Real-Time Systems, 32, 235–252, 2006
© 2006 Springer Science + Business Media, Inc. Manufactured in The Netherlands.
DOI: 10.1007/s11241-005-4682-5

# A Dynamic Shadow Approach to Fault-Tolerant Mobile Agents in an Autonomic Environment

JIE XU
*School of Computing University of Leeds, Leeds, UK, LS2 9JT, UK*

SIMON PEARS
*Department of Computer Science, University of Durham, Durham, UK, DH1 3LE, UK*

**Published online:** 28 December 2005

**Abstract.** Large-scale distributed applications such as online information retrieval and collaboration over computational elements demand an approach to self-managed computing systems with a minimum of human interference. However, large scales and full distribution often lead to poor system dependability and security, and increase the difficulty in managing and controlling redundancy for fault tolerance. In particular, fault tolerance schemes for mobile agents to survive agent server crash failures in an autonomie environment are complex since developers normally have no control over remote agent servers. Some solutions inject a replica into stable storage upon its arrival at an agent server. But in the event of an agent server crash the replica is unavailable until the agent server recovers. In this paper we present a failure model and an exception handling framework for mobile agent systems. An exception handling scheme is developed for mobile agents to survive agent server crash failures. A replica mobile agent operates at the agent server visited prior to its master's current location. If a master crashes its replica is available as a replacement. The proposed scheme is examined in comparison with a simple time-out scheme. Experimental evaluation is performed, and performance results show that the scheme leads to some overhead in the round trip time when fault tolerance measures are exercised. However the scheme offers the advantage that fault tolerance is provided during the mobile agent trip, i.e. in the event of an agent server crash all agent servers are not revisited.

**Keywords:** autonomic computing, exception handling, fault tolerance, mobile agents, performance evaluation, server crash failures

## 1. Introduction

Traditionally, fault tolerance and security policies for large-scale distributed systems have taken a "host-central" view so that the emphasis has been on protecting hosts, and making hosts dependable and fault-tolerant. This has worked very well for traditional client-server models, but is no longer adequate for envisaged approaches for the *next* generation of distributed computing in an autonomic environment. One of the major problems with client-server architectures is that there can be a very high overhead in moving resources round a big distributed system (e.g. data). In contrast, with mobile code, we aim to move the code so that it is local to the associated resources needed. The gain is in terms of performance (locality), late binding, reconfiguration (resource location is not built into the application) and scalability. Specific application examples include mobile e-commerce, online purchasing and delivery, and Web access for retrieval of real-time information such as stock quotes, flight and reservation information, navigational maps, and weather reports (Fuggetta et al., 1998). However, if mobile code is to be used for serious industrial applications, it is imperative that fault tolerance and security architectures are used a priori, otherwise users will not be able to trust the system. Fault tolerance must be provided for both hosts and mobile code. Especially, effective and feasible solutions must be developed to protect mobile code against host crashes and against malicious attacks from hosts.

Suppose that a computer *C* containing a user application, in a large-scale, distributed and autonomic environment, needs remote resources. It therefore despatches executable code to remote hosts to acquire the resources. The resource could be data, and /or it could be side effecting (e.g. 'switch off this valve'). The code could be executed on hardware, or using a software interpreter. The code reports back to *C* in terms of some state information. Thus *mobile code* is defined as executable code which is despatched, which uses remote resources, and which reports back on its results. In other words, code mobility is the capability to dynamically change the bindings between code fragments and the location where they are executed. A special design paradigm of mobile code technology is *mobile agents*; a mobile agent is distinguishable by the properties of autonomy and the capability to migrate the entire execution unit freely to a foreign host. At a remote host, the mobile code could relocate again, or it could spawn further mobile code (thus giving a recursive structure).

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 (Silva et al., 2000). Without replication, a mobile agent is a single point of failure whereby all its internal state is lost in the event of:

- A remote agent server crashing during execution;
- A mobile agent spawning a child and migrating to a remote agent server that crashes. The child will be orphaned and unable to communicate with its parent;
- A mobile agent spawning a child and subsequently migrating to a remote agent server. The child will be lost when its agent server crashes. Subsequently the parent blocks waiting for its child to report back.

An information retrieval mobile agent visits a sequence of remote hosts consuming information that satisfies criteria provided by its user. At each host the mobile agent updates its internal state (e.g. to log the cheapest hard disk drive on the behalf of its user). In the event of an agent server crash all information relating to the current cheapest buy is lost. Consequently there is significant attention for fault tolerance against mobile agent loss at agent servers that fail by crashing (Silva et al., 2000; Strasser et al., 1998; Schneider, 1997; Pleisch and Schiper, 2000; Silva and Popescu-Zeletin, 2000; Vogler et al., 1997; Mohindra et al., 2000). Some solutions (Silva et al., 2000; Strasser et al., 1998; Silva and Popescu-Zeletin, 2000; Vogler et al., 1997) employ transaction processing to satisfy failure dependencies with agent servers, i.e. execution of a mobile agent modifies its internal state and the state of the agent server. However, for information retrieval applications, transaction processing solutions are too costly and introduce unnecessary performance overheads since there are no state dependencies introduced between the mobile agent and remote agent servers. Furthermore, some solutions (Mohindra et al.,

2000) introduce fault tolerance into the agent server platform, e.g. mobile agents are replicated into stable storage upon arrival.

In this paper we propose an exception handling approach that uses mobile shadows (Pears et al., 2003) to maintain mobile agent availability in the presence of agent server crashes. An exception handler design is proposed and analysed for performance using an experimental case study application. Mobile agents filter information, i.e. no state dependencies are introduced with the agent server. Furthermore there is no stable storage at remote agent servers. (Stable storage and transaction processing will be addressed in a future paper.) In Pears et al. (2003) an Ajanta (Tripathi and Karnik, 1998) implementation was used for a single faulty agent server. This paper presents an IBM Aglets (Oshima et al., 1998) implementation for a random agent server crash.

This paper has the following structure. Section 2 introduces exception handling for mobile agents and provides a failure model. Section 3 outlines the exception handler designs and failure assumptions. Section 4 describes the case study. Section 5 introduces the experiment. Section 6 analyses exception handler performance. Finally Section 7 provides a conclusion and outlines future work.

## 2. A Framework for Mobile Agent Fault Tolerance

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 (Fuggetta et al., 1998) is assumed in this paper, i.e. the mobile agent restarts its execution at each host.) A 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. Remote hosts visited by a mobile agent are assumed to execute the same agent server platform.

### *2.1. Exception Handling Model for Mobile Agents*

A mobile agent can be regarded naturally as a software component. Figure 1 illustrates an adaptation of the exception model for software components presented in Xu and Randell (2000) for mobile agents. An agent server $AG_j$ offers a set of services $S = \{s_1, s_2, \ldots, s_n\}$. A service $s_i$ is a software component that a mobile agent manip-

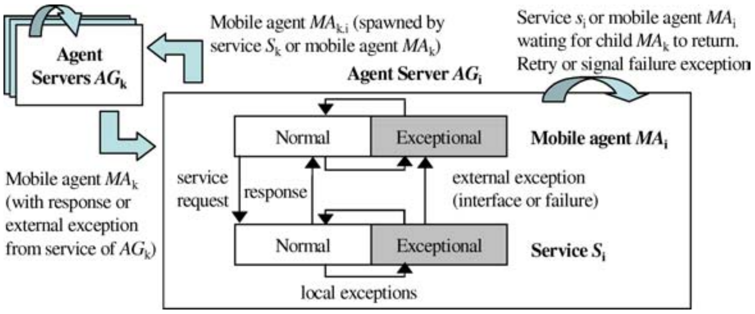

Agent Servers $AG_k$
Mobile agent $MA_{k,i}$ (spawned by service $S_k$ or mobile agent $MA_k$)
Service $s_i$ or mobile agent $MA_i$ wating for child $MA_k$ to return. Retry or signal failure exception
Agent Server $AG_i$
Normal
Exceptional
Mobile agent $MA_i$
Mobile agent $MA_k$ (with response or external exception from service of $AG_k$)
service request
response
external exception (interface or failure)
Normal
Exceptional
Service $S_i$
local exceptions

*Figure 1.* Mobile agent exception handling framework.

ulates by issuing method calls (service requests). A software component (i.e. an agent or a service) defines its own set of internal or local exceptions $I = \{e_1, e_2, \ldots, e_n\}$ and associated handlers $IH = \{h_1, h_2, \ldots, h_n\}$ that serve to provide corrective actions. An internal exception occurrence $e_i$ triggers the exceptional activity $h_i$ within the software component. If the exception is successfully handled normal activity resumes and completes, e.g. a service $s_i$ 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 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 $AG_{i-1}$ 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 service specification.
2. *failure*: the service is unable to provide a suitable response, e.g. a commerce service is unable to meet the delivery deadline for a given order.

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 either 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 a mobile agent $MA_k$ to visit an agent server $AG_k$ in reaction to a request made by $MA_i$. Similarly mobile agent $MA_i$ may spawn child $MA_k$ to perform a delegated task such as information retrieval. Consequently the *owner* of a child is either a service or mobile agent. A mobile agent is dispatched a second time if it crashes or reports a failure exception to its owner. If 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 a 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 returns. For example, assume a mobile agent maybe dispatched to determine a purchase plan for PC system components, e.g. motherboard, CPU etc. The mobile agent dispatches a child to determine the best deal for a CPU. Due to hardware dependencies the parent can only consider a motherboard when its child returns.

### ***2.2. Failure Model for Mobile Agents***

A failure model defines the ways in which failures may occur in order to provide an understanding of the effects of failure (Coulouris et al., 2001). Ideally, a failure model should be defined by conducting field-based observations over a large time period for different systems in operation (Marsden et al., 2002). Information may be collected such

*Table 1.* Mobile agent system failure model.

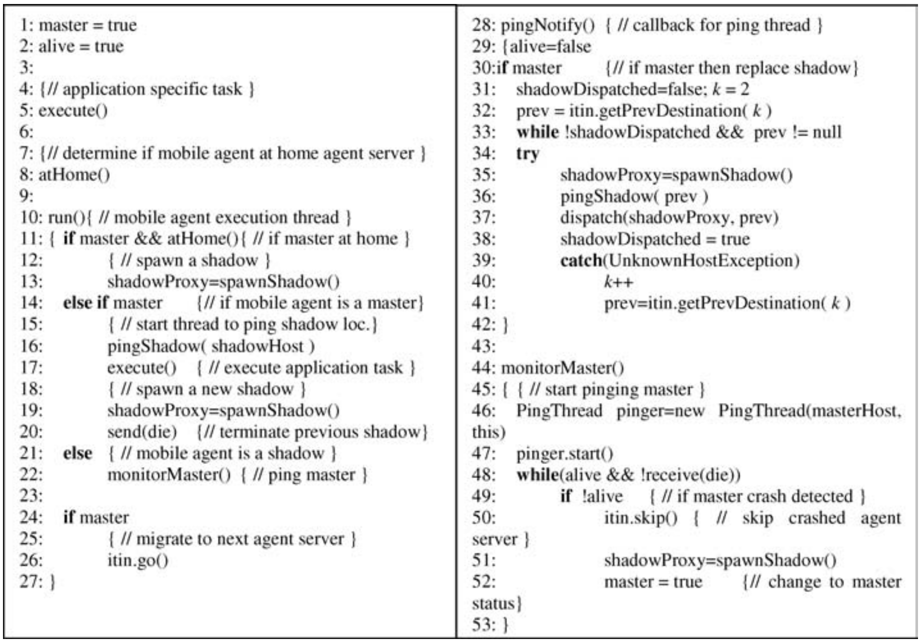
| Class | Description |
|---|---|
| Security | There are two types of malicious attacker: agent or host |
| Malicious agent | Example malicious agent attacks include: unauthorised access to services, denial of service (e.g. recursive cloning), and monitoring agent server and mobile agent activity. |
| Malicious host | Example malicious host attacks include: denial of execution, masquerading as a trusted agent server or mobile agent, state corruption, improper execution of code and modifying system call results. |
| Communication | There are two classes of communication fault: transport and message. |
| Transport | Mobile agent transport failure, e.g. communication link failure or an untrusted agent is refused permission to execute at an agent server. |
| Message | Message failure due to dynamic location, e.g. out of sequence, duplicate or corrupted messages. |
| Software | Software faults exist within a mobile agent or a service at the agent server |
| | Invalid inputs or response values from a software service at agent server. |
| | Denied access to service due to heavy load. |
| | Locking failures, e.g. deadlock, livelock. |
| Crash failures | Mobile agent fails to report back to the home agent server |
| Mobile agent loss | Node crash due to hardware or systems software fault. |
| | Agent server crash due to OS process deletion, system software failure or security attack. All active mobile agents and services at the agent server are lost. |
| | Mobile agent crash due to deadlock, node or agent server crash. |
| Application crash | A mobile agent blocks waiting to communicate with its child (parent) that has failed by crashing. |

as failure classification, frequency and the activities that lead to failure. However, few studies exist for mobile agents. So far there are few concrete failure models for mobile agent systems (Waldo, 2001). To the best of our knowledge the only existing failure model is Tripathi and Milner (2001). Table 1 illustrates a suggested failure model for mobile agent systems based on Tripathi and Milner (2001). This will provide the basis for designing and developing mobile agent fault tolerance schemes.

This paper, in particular, presents a scheme for tolerating mobile agent loss due to agent server crash failure. Consequently the scheme provides the foundation for the revised exception handling framework to operate in the presence of agent server crash failures.

## 3. A Dynamic Mobile Shadow Scheme for Mobile Agent Fault Tolerance

The mobile shadow scheme employs a pair of replica mobile agents, master and shadow, to survive remote agent server crashes. It is assumed that the home agent server is always available. For example, if the home agent server crashes its mobile agents may return to a replica agent server. 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 $shadow_{\text{home}}$ at its home agent server before it migrates and executes



```
master = true
alive = true

{// application specific task }
execute()

{// determine if mobile agent at home agent server }
atHome()

run(){ // mobile agent execution thread }
{ if master && atHome(){ // if master at home }
        { // spawn a shadow }
        shadowProxy=spawnShadow()
  else if master      {// if mobile agent is a master}
        { // start thread to ping shadow loc.}
        pingShadow( shadowHost )
        execute()   { // execute application task }
        { // spawn a new shadow }
        shadowProxy=spawnShadow()
        send(die)   {// terminate previous shadow}
  else  { // mobile agent is a shadow }
        monitorMaster()  { // ping master }

  if master
        { // migrate to next agent server }
        itin.go()
}
```

```
pingNotify()  { // callback for ping thread }
{alive=false
if master        {// if master then replace shadow}
  shadowDispatched=false; k = 2
  prev = itin.getPrevDestination( k )
  while !shadowDispatched &&  prev != null
  try
        shadowProxy=spawnShadow()
        pingShadow( prev )
        dispatch(shadowProxy, prev)
        shadowDispatched = true
        catch(UnknownHostException)
              k++
              prev=itin.getPrevDestination( k )
}

monitorMaster()
{ { // start pinging master }
  PingThread pinger=new PingThread(masterHost, this)
  pinger.start()
  while(alive && !receive(die))
        if !alive    { // if master crash detected }
              itin.skip()  { // skip crashed agent server }
              shadowProxy=spawnShadow()
              master = true     {// change to master status}
}
```

*Figure 2.* Mobile shadow scheme pseudo code.

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 terminate $shadow_{\text{home}}$. The $shadow_i$ repeatedly pings agent server $AG_{i+1}$ until it receives a *die* message from its master. The functionality of shadow and master roles (Figure 2) is now discussed with respect to exception handling.

**Shadow**: A shadow is a clone of the master that acts as an exception handler for a master crash. The shadow pings its master's agent server. If a shadow detects a master crash it raises a local exception to signify master failure. The exception handler skips the master's current location and migrates the shadow to the next agent server.

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 $AG_{i+1}$. However, assume the master is lost due to an agent server crash at $AG_{i+1}$, e.g. $AG_{i+1}$ could crash before the master migrates or during execution. In this case $shadow_i$ at $AG_i$ detects the master crash, spawns a new clone $shadow'_i$ and visits the next agent server $AG_{i+2}$. The new master is now $shadow_i$.

**Master**: A master pings its shadow's agent server $AG_{i-1}$ 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 in the itinerary $AG_{i+1}$. 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 raises a local exception to signify the failure of its shadow. The master's exception handler then spawns and dispatches a replacement *shadow"* to the next 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 $AG_{i-k}$. It is assumed that an itinerary and mobile agent have the following meta operations and state:

### Itinerary

- *_destinations*: queue of agent servers to visit.
- *_visited*: stack of visited agent servers.
- *go()*: remove next agent server in the sequence from *_destinations* queue and push onto *_visited* stack. Dispatch mobile agent to the next agent server.
- *skip()*: skip next agent server by removing its address from the head of *_destinations* queue.
- *loc = itin.getPrevDestination(k)*: get the address of the *k*th previous agent server visited.

### Mobile Shadow Design

- *itin*: itinerary instance.
- *master*: true when the mobile agent is a master.
- *alive*: false when a mobile agent is notified that its shadow or master has crashed.
- *dieProxy*: proxy reference to shadow that the master will terminate before it is dispatched to next agent server.
- *shadowProxy*: proxy reference to master's shadow.
- *masterHost*: address of master agent server that shadow pings.
- *shadowHost*: address of shadow agent server that master pings.
- *shadowProxy = spawnShadow()*: spawn a new replica and return its reference. The *dieProxy* is updated to reference the master's previous shadow and master = *false* in the replica. If the agent is a shadow that has detected its master has crashed then *masterHost* is set to the next available host in the itinerary. If the agent is a master then *masterHost* is the next host visited in the itinerary. *shadowHost* is always the address of the current agent server.
- *PingThread(HostName, proxy)*: thread that pings host *HostName*. The mobile agent *proxy* is notified of a crash by sending it a *pingNotify* message.

- *dispatch(proxy, HostName)*: dispatch mobile agent *proxy* to agent server *HostName*.
- *send(die)*: message master sends to terminate shadow.
- *receive(die)*: shadow listens for die message sent by its master for termination.
- *receive(pingNotify)*: notification of a master or shadow crash.
- *execute()*: start execution at the current agent server.
- *atHome()*: return true if at home agent server.

Figure 2 describes the protocol. When the master starts at its home agent server (line 11), i.e. *atHome()* = *true*, it spawns a shadow (line 13) and migrates to the first host in the itinerary (line 26). If the mobile agent is a master and is at a remote agent server (line 14) it creates a thread to ping its shadow (line 16). Before the master migrates to the next agent server it spawns a new shadow (line 19) and sends a die message (line 20) to terminate the old one.

However, if the mobile agent is a shadow, i.e. *master* = *false*, it invokes *monitorMaster()* (lines 44–53) 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 the ping thread detects a crash the *pingNotify()* callback method is invoked (lines 28–42) and the alive flag is set to false to trigger exception handling activity. If the mobile agent is a master then the shadow exception handler is activated (lines 31–41) to spawn a replacement shadow at the first active previous agent server, *itin.getPrevDestination(k)*. The master can then ping the location of the new shadow (line 36). Alternatively if the mobile agent is a shadow then master exception handling activity is activated (lines 49–52). The master exception handler spawns a new shadow and initialises the shadow to become the new master, i.e. *master* = *true*.

The mobile shadow exception handler offers the advantage that all agent servers are not revisited in the event of a crash failure since a replica is available at an agent server that precedes the master. Consequently there is less information loss. 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. Furthermore a limited number of remote agent server crashes are addressed.

In this research the following assumptions are made to tolerate loss of mobile agents from agent server crashes:

- Reliable communication links are assumed.
- All agent servers are correct and trustworthy.
- A mobile agent crashes when its current local agent server halts execution due to a host crash or fault in the agent server process.

- No stable storage mechanism is provided at visited agent servers for the recovery of executing agents.
- At least once failure semantics are assumed whereby the agent performs its designated task at least once. If an agent server crashes it is possible to repeat the task at agent servers ignoring those that crashed.
- A mobile agent ignores crashed agent servers.
- A mobile agent visits agent servers to consume information, i.e. agent server state is not modified.
- There are no simultaneous crashes of the agent servers where the master and shadow are operating.

### *3.1. An Aternative Design: Timeout*

We now consider a timeout-based scheme for the purpose of comparison and evaluation. In this scheme a crash exception handler at the home agent server is associated with a group of independent mobile agents dispatched to perform an information retrieval task. The handler (Figure 3) 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.

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

*Figure 3*. Timeout scheme pseudo code.

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.

## 4. A Case Study Application

This research employs a case study application using IBM Aglets (Oshima et al., 1998) to provide an experimental environment for the simulation of agent server crash failures and subsequent analysis of the exception handler design. A frequently adopted application domain for mobile agents is within an electronic commerce business supply chain.

### *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.

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 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.

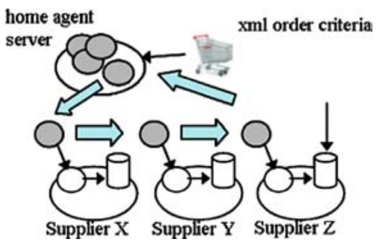


*Figure 4*. Supply chain architecture.

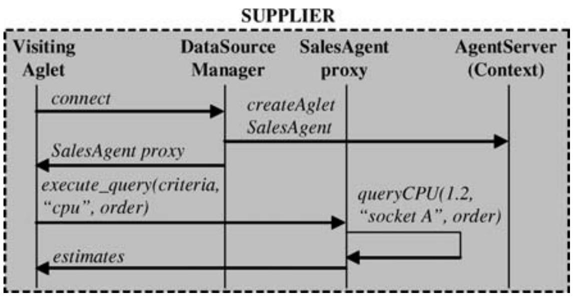


*Figure 5*. Interaction sequence for querying supplier product catalogue.

## *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 Supplier Interface*

Every supplier hosts an agent server to provide a static *DataSourceManager* agent that visiting mobile agents interact with to retrieve a proxy to the product catalogue (Figure 5). A *DataSourceManager* agent listens for *connect* messages and responds by spawning a *SalesAgent* and returning its proxy to the sender. A *SalesAgent* is a static agent that provides mobile agents with an interface to the supplier product. A mobile agent queries the product catalogue by sending an *execute_query* message to the *SalesAgent* proxy with the following parameters:

- **product class**: the class of product, e.g. hard drive.
- **product criteria**: search criteria, e.g. a CPU has criteria regarding speed and hardware interface.
- **order**: order constraints, i.e. stock and delivery date.

In response to an *execute_query* message the *SalesAgent* returns a list of matching items that satisfy both the product and order criteria. For a given business domain the *SalesAgent* must implement an interface for product queries. For example "*public Estimate[] Query Case(String form Factor, 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. The appropriate method is invoked in the *SalesAgent* depending upon the product class provided in the *execute_query* message. Figure 5 illustrates a simple example for a 1.2 Ghz socket A CPU.

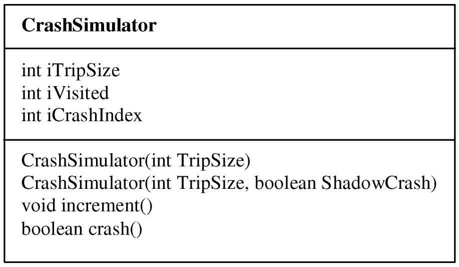
## 5. Experimental Evaluation

The aim of the experimental evaluation is to apply the case study to analyse the performance of the crash exception handler design in the presence of a single agent server crash.

A single mobile agent will visit three suppliers to determine the best buy for fifteen 8 GB IDE hard drives. For simplicity, each supplier represents its product catalogue using Mysql 3.23 with JDBC driver 2.0.8. The experiment is performed on a 10 Mbps LAN using four 64 MB Intel Pentium II 400 MHz (Celeron) PC's running RedHat Linux 7.2 and IBM Aglets (Oshima et al., 1998).

### *5.1. Simulating a Random Crash*

To simulate a crash the mobile agent terminates an agent server using the Java command *System.exit(1)*. Permission to terminate the Java virtual machine is assigned in the security policy file for each supplier agent server. The *CrashSimulator* class (Figure 6) is initialised with the total number of suppliers to visit (*iTripSize*). A random number is generated (*iCrashIndex*) to represent the *n*th visited supplier that will crash, i.e. *0 < iCrashIndex <= iTripSize*. Before a mobile agent migrates to the next agent server in its itinerary it increments the total number of hosts visited (*iVisited*), i.e. *crash.increment()*. When execution begins at the next agent server the mobile agent determines if it is at the agent server selected to crash (*iVisited = = iCrashIndex*). This is done by invoking *CrashSimulator.crash()*. If so, the mobile agent terminates the agent server.



```
CrashSimulator

int iTripSize
int iVisited
int iCrashIndex

CrashSimulator(int TripSize)
CrashSimulator(int TripSize, boolean ShadowCrash)
void increment()
boolean crash()
```

*Figure 6.* CrashSimulator class.

The crash simulation outlined above is applicable to a master crash. To simulate a shadow crash the master delegates the responsibility of terminating the agent server to its shadow. Furthermore a random number must be generated in the range 0 < iCrashlndex < = iTripSize-1 since it is assumed that when a master returns back to the home agent server it discards its shadow. Consequently the crash simulator class provides a second constructor *CrashSimulator(TripSize, ShadowCrash).*

### 5.2. *Performance measurements*

Two performance measures are obtained to gain an insight into the overheads introduced:

- **Normal round trip time:** time taken to complete an itinerary and return to home agent server with the best buy. Visited agent servers suffer no crash failures.
- **Crash round trip time:** time taken to complete an itinerary and return to home agent server with the best buy in the presence of one agent server crash.

Furthermore, the performance overheads in Figure 7 are measured in the event of a single agent server crash.

To simulate a random agent server crash the mobile agent state is augmented with an instance of the *CrashSimulator* class in addition to performance measurements. To provide accurate normal and crash round trip times there are two sets of normal round trip times:

1. Normal round trip time assumes an augmented mobile agent state that includes an instance of the *CrashSimulator* class and performance variables.
2. Normal round trip time assumes no augmented mobile agent state.

## 6. Results and Analysis

The normal and crash round trip times are obtained from forty trial runs. Under normal conditions, the same mobile agent is dispatched forty times. When a crash is simulated, the agent server is reset before the next trial.

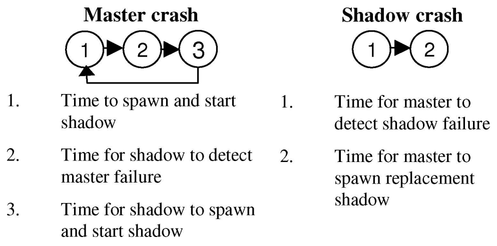

Master crash
1
2
3
1. Time to spawn and start shadow
2. Time for shadow to detect master failure
3. Time for shadow to spawn and start shadow
Shadow crash
1
2
1. Time for master to detect shadow failure
2. Time for master to spawn replacement shadow

*Figure 7.* Performance measurements.

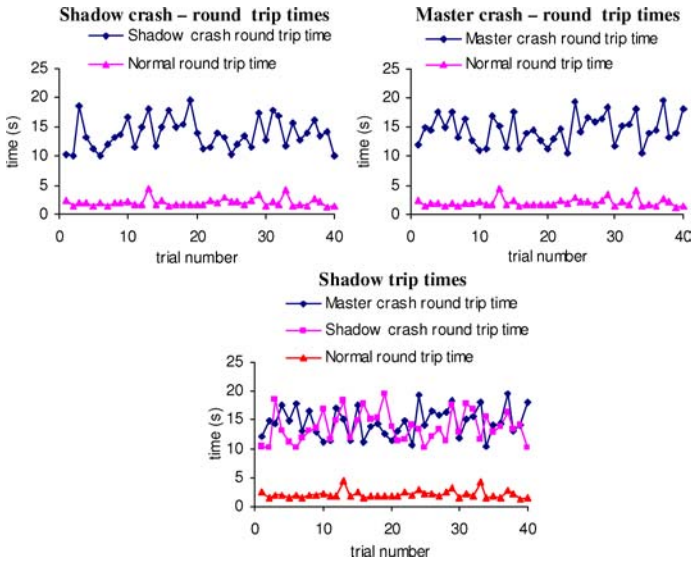


*Figure 8*. Mobile shadow trip times.

### *6.1. Round Trip Time Performance*

A comparison of crash round trip times for the crash of a master and a shadow are illustrated in Figure 8. Performance calculations for the mobile shadow scheme impose a minor increase of 0.50% on the round trip times. In the results that follow, the normal round trip time reflects no state augmentation with performance calculations or the *CrashSimulator* class.

The mobile shadow scheme provides an average normal round trip time of 2.1 s. When fault tolerance measures are exercised in the presence of a single agent server crash the round trip time significantly increases. A shadow crash offers a quicker round trip time of 13.8 s (11.7 s increase) compared to 14.6 s (12.5 s increase) for a master crash. A shadow crash performs quicker since the shadow terminates its agent server while its master is in transit. Consequently the ping thread detects the crash early before the mobile agent begins application execution. The following sections analyse the performance overheads for a master and shadow crash.

### *6.2. Exception Handler Overheads*

Figure 9 illustrates the performance overheads for both a master and shadow crash. The time taken to spawn and start a shadow is negligible for both a master and shadow crash. For example, when handling a master crash, the average time to spawn a shadow or a sub-shadow is 159.4 ms (0.16 s) and 76.5 ms (0.08 s) respectively.

The largest overhead is the time taken for a shadow to detect its parent's agent server crash (4.4 s). This is explained by the concurrent execution of the shadow and its ping thread. Every shadow starts a thread to ping its master's agent server. The ping thread

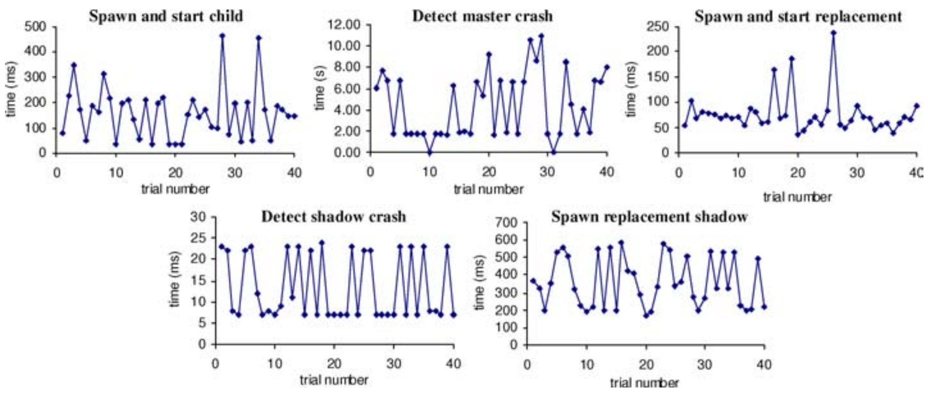


*Figure 9*. Performance overheads of mobile shadow exception handler.

pings until it detects a crash and notifies the blocked shadow. Similarly the master pings its shadow concurrently.

The mobile shadow exception handler demonstrates a significant increase to the round trip time when fault tolerant measures are activated in the event of an agent server crash. A shadow crash marginally offers the quicker round trip time with a saving of 0.4 s.

With respect to the exception handling model presented in Section 2.1, the mobile shadow scheme provides a fault tolerant service for maintaining mobile agent availability in the presence of agent server crashes. This service is embedded within the mobile agent. Provided that developers use a mobile shadow agent, the exception handler is invoked during the mobile agent trip.

So far the mobile shadow exception handler scheme is recursive for child mobile agents, provided there is no synchronous relationship between the child and parent. Furthermore the shadow does not notify the home agent server in the event of a master crash, i.e. the shadow skips the crashed agent server and continues at the next host in the itinerary. In the worst case scenario a mobile agent may visit none of the hosts in its itinerary if all have crashed. One solution is to use an itinerary pattern (Tripathi and Milner, 2001) whereby the mobile agent logs the success for each itinerary entry. The home agent server can take appropriate action when the mobile agent returns, e.g. a mobile agent may be dispatched to all failed agent servers.

The guardian exception handling model presented in Tripathi and Milner (2001) provides a *guardian* at the home agent server that encapsulates recovery behaviour for exceptions that cannot be handled by mobile agents. The guardian may also be used to co-ordinate recovery of mobile agent groups. In the exception handling model presented in Section 2, the guardian may encapsulate the recovery behaviour at the home agent server when a mobile agent fails to retry or locate an alternative software service.

### *6.3. Timing Issues*

Figure 10 indicates the crash trip time for both mobile shadow and timeout schemes. The mobile time out scheme offers a quicker average normal and crash round trip, i.e. 3 s and 23.2 s respectively. This is compared to the mobile shadow scheme that provides an average of 22.2 s and 39.6 s. However, the crash trip time for the timeout

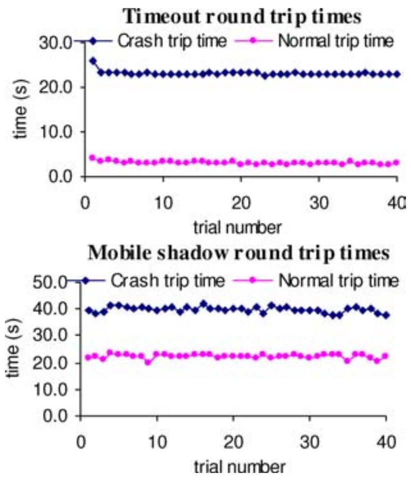


*Figure 10*. Crash trip time increase.

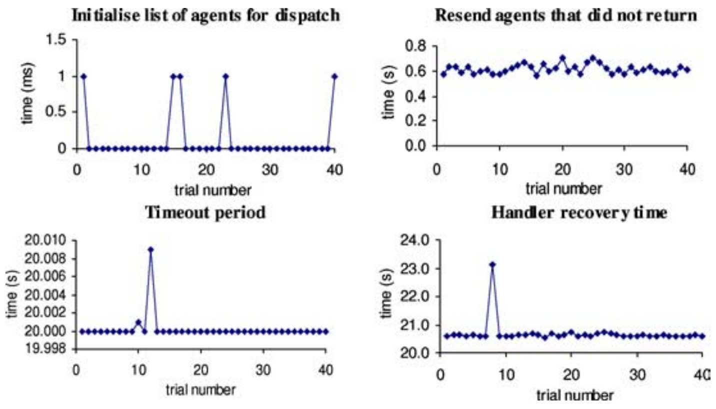


*Figure 11*. Timeout overheads.

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 11 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 (0.1 ms) and resending those agents (0.6 s) 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 (20 s) and recovery time (20.7 s). 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.

To summarise, the mobile shadow scheme provides the foundation for using the exception handling framework in information retrieval environments where mobile agent loss must be tolerated. The scheme offers two advantages. Firstly, mobility and replication provide fault tolerance during the mobile agent trip, i.e. an exception handler that is independent of trip length migrates with the mobile agent. This is compared to a timeout scheme (Pears et al., 2003) where the application developer must vary the timeout interval for different trip lengths. Secondly, the scheme is useful for groups of collaborative information retrieval mobile agents since the master and shadow comprise a single fault tolerant group member. Alternative schemes exist (e.g., Strasser et al., 1998; Pleisch and Schiper, 2000; Silva and Popescu-Zeletin, 2000), that replicate a mobile agent at each stage of its itinerary to an anticipated set of agent servers. However this may introduce a complex design for collaborative information retrieval mobile agents, i.e. for one mobile agent group member a group of replicas must exist at alternative agent servers for each itinerary stage.

## 7. Conclusions and Future Work

This paper has presented an exception handling scheme for tolerating crash failures of mobile agents in an autonomic environment that satisfies the failure assumptions outlined in Section 3. Results and analysis show that the scheme leads to an increase in the round trip time when fault tolerance measures are exercised. However the scheme offers the advantage that fault tolerance is provided during the mobile agent trip, i.e. in the event of an agent server crash all agent servers are not revisited. Furthermore, it is believed that the scheme is simple to employ for collaborative groups of information retrieval mobile agents.

Potential areas of future work involve the experiment and exception model. Firstly, a stricter failure model could be introduced to enforce *exactly once* semantics. Secondly, a simulation system may be built to measure the cost of schemes in different failure environments (Park et al., 2002). Finally the exception model could be extended for mobile agent groups (Nagamuta and Endler, 2002; Macedo and Silva, 2001).

Further research is underway regarding an exception model for groups of mobile agents. Process groups, e.g. (Renesse et al., 1996; Moser et al., 1996), are traditionally used for reliable event communication. The process group abstraction may be considered for communicating exception events between a group of mobile agents. However, adapting process groups for mobile agents is difficult since the location of mobile agents dynamically changes.

## References

Coulouris, G., Dollimore, J., and Kindberg, T. 2001. *Distributed Systems Concepts and Design*, 3rd edition, Addison Wesley.

Fuggetta, A., Picco, G. P., and Vigna, G. 1998. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361.

Macedo, R. J. A., and Silva, F. M. 2001. Integrating mobility into groups. In *European Research Seminar on Advances in Distributed Systems (ERSADS)*, Bertinoro.

Marsden, E., Fabre, J., and Arlat, J. 2002. Dependability of CORBA systems: Service characterization by fault injection. In *Proc. 21st IEEE Symposium on Reliable Distributed Systems*, Suita, pp. 276–285.

Mohindra, A., Purakayastha, A., and Tahiti, P. 2000. Exploiting non-determinism for reliability of mobile agent systems, In *Proc. International Conference on Dependable Systems and Networks*, New York, pp. 144–153.

Moser, L. E., Melliar Smith, P. M., Agarwal, D. A., Budhia, R. K., and Lingley-Papadopoulos, C. A. 1996. Totem: A fault tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63.

Nagamuta, V., and Endler, M. 2001. Coordinating mobile agents through the broadcast channel. *Anais do Simp$osio Brasileiro de Redes de Computadores (SBRC 2001)*, Florianopolis.

Oshima, M., Karjoth, G., and Ono, K. 1998. Aglets specification 1.1 draft, http://www.trl.ibm.co.jp/aglets/specll.httml.

Park, T., Byun, I., Kim, H., and Yeom, H. 2002. The performance of checkpointing and replication schemes for fault tolerant mobile agent systems. In *Proc. 21st IEEE Symposium on Reliable Distributed Systems*, Suita, pp. 256–261.

Pears, S., Xu J., and Boldyreff, C. 2003. Mobile agent fault tolerance for information retrieval applications: An exception handling approach. In *Proc. 6th International Symposium on Autonomous Decentralized Systems*, Pisa.

Pleisch, S., and Schiper, A. 2000. Modeling fault-tolerant mobile agents as a sequence of agreement problems. In *Proc. 19th Symposium on Reliable Distributed Systems (SRDS)*, Nuremberg, pp. 11–20.

Renesse, R., Birman, K. P., and Maffeis, S. 1996. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83.

Schneider, F. 1997. Towards fault-tolerant and secure agentry. In *Proc. 11th International Workshop on Distributed Algorithms*, Saarbrucken, pp. 1–14.

Silva, L. M., Batista, V., and Silva, J. G. 2000. Fault-tolerant execution of mobile agents. In *Proc. International Conference on Dependable Systems and Networks*, New York, pp. 144–153.

Silva, F. M., and Popescu-Zeletin, R. 2000. Mobile agent-based transactions in open environments. *IEICE Transactions on Communications*, Vol. E83-B, No. 5, pp. 973–987.

Strasser, M., Rothermel, K., and Maihofer, C. 1998. Providing reliable agents for electronic commerce. In *Trends in Distributed Systems for Electronic Commerce (TREC'98)*, LNCS 1402, Springer-Verlag, pp. 241–253.

Tripathi, A., and Karnik, N. 1998. Protected resource access for mobile agent-based distributed computing. In *Proc. ICPP workshop on Wireless Networking and Mobile Computing*, Minneapolis, pp. 144–153.

Tripathi, A., and Milner, R. 2001. Exception handling in agent-oriented systems. In *Advances in Exception Handling Techniques*, LNCS-2022, Springer-Verlag, pp. 128–146.

Vogler, H., Hunklemann, T., and Moschgath, M. 1997. An approach for mobile agent security and fault tolerance using distributed transactions. In *Proc. International Conference on Parallel and Distributed Systems (ICPADS'97)*, Seoul, pp. 268–274.

Waldo, J. 2001. Mobile code, distributed computing and agents. *IEEE Intelligent Systems*, 16(2):10–12.

Xu, J., and Randell, B. 2000. Tutorial: Exception handling and software fault tolerance. In *Proc. International Conference on Dependable Systems and Networks*, New York.