispatch list when it returns home.
5. *handler()*: crash exception handler that executes after task operation completed.
6. *handlerTimeOut(t)*: wait for *t* seconds
7. *handlerResend(dispatch)*: resend mobile agents in dispatch list.

The time out exception handler tolerates any number of agent server crash failures with no additional overheads imposed on mobile agents and remote agent servers. However, in the event of an agent server crash all agent servers are revisited. Furthermore it is difficult to select a timeout value in asynchronous systems since there are no established boundaries for processor speed and communication delay. If an aggressive timeout value is used many duplicate agents are dispatched, e.g. a mobile agent may not return within the timeout period if it executes at one or more slow agent servers. If a conservative timeout value is chosen the application blocks until timeout expiry, even when some mobile agents return.

3.2 Mobile shadow design

The mobile shadow scheme employs a pair of replica mobile agents, master and shadow, to survive remote agent server crashes. The *master* is created by its home agent server *H* and is responsible for executing a task *T* at a sequence of hosts described by its itinerary. Initially the master spawns a shadow *shadow_{home}* at its home agent server before it migrates and executes at the first agent server in its itinerary, i.e. *AG_i*. Before the *master* migrates to the next host in the itinerary, i.e. *AG_{i+1}*, it spawns a clone or *shadow_i* and sends a *die* message to *shadow_{home}*. The *shadow_i* repeatedly pings agent server *AG_{i+1}* until it receives a *die* message from its master. The functionality of master and shadow roles is now discussed. Figure 3 illustrates the pseudocode for the mobile shadow scheme.

<pre> master = true; masterLoc=null; shadowLoc=null; alive = true; {// application specific task } execute() {// determines if mobile agent at home agent server } atHome() run() { // mobile agent execution thread } { if master && atHome() { // if master at home } shadow=spawnShadow(this) { // spawn shadow } masterLoc=itin.next() { // get next host } if master { // mobile agent is master } shadowLoc = itin.prev(1) PingThread pinger = new PingThread(shadowLoc, 2, this) pinger.start() { // start pinging shadow } execute() { // execute application task } spawnShadow(this) { // spawn shadow } send(die) { // terminate old shadow } masterLoc=itin.next() { // get next host } else { // mobile agent is shadow } monitorMaster() { // ping master } } } </pre>	<pre> pingNotify() { // callback function for ping thread } { alive=false; if master { // master detects shadow crash } shadow = spawnShadow(this) { //spawn replacement } shadowNotDispatched=false; k = 1 while !shadowNotDispatched try { // send replacement shadow to i-k } shadowLoc = itin.prev(k) shadowNotDispatched = true catch(UnknownHostException) { k++ } PingThread pinger=new PingThread(shadowLoc,2,this) pinger.start() { // start pinging shadow } } monitorMaster(){ // shadow monitors master location } { { // start pinging master } PingThread pinger = new PingThread(MasterLoc,2,this) pinger.start() { // wait while master ok and no die message } while (alive && !receive(die)) if !alive { // if master crash detected spawn shadow } shadow=spawnShadow(this) master = true { // change to master status } masterLoc=itin.next() { // move to next host } } } </pre>
---	--

Figure 3 Mobile shadow scheme pseudocode

Shadow: A shadow terminates when it receives a *die* message from its master. This signifies the master has completed execution at AG_{i+1} and spawned a new clone $shadow_{i+1}$ to monitor agent server AG_{i+2} . However, assume the master is lost due to an agent server crash at AG_{i+1} . For example, AG_{i+1} may crash before the master migrates or during its execution. In this case $shadow_i$ at AG_i detects the crash of its master, spawns a new clone $shadow'_i$ and proceeds to visit agent server AG_{i+2} . Consequently $shadow_i$ is the new master.

Master: A master pings its shadow at AG_{i-1} concurrently with the execution of task t . In the normal case the master completes its execution and spawns a new clone $shadow'$ to monitor the next host, AG_{i+1} . Before the master migrates a *die* message is sent to terminate the shadow at AG_{i-1} . If the master detects a shadow crash it spawns and dispatches a replacement $shadow''$ to the preceding active agent server, i.e. AG_{i-k} . Before the master migrates to the next host in its itinerary it sends a *die* message to terminate the replacement shadow at AG_{i-k} .

A mobile agent has the following meta operations and state:

- *master*: true if the mobile agent is the master.
- *itin*: sequential itinerary pattern.
- *execute()*: the task to execute.
- *atHome()*: return true if at home agent server
- *masterLoc=itin.next()*: migrate to next agent server and return its address. If next agent server is inactive an exception is thrown and mobile agent terminates.

- *shadowLoc=itin.prev(k)*: migrate to k^{th} previous agent server and return its address. If k^{th} agent server is inactive an exception is thrown and mobile agent terminates.
- *PingThread(HostName,t,mob_agent)*: thread that pings host *HostName* every t seconds. The mobile agent *mob_agent* is notified of a crash failure by invoking its *pingNotify()* method
- *shadow=spawnShadow(master)*: create a replica and initialise with master's next location.
- *send(die)*: used by master to terminate its shadow
- *receive(die)*: shadow listens for die message sent by master for termination.

Figure 3 describes the protocol. When a master is created, i.e. *atHome()=true*, it spawns a shadow and migrates to the first host, *masterLoc=itin.next()*.

If the mobile agent is a master and is not at home it creates a thread to ping its shadow. Before the master migrates to the next host it spawns a new shadow and sends a die message, *send(die)*, to terminate the old one.

If the mobile agent is a shadow, i.e. *master=false*, it invokes *monitorMaster()* which creates a ping thread to monitor the master's current agent server. Pinging continues if the master is alive and has not dispatched a die message, i.e. *alive=true* and *!receive(die)*. If a master crash is detected, i.e. *alive=false*, the shadow spawns a new shadow and becomes the new master, i.e. *master = true*. Otherwise the shadow receives a die message and terminates.

If the ping thread detects a crash the *pingNotify()* callback method is invoked and *alive* is set to false. If the mobile agent is a master then its shadow has crashed and a replacement is required, i.e. the master dispatches a shadow to the first active previous agent server, *itin.prev(k)*, and pings its location.

The mobile shadow exception handler offers the advantage that all agent servers are not revisited in the event of a crash since a replica is available at an agent server that precedes the master. However, greater performance overheads are imposed on a mobile agent since a replica must be spawned by the master before it migrates to the next host in its itinerary. Unlike the timeout scheme a limited number of remote agent server crashes are addressed, i.e. no simultaneous crash of master and shadow is assumed.

4. A case study

This research employs a case study application to provide an experimental environment for simulation of agent server crash failures and subsequent analysis of the two exception handler designs. A frequently adopted application domain for mobile agents is within an electronic commerce business supply chain [6][11].

4.1 General requirements

The supply chain case study scenario (figure 4) executes within a local area network, each node hosting an agent server that represents a supplier of computer hardware components. The system enables each supplier to replenish stock using mobile agent technology. A mobile agent is dispatched for each product component to several known suppliers to dynamically determine the best deal with respect to delivery date and price.

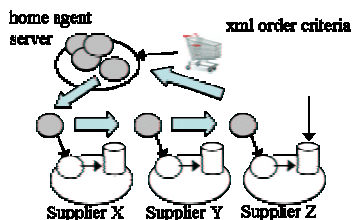


Figure 4 Supply chain architecture

A supplier provides a mobile agent server, capable of hosting mobile agents from other suppliers. It is assumed that each supplier hosts the same agent server platform, e.g. Ajanta [12], that consists of a proxy to a data source (catalogue) enabling mobile agents from other suppliers to query item prices and availability.

The following sections discuss the architecture further looking at the interaction between the mobile agent and the seller for determining the best buy.

4.2 Using mobile agents to obtain estimates

An xml shopping list describes the order criteria for each product. Order criteria include the class of the product (e.g. motherboard), required stock, delivery date, search criteria parameters and a list of suppliers.

The buying subsystem is responsible for parsing the xml file and constructing an object graph of components. The object graph is traversed to create a mobile agent for each component, initialised with an itinerary of suppliers, search criteria and constraints (delivery date and stock). At each supplier's agent server the mobile agent queries the product catalogue to determine the best buy advertised by the supplier that satisfies the given order criteria. Provided the best buy is competitive the mobile agent updates its best buy parameter and visits the next agent server in its itinerary.

4.3 The seller interface

A supplier provides an agent server that hosts a proxy to its product catalogue. The proxy provides methods that a visiting mobile agent invokes to query the catalogue. For example `public Estimate[] QueryCase(String formFactor, int intCapacity, int extCapacity, String style, Order o)` represents a method to query estimates for case units that match a specific form factor, internal bay capacity, external bay capacity and design style.

Each method represents a specific product query and accepts an *Order* object. A matching product is represented as an *Estimate* object. An *Order* object encapsulates the order constraints including total stock and required delivery date. The *Estimate* object encapsulates matching product details including item id, total stock and delivery date.

5. An experiment

The aim of the experiment is to apply the case study to compare the performance of the two crash exception handler designs. The Ajanta [12] mobile agent system is used. Every Ajanta agent server includes a registry to maintain application resources and track its mobile agents. Most of the other agent systems considered e.g., [13][14][15], require application developers to write their own customised resource registry. Resources in such systems may be provided by static agents.

A single mobile agent will visit three suppliers to determine the best buy for fifteen 8GB IDE hard drives. For simplicity, each supplier represents the product catalogue using Mysql 3.23 with JDBC driver 2.0.8. The experiment will be performed on a 10mbps local area network using four 64MB Intel Pentium II 400Mhz (Celeron) PCs running RedHat Linux 7.2 and Ajanta [12].

Two performance measures will be obtained:

- **Normal round trip time:** the time taken for a mobile agent to complete its itinerary and return to the home agent server with the best buy. Agent servers visited by the mobile agent suffer no crash failures.
- **Crash round trip time:** the time taken to complete an itinerary and report back to the home site with the best buy in the presence of one agent server crash.

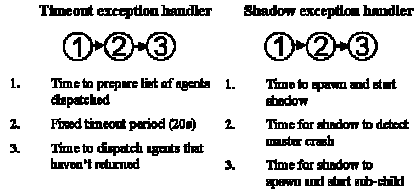


Figure 5 Exception handler performance overheads

Furthermore the performance overheads in figure 5 will be measured for a single agent server crash. Path 1-2-3-1 for the mobile shadow scheme represents handler execution for an agent server crash. Subsequently, path 1-2-1 represents normal execution.

The source code to calculate the performance overheads for the timeout exception handler scheme will impose no additional increase to the normal or crash round trip time. This is because the calculations are determined at the agent server that dispatched the mobile agent. However, this is not true for the mobile shadow exception handler scheme since the mobile agent state must be augmented to log the times for spawning a child and detecting the crash of the agent server where the primary mobile agent is located. Therefore, to provide an accurate comparison there will be two sets of round trip times for the mobile shadow exception handler scheme:

1. Round trip times assume augmented mobile agent state with performance variables
2. Round trip times assume no augmented mobile agent state, i.e. no performance variables.

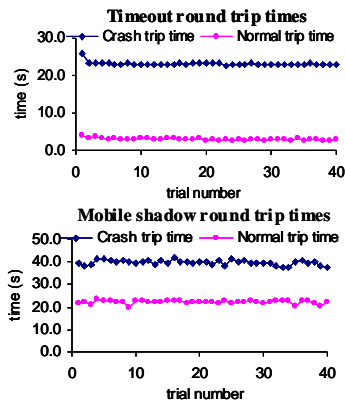


Figure 6 Crash trip time increase

Conservative timeout values were selected for the timeout scheme so that a mobile agent has sufficient time to return before another replica is dispatched. (Due to certain implementation-related considerations, the mobile shadow design for the experiment has been slightly adjusted from the outlined design in Section 3).

6. Results and Analysis

The round trip times are obtained from forty trial runs. Under normal conditions the same mobile agent is dispatched forty times. To simulate an agent server crash a mobile agent waits at a specific agent server to enable manual termination. After recovery the crashed agent server is restarted. This is done due to the enforced security model of Ajanta [12], i.e. there is no mechanism to enable a mobile agent to terminate an agent server or for an agent server to halt when a specific mobile agent arrives.

6.1 Round trip time performance

Figure 6 illustrates the time increase introduced by the crash round trip for each exception handler.

Performance calculations for the mobile shadow scheme impose a minor increase of 0.50% on the round trip times. The mobile shadow round trip times represent the case where the mobile agent state is not augmented with performance calculations.

The mobile time out scheme offers a quicker average normal and crash round trip, i.e. 3s and 23.2s respectively. This is compared to the mobile shadow scheme that provides an average of 22.2s and 39.6s. However, the crash trip time for the timeout scheme depends upon the total agent servers visited. Longer trips need a larger timeout, increasing the crash round trip time. The mobile shadow scheme is independent of trip length and consequently may perform better for longer trips.

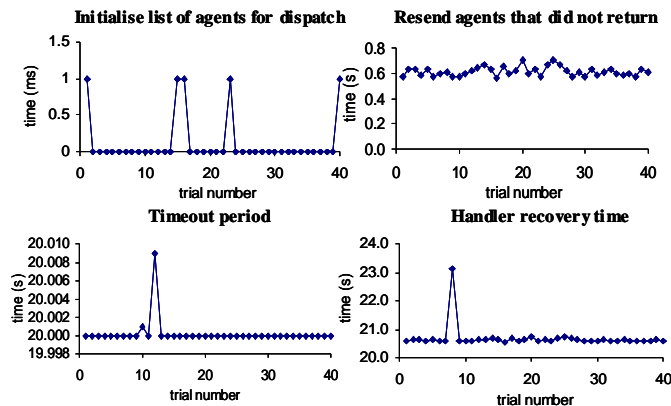


Figure 7 Timeout overheads

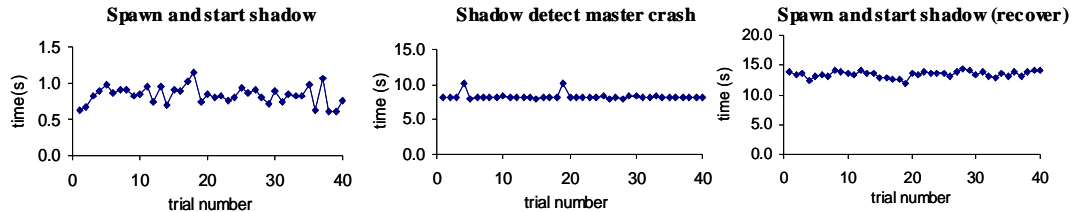


Figure 8 Mobile shadow overheads

6.2 Timeout scheme overheads

Figure 7 illustrates the performance overheads for the timeout exception handler scheme. The timeout exception handler has insignificant times for initialising a record of agents to dispatch (0.1ms) and resending those agents (0.6s) that failed to return. Both measures fall within 1 second. However these measures were obtained from dispatching a single mobile agent.

The major performance overheads for the mobile timeout exception handler are the timeout period (20s) and recovery time (20.7s). The recovery time represents the time to resend agents that have not returned in addition to the timeout period that elapsed when the handler completed

6.3 Mobile shadow scheme overheads

The mobile shadow scheme offers slower round trip times. Figure 8 illustrates the performance overheads. A shadow starts a thread to ping the next agent server where its master will execute. The average time for a shadow to detect its master's agent server crash is 8.3s. The average time to spawn and start a shadow under normal execution is negligible, i.e. 0.8s. A larger overhead (13.4s) is introduced when a shadow spawns and starts its own shadow during recovery. This may be because the shadow migrates while its child pings its destination agent server.

7. Related Work

An alternative approach [2][4][5] to the shadow scheme replicates a mobile agent, at each stage of its itinerary, to a set of agent servers that provide the desired service. If a mobile agent is lost due to an agent server crash a voting algorithm is run by the replicas to elect a new leader. There may be savings in recovery performance since a replica is immediately available at an alternative agent server. However overheads still exist for electing a new leader and sending replicas to redundant agent servers at each stage of the itinerary. It would be interesting to compare the performance of this approach with the mobile shadow scheme.

Further research is underway regarding an exception model for a group of mobile agents. There are two approaches for distributed exception handling.

The atomic action [16] scheme employs a transaction structure to confine errors within a group of participating processes that communicate using transactional shared objects. Concurrent exceptions are resolved into a single exception using a tree hierarchy [17]. The resolved exception is the root of the smallest subtree that contains all raised exceptions. However for some applications the construction of a resolution tree is difficult to provide a sensible resolved exception.

The guardian model [18] employs a global exception handler to encapsulate application specific recovery rules and invoke the correct handler in each participant for an exception occurrence(s). The implementation [18] uses a process group to represent a global exception handler. Each participant has a guardian that signals exceptions to the group and receives exceptions on the behalf of its participant. However it is not apparent what error confinement mechanisms exist for participants to rollback resources that are in an incorrect state.

8. Conclusion

The paper has presented two exception handler designs for handling crash failures of mobile agents. Results and analysis show that although the mobile shadow scheme offers the slower round trip times it introduces the smallest round trip time increase in the presence of agent server crashes. The timeout scheme has the disadvantage that the crash round trip time depends upon the total agent servers visited. A larger timeout is needed for longer trips, increasing crash round trip time.

The mobile shadow scheme uses mobility and replication to provide fault tolerance. Essentially an exception handler migrates with the mobile agent, enabling the scheme to be used for all trip lengths. It is expected that this will provide the foundation for a fault tolerant group of collaborating mobile agents.

Potential future work involves the experiment and exception model. Firstly, a stricter failure model could be enforced e.g., if the mobile agent modifies the state of agent servers, *exactly once* semantics must be enforced. Secondly, the experiment could be extended to deal with orphaned children due to parent server crash failures. Thirdly, dynamic service location may be introduced [19].

This paper has presented an exception model whereby application exceptions are signalled between agent server

and mobile agent. However there is a different model to consider concerning groups of mobile agents [20][21]. For example a mobile agent group may exist to purchase components for a composite product. In this case the mobile agents must be aware of dynamically changing global state such as total budget, latest delivery date, and design constraints. If a mobile agent performs an operation that violates the group integrity, e.g. buying a product that exceeds an agreed budget, participants must be notified. A process group [22][23][24][25] is traditionally used for reliable event communication. Adapting process groups for mobile agents is difficult due to their ability to dynamically relocate.

9. Acknowledgements

The authors wish to thank the Ajanta team at the University of Minnestota for technical support. Simon Pears is funded by an EPSRC doctoral studentship. The work is also funded practically by the DTI/EPSRC e-demand project and the EPSRC IBHIS project.

References

- [1] L.M.Silva, V.Batista and J.G.Silva, "Fault-Tolerant Execution of Mobile Agents," in *Proc. International Conference on Dependable Systems and Networks*, New York, June 2000, pp.144-153.
- [2] M.Strasser, K.Rothermel and C.Maihofer, "Providing Reliable Agents for Electronic Commerce," in *Trends in Distributed Systems for Electronic Commerce (TREC'98)*, LNCS 1402, Springer-Verlag, 1998, pp.241-253.
- [3] F.Schneider, "Towards Fault-Tolerant and Secure Agency," in *Proc. 11th International Workshop on Distributed Algorithms*, Saarbrucken, September 1997, pp.1-14.
- [4] S.Pleisch and A.Schiper, "Modeling Fault-Tolerant Mobile Agents as a Sequence of Agreement Problems," in *Proc. 19th Symposium on Reliable Distributed Systems (SRDS)*, Nuremberg, October 2000, pp.11-20.
- [5] F.M.Silva and R.Popescu-Zeletin, "Mobile Agent-Based Transactions in Open Environments," *IEICE Transactions on Communications*, E83-B(5), pp.973-987, 2000.
- [6] H.Vogler, T.Hunkleemann and M.Moschgath, "An Approach for Mobile Agent Security and Fault Tolerance Using Distributed Transactions," in *Proc. International Conference on Parallel and Distributed Systems (ICPADS'97)*, Seoul, December 1997, pp.268-274.
- [7] Mohindra, A. Purakayastha and P. Tahiti. "Exploiting Non-determinism for Reliability of Mobile Agent Systems," in *Proc. International Conference on Dependable Systems and Networks*, New York, June 2000, pp.144-153.
- [8] T.Anderson and P.A.Lee. "Fault Tolerance Principles and Practice," Prentice Hall, 1981.
- [9] A.Fuggetta, G.P.Picco and G.Vigna, "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, 24(5), pp.342-361, 1998.
- [10] Jie Xu and Brian Randell, "Tutorial: Exception Handling and Software Fault Tolerance," in *Proc. International Conference on Dependable Systems and Networks*, New York, June 2000.
- [11] P. Dasgupta, N. Narasimhan, L. Moser, and P.M. Melliar Smith, "MAGNET: Mobile Agents for Networked Electronic Trading," *IEEE Transactions on Knowledge and Data Engineering*, 24(6), pp.509-525, 1999.
- [12] A.Tripathi and N.Karnik. "Protected Resource Access for Mobile Agent-based Distributed Computing," in *Proc. ICPP workshop on Wireless Networking and Mobile Computing*, Minneapolis, August 1998, pp.144-153.
- [13] M.Oshima, G.Karjoth and K.Ono, "Aglets Specification 1.1 Draft," <http://www.trlibm.co.jp/aglets/spec11.html>, 1998.
- [14] IKV++ Technologies AG, "Grasshopper 2.2 Programmers Guide," <http://213.160.69.23/grasshopper-website>, 2002.
- [15] R.S.Gray, G.Cybenko, D.Kotz, R.A.Peterson, and D.Rus, "D'Agents: Applications and Performance of a Mobile Agent System," *Software Practice and Experience*, 32(6), pp.543-573, 2002.
- [16] J.Xu, A.Romanovsky and B.Randell, "Concurrent Exception Handling and Resolution in Distributed Object Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol.11, pp.1019-1031, October 2000.
- [17] R.H.Cambell and B.Randell, "Error Recovery in Asynchronous Systems," *IEEE Transactions on Software Engineering*, vol.12, pp.811-826, 1986.
- [18] R.Miller and A.Tripathi, "The Guardian Model for Exception Handling in Distributed Systems," in *Proc. 21st IEEE Symposium on Reliable Distributed Systems*, Suita, October 2002, pp.304-313.
- [19] T.Bellwood, L.Clement, D.Ehnebuske, A.Hatley, M.Hondo, Y.L.Husband, K.Januszewski, S.Lee, B.Mckee, J.Munter, and C.Riegen, "UDDI Version 3.0," <http://uddi.org/uddi-v3.00-published-20020719.htm>, 2002.
- [20] V. Nagamuta and M. Endler. "Coordinating Mobile Agents through the Broadcast Channel," *Anais do Simpósio Brasileiro de Redes de Computadores (SBRC 2001)*, Florianopolis, May 2001.
- [21] R.J.A.Macedo and F.M.Silva. "Integrating Mobility into Groups," in *European Research Seminar on Advances in Distributed Systems (ERSADS)*, Bertinoro, May 2001.
- [22] K.P.Birman, "The Process Group Approach to Reliable Distributed Computing," *Communications of the ACM* 36(12), pp.37-53, 1993.
- [23] R.Renesse, K.P.Birman and S.Maffei, "Horus, A Flexible Group Communication System," *Communications of the ACM*, 39(4), pp.76-83, 1996.
- [24] L.E.Moser, P.M.Melliar Smith, D.A.Agarwal, R.K.Budhia and C.A.Lingley-Papadopoulos, "Totem: A Fault Tolerant Multicast Group Communication System," *Communications of the ACM*, 39(4), pp54-63, 1996.
- [25] Y.Amir, D.Dolev, S.Kramer and D.Malki, "Transis: A Communication Subsystem for High Availability," in *Proc. 22nd International Symposium on Fault-Tolerant Computing*, Boston, July 1992, pp.76-84.