A Dynamic Shadow Approach for Mobile Agents
to Survive Crash Failures

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

Abstract

Fault tolerance schemes for mobile agents to survive agent server crash failures are complex since developers normally have no control over remote agent servers. Some solutions modify the agent server platform, e.g. a replica of the mobile agent is injected into stable storage upon its arrival at an agent server. However in the event of an agent server crash the replica remains unavailable until the agent server recovers. This paper presents a failure model and a revised exception handling framework for mobile agent systems. An exception handler design is presented 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. Experimental evaluation is performed and performance results are used to suggest some useful design guidelines.

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]. 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 crashes during execution;
- A mobile agent spawns a child and then migrates to a remote agent server that crashes. Subsequently the child becomes an orphan and is unable to communicate with its parent;
- A mobile agent spawns a child and subsequently migrates to a remote agent server. The child is 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 may dynamically update its internal state. For example a mobile agent may migrate to an itinerary of hard disk drive suppliers, dynamically updating its state to log the cheapest price 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 [1][2][3][4][5][6][7]. Some solutions [1][2][5][6] 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 introduce unnecessary performance overheads since there are no state dependencies introduced between the mobile agent and remote agent servers. Furthermore some solutions [7] introduce fault tolerance into the agent server platform, e.g. mobile agents are replicated into stable storage upon arrival. In the event of an agent server crash the replica is unavailable for an unknown time period.

This paper proposes an exception handling approach that uses mobile shadows [8] 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. It is assumed that mobile agents filter information, i.e. no state dependencies are introduced with the agent server. Furthermore no stable storage is provided at remote agent servers. (Stable storage and transaction processing will be addressed in a future paper.) In [8] an Ajanta [9] implementation was presented and performance results were obtained for one faulty agent server. This paper presents an IBM Aglets [10] implementation that simulates a random agent server crash.

The paper is structured as follows. Section 2 introduces exception handling for mobile agents and provides a failure model. Section 3 outlines the exception handler design and failure assumptions. Section 4 describes the case study and section 5 introduces the experiment. Section 6 analyses exception handler performance. Finally section 7 provides a conclusion and outlines future work.
2. Mobile agent fault tolerance framework

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 [11] is assumed, 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

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 [12] for mobile agents. An agent server AG$_i$ offers a set of services $S = \{s_1, s_2, ..., s_n\}$. A service $s_i$ is a software component that a mobile agent manipulates 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 $I = \{e_1, e_2, ..., e_n\}$ and associated handlers $IH = \{h_1, h_2, ..., h_n\}$ that serve to provide corrective action. 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 $E = \{\text{interface, failure}\}$ 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 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 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 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 has returned. For example, assume a mobile agent is 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

A failure model defines the ways in which failures may occur in order to provide an understanding of the effects of failure [13]. Only then can recovery prove effective. A failure model is ideally defined by conducting field-based observations over a large time period for different systems in operation [14]. Information may be collected such as failure classification, frequency and the activities that lead
to failure. However, it is believed that few studies exist for mobile agents. So far there are few concrete failure models for mobile agent systems [15]. To the best of our knowledge the only existing failure model is [16]. Table 1 illustrates a suggested failure model and failure modes for mobile agent systems.

This paper 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. The dynamic mobile shadow scheme

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 to its mobile agents. 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 $\text{shadow}_{\text{home}}$ 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 $\text{shadow}_1$ and sends a die message to terminate $\text{shadow}_{\text{home}}$. The shadow, 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.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>There are two types of attacker: malicious agent or host.</td>
</tr>
<tr>
<td>Malicious agent</td>
<td>Example malicious agent attacks include: unauthorised access to services, denial of service (e.g. recursive cloning), and monitoring agent server and mobile agent activity.</td>
</tr>
<tr>
<td>Malicious agent server</td>
<td>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.</td>
</tr>
<tr>
<td>Communication</td>
<td>There are two classes of communication fault: transport and message</td>
</tr>
<tr>
<td>Transport</td>
<td>Mobile agent transport failure, e.g. communication link failure or an untrusted agent is refused permission to execute at an agent server.</td>
</tr>
<tr>
<td>Message</td>
<td>Message failure due to dynamic location, e.g. out of sequence, duplicate or corrupted messages.</td>
</tr>
<tr>
<td>Software</td>
<td>Software faults exist within a mobile agent or a service at the agent server</td>
</tr>
<tr>
<td></td>
<td>Invalid inputs or response values from a software service at agent server.</td>
</tr>
<tr>
<td></td>
<td>Denied access to service due to heavy load.</td>
</tr>
<tr>
<td></td>
<td>Locking failures, e.g. deadlock, livelock.</td>
</tr>
<tr>
<td>Crash failures</td>
<td>Mobile agent fails to report back to the home agent server</td>
</tr>
<tr>
<td>Mobile agent loss</td>
<td>Node crash due to hardware or systems software fault.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Mobile agent crash due to deadlock, node or agent server crash.</td>
</tr>
<tr>
<td>Application crash</td>
<td>A mobile agent blocks waiting to communicate with its child (parent) that has failed by crashing.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td>There are two types of attacker: malicious agent or host.</td>
</tr>
<tr>
<td>Malicious agent</td>
<td>Example malicious agent attacks include: unauthorised access to services, denial of service (e.g. recursive cloning), and monitoring agent server and mobile agent activity.</td>
</tr>
<tr>
<td>Malicious agent server</td>
<td>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.</td>
</tr>
<tr>
<td>Communication</td>
<td>There are two classes of communication fault: transport and message</td>
</tr>
<tr>
<td>Transport</td>
<td>Mobile agent transport failure, e.g. communication link failure or an untrusted agent is refused permission to execute at an agent server.</td>
</tr>
<tr>
<td>Message</td>
<td>Message failure due to dynamic location, e.g. out of sequence, duplicate or corrupted messages.</td>
</tr>
<tr>
<td>Software</td>
<td>Software faults exist within a mobile agent or a service at the agent server</td>
</tr>
<tr>
<td></td>
<td>Invalid inputs or response values from a software service at agent server.</td>
</tr>
<tr>
<td></td>
<td>Denied access to service due to heavy load.</td>
</tr>
<tr>
<td></td>
<td>Locking failures, e.g. deadlock, livelock.</td>
</tr>
<tr>
<td>Crash failures</td>
<td>Mobile agent fails to report back to the home agent server</td>
</tr>
<tr>
<td>Mobile agent loss</td>
<td>Node crash due to hardware or systems software fault.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Mobile agent crash due to deadlock, node or agent server crash.</td>
</tr>
<tr>
<td>Application crash</td>
<td>A mobile agent blocks waiting to communicate with its child (parent) that has failed by crashing.</td>
</tr>
</tbody>
</table>

**Table 1 Mobile agent system failure model**

**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 $\text{shadow}_{i+1}$ 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, $AG_{i+1}$ could crash before the master migrates or during execution. In this case $\text{shadow}_i$ at $AG_i$ detects the crash of its master, spawns a new clone $\text{shadow}',$ and proceeds to visit agent server $AG_{i+2}$. Consequently $\text{shadow}_i$ is the new master.

**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 $\text{shadow}'$ 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 $\text{shadow}''$ 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:** An itinerary encapsulates a queue, _destinations_, of agent servers to visit and a stack, _visited_, of agent servers visited by the mobile agent.
1: master = true;
2: alive = true
3:
4: { // application specific task }
5: execute()
6:
7: { // determine if mobile agent at home agent server }
8: atHome()
9:
10: run(){ // mobile agent execution thread }
11: { if master & & atHome(){ // if master at home }
12:     [ // spawn a shadow ]
13:     shadowProxy=spawnShadow()
14:     else if master { // if mobile agent is a master}
15:     { // start thread to ping shadow loc.}
16:     pingShadow( shadowHost )
17:     execute() { // execute application task }
18:     [ // spawn a new shadow ]
19:     shadowProxy=spawnShadow()
20:     send(die) { // terminate previous shadow}
21:     else { // mobile agent is a shadow }
22:     monitorMaster() { // ping master }
23:     }
24:     if master
25:     { // migrate to next agent server }
26:     itin.go()
27: }
28: pingNotify() { // callback for ping thread }
29: { alive=false;
30:     if master { // if master then replace shadow}
31:     shadowDispatched=false; k = 2
32:     prev = itin.getPrevDestination( k )
33:     while !shadowDispatched & & prev != null
34:         try
35:             shadowProxy=spawnShadow()
36:             pingShadow( prev )
37:             dispatch(shadowProxy, prev)
38:             shadowDispatched = true
39:         catch(UnknownHostException)
40:             k++
41:             prev=itin.getPrevDestination( k )
42: }
43: }
44: monitorMaster()
45: { { // start pinging master }
46:     PingThread pinger=new PingThread(masterHost, this)
47:     pinger.start()
48:     while(alive & & !receive(die))
49:         if !alive { // if master crash detected }
50:         itin.skip(){ // if skip crashed agent server }
51:         shadowProxy=spawnShadow()
52:         master = true { // change to master status}
53: }

Figure 2 Mobile shadow scheme pseudocode

- **go():** remove next agent server from the _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 kth previous agent server visited.

**Mobile Shadow**

- **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 the master will terminate before 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 crash 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, \textit{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 44-52). The master exception handler spawns a new shadow and initialises the shadow to become the new master, i.e. $\textit{master} = \text{true}$.

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.

4. A Case Study Application

This research employs a case study application using IBM Aglets [10] to provide an experimental environment for the simulation of agent server crash failures and subsequent analysis of the exception handler design. Mobile agents are used to determine the best deal offered by suppliers regarding delivery date and price. An earlier paper [8] details the general requirements. This section details the interaction with a mobile agent and supplier.

4.1. The supplier interface

Every supplier hosts an agent server to provide a static \textit{DataSourceManager} agent that visiting mobile agents interact with to retrieve a proxy to the product catalogue (figure 3). A \textit{DataSourceManager} agent listens for \textit{connect} messages and responds by spawning a \textit{SalesAgent} and returning its proxy to the sender. A \textit{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 \textit{execute_query} message to the \textit{SalesAgent} proxy with the following parameters:

- \textbf{product class}: the class of product, e.g. hard drive.
- \textbf{product criteria}: search criteria, e.g. a cpu has criteria regarding speed and hardware interface.
- \textbf{order}: order constraints, i.e. stock and delivery date.

![Figure 3 Interaction sequence for querying supplier product catalogue](image)

In response to an \textit{execute_query} message the \textit{SalesAgent} returns a list of matching items that satisfy both the product and order criteria. For a given business domain the \textit{SalesAgent} must implement an interface for product queries. For example “public Estimate[] QueryCase(String \textit{formFactor}, int \textit{intCapacity}, int \textit{extCapacity}, String \textit{style}, Order \textit{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 \textit{Order} object. A matching product is represented as an \textit{Estimate} object. An \textit{Order} object encapsulates the order constraints including total stock and required delivery date. The \textit{Estimate} object encapsulates matching product details including item id, total stock and delivery date. The appropriate method is invoked in the \textit{SalesAgent} depending upon the product class provided in the \textit{execute_query} message. Figure 3 illustrate a simple example for a 1.2Ghz socket A cpu.

5. An experiment

The aim of the experiment 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 8GB IDE hard drives. For simplicity, each supplier represents its product catalogue using mysql 3.23 with JDBC driver 2.0.8. The experiment will be performed on a 10mbps LAN using four 64MB Intel Pentium II 400Mhz (celeron) PC’s running RedHat Linux 7.2 and IBM Aglets [10].
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 4) is initialised with the total number of suppliers to visit (`iTripSize`). A random number is generated (`iCrashIndex`) to represent the nth 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 4 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 < iCrashIndex <= 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 will be 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 5 will be measured in the event of a single agent server crash.

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

```
Master crash  
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. Time for master to detect shadow failure
2. Time for master to spawn replacement shadow
```

**Figure 5 Performance measurements**

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.

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 6. 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.

**Figure 6 Mobile shadow trip times**

The mobile shadow scheme provides an average normal round trip time of 2.1s. 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 13.8s (11.7s increase) compared to 14.6s (12.5s 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 execution. The following sections analyse the performance overheads for a master and shadow crash.

6.2. Exception handler overheads

Figure 7 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 159.4ms (0.16s) and 76.5ms (0.08s) respectively.

The largest overhead is the time taken for a shadow to detect its parent’s agent server crash (4.4s). 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 pings until it detects a crash and notifies the blocked shadow. Similarly the master pings its shadow concurrently.

6.3. Evaluation

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 0.4s.

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 [16] 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 [16] 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 fails to retry or locate an alternative software service.

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 [8] 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. [2][4][5], that replicate a mobile agent at each stage of 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.

Figure 7 Performance overheads of mobile shadow exception handler
7. Conclusions and Future Work

The paper has presented an exception handler design to tolerate crash failures of mobile agents that satisfy the failure assumptions outlined in section 3. Results and analysis show that the scheme offers a significant 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 [17]. Finally the exception model could be extended for mobile agent groups [18][19].

Further research is underway regarding an exception model for groups of mobile agents. A process group abstraction, e.g. [20][21], is traditionally used for reliable event communication. Adapting process groups for mobile agents is difficult since the location of mobile agents dynamically changes.

8. Acknowledgements

Simon Pears is funded by an EPSRC doctoral studentship. The work is funded partially by the DTI/EPSRC e-demand and EPSRC IBHIS projects.

References