An Approach to Rollback Recovery of Collaborating Mobile Agents
Taha Osman, Waleed Wagealla, and Andrzej Bargiela, Member, IEEE

Abstract—Fault-tolerance is one of the main problems that must be resolved to improve the adoption of the agents’ computing paradigm. In this paper, we analyze the execution model of agent platforms and the significance of the faults affecting their constituent components on the reliable execution of agent-based applications, in order to develop a pragmatic framework for agent systems fault-tolerance.

The developed framework deploys a communication-pairs independent checkpointing strategy to offer a low-cost, application-transparent model for reliable agent-based computing that covers all possible faults that might invalidate reliable agent execution, migration and communication and maintains the exactly-once execution property.

Index Terms—Checkpointing, cooperative systems, fault tolerance, mobile agents, rollback recovery.

I. I NTRODUCTION

T HE PAST few years witnessed the emergence of mobile agents as the most promising technology that employs computational intelligence in order to take advantage of the global connectivity of the Internet. Mobile agents are autonomous software entities that can be dispatched from one computer and transported to a remote computer for execution. Arriving at the remote computer, they present their credentials and obtain access to local services and data. Hence, a mobile agent accesses network resources more efficiently because it moves to their network location rather than transferring multiple requests and responses across congested network links. Moreover, since it is not in continuous contact with the originator node, the agent is not affected by sudden loss of connection, and can continue its task even if the user’s computer powers down or disconnects from the network. When the user reconnects, the agent returns to the originator node with its results [1].

Another advantage of mobile agents is the nonintrusive customization of server and client software. Agents can add new operations to the repertoire of services offered by the server to effectively customise the server for that particular client. Similarly, service providers can update the access logic of client software.

Despite these advantages, only few real agent-based applications exist today. To promote wide adoption by application developers, the agent technology needs to resolve important problems such as security, trust, and reliability [2]. These problems are critical for a large number of Internet-based applications such as e-commerce and GPS navigation systems [3].

Many academic and commercial systems offer agent-based computing platforms, such as aglets from IBM [4], grasshopper from IKV [5], FIPA-OS from Emorphia [6], and Jade from TILAB [7]. Most these platforms support the essential features of agents such as autonomy, intelligence, and mobility, but they all lack comprehensive support for fault-tolerance. In the glossary of the IEEE, fault tolerance has been defined as: “The ability of a system or component to continue normal operation despite the presence of hardware or software fault” [8]. Fault-tolerance is especially important to distributed systems because of the interdependency of their components (processes), hence, we need to take into account the state of distributed system processes and the state of the communication channels between them. In agent-based distributed systems the situation is further exasperated because agent platforms communicate live agents (agent migration) as well as inter-agent communication messages. However, despite its criticality to agent systems, fault-tolerance is a very complex problem that is difficult to address by the agent platforms in their core releases partly because the technology is still in its infancy and partly because of the diverse reliability requirements of potential agent applications [9], for instance, in a flight reservation system a restart (relaunch) of the agent might be sufficient if the agent crashes, while a higher level of reliability management is required for an e-commerce system where we need to consider the state of the agent and in-transit transactions at the moment of failure.

Unfortunately, a lot of research work on agents’ reliability focussed on refitting conventional fault-tolerance methodologies to the agent paradigm without carefully analysing the working of existing agent systems and the applications that build on them. This has resulted in overlooking important issues such as the reliability of the communication channels between interacting agents as in [10] and the response-time requirements of potential agent applications, which affects the choice of the appropriate methodology for recovering the agent-based applications from faults affecting agents’ execution, migration, and interaction [11].

Our approach addresses these problems by initially studying the execution model of agent platforms according to a practical reading of the hardware and software platforms in which agents execute today. Then we assess how the failure of the components of this execution model affects the reliable execution of agent-based applications in order to develop a pragmatic framework for agent systems fault-tolerance. This framework builds
on checkpointing-based reliability theory to provide low-cost, but comprehensive protection to agents against possible faults during their execution, migration and interaction. The ultimate goal is to make agent-based computing more attractive to reliability-critical applications.

The rest of this paper is organized as follows: Section II describes the motivation for this research and discusses related work. Section III analyzes the agent execution and failure model. Sections IV and V discuss the fault-tolerance framework for mobile agents. Section VI presents a formal proof of correctness to the proposed protocol, and finally Section VII concludes the paper and presents plans for further development.

II. MOTIVATION AND RELATED WORK

To build a fault-tolerance model that is attractive for application developers, we need to develop a reliability framework that is transparent to the agent applications, i.e. seamlessly integrates into the agent-computing environment while minimizing the overheads of managing the fault-tolerance layer. Therefore, we need to analyze the characteristics of potential agent applications to tailor-fit the fault-tolerance model to their reliability needs such as how stringent are the response requirements of the applications and the significance of considering the state of the inter-agent communication channel, which affect the choice of fault-tolerance methodology and strategy.

There are two main approaches to fault-tolerance of distributed systems: replication and checkpointing. Replication techniques rely on executing replicas of the application processes (agents) on redundant hardware, then the application should be able to continue executing reliably as long as at least one replica is alive. The checkpointing approach makes a substitution of failure by a so-called checkpoint image. A checkpoint image is a copy of a failure-free state of an application that has been saved prior to fault occurrence. The restoration of a previous checkpoint upon recovery is called rollback recovery.

Khalid Nagi in [12] presents an approach for building robust multiagent systems based on transactional agents. The proposed approach relies on imposing a structure for representing the agent operational logic (plan) within the domain of the database technology. The multiagent system then relies on well-established robustness techniques within the database technology such as maintaining the serializability and reliability of transaction models [13] to increase the system robustness. The proposed structure builds directly on the FIPA agent management model, therefore it represents a valuable contribution toward improving the reliability management of FIPA-based agent platforms. However, Nagi’s proposed approach is tightly coupled with the database technology and necessitates the adoption of a transaction-based infrastructure to design the agent execution plan, which is different from our objectives of providing a generic and transparent approach to agent systems reliability. For instance, the execution state of the agent in the proposed approach in [12] is not taken into account, but it is required to identify parts of the agent mental states that must be recovered. The approach also does not deal with agent migration.

Silva and Popescu in [14] describe an approach that relies on combining agent replication and transaction-controlled mobility to provide reliability for distributed agent applications. Their approach doesn’t consider the state of the inter-agent communications channels upon recovery and thus does not cater for duplicate and out-of-order messages. The approach also doesn’t guarantee the exactly-once execution property. Maintaining this property ensures that an agent recovered from a failed state (whether by means of rolling back to earlier saved execution state (checkpoint) or by electing a live replica) does not re-execute an itinerary step previously performed by the failed agent, thus compromising the integrity of services accessed by multiple agent replicas [15]. This problem usually arises from imperfect detection of agent failure.

The latter problem is handled in another replication-based approach [16]. Here the exactly-once problem is tackled by agreeing a consensus between simultaneously executing agent replicas. However, their fault-tolerance model also doesn’t consider inter-agent communications, which might be crucial for applications requiring multiagent collaboration.

Checkpointing offers a low-cost alternative to agent systems reliability, where live replicas running on redundant hardware are not required. Checkpointing is easier to implement, as the management of consensus between many replicas is not required. It also fits naturally into the agent computing model since serializing the agent code in preparation for migration effectively constitutes taking a checkpoint.

In-principle, replication techniques should have smaller overhead than checkpointing methods, which can block the application execution while retrieving a previously stored state of the agent (checkpoint) from stable storage.

However, we argue that the mobile agents computing paradigm is not the natural computing engine for high-performance applications with strictly constrained response time, for which the checkpointing approach overhead can be regarded intolerable. The agent code has to be interpreted to support the portability necessary for agent mobility, which slows-down the performance of the agent code in comparison to executables that are precompiled into native machine code. Moreover, there is the overhead of marshalling/unmarshalling the execution code (byte-code serialization) and agent load/start-up time as agents travel their itinerary to fulfill a user task [17]. These overheads continuously and incrementally increase the running time of the application, which prevents attaining the maximum performance from the computing platform. The agent-computing paradigm was primarily designed to enhance the human-computer and computer-computer communication rather than delivering high performance.

There is very few reported work on checkpointing-based agent fault-tolerance. An early contribution in [18] utilizes a checkpoint manager in every Java Virtual Machine (agent space) for both error detection and recovery of mobile agents. Their checkpointing scheme caters only for the reliability of agent communications within the same space (intra-site), but the synchronization of execution checkpoints and distributed agent communication (inter-site) is assigned to the application programmer.

One of the main contributions in agent rollback recovery is the James platform [19]. The platform provides schemes for error detection, checkpointing and restart of failed agents, and a
reliable migration protocol that deals with network partitioning. The proprietary mobility protocol of the James platform relies on weak migration, i.e., proxies are used at the remote platform instead of physically deserializing and transporting the agent code. This significantly complicates the fault-tolerance protocol because we have to take into consideration the state of the acting proxies when saving the execution state of the agents, i.e., the execution state of the agents is no longer autonomous. The protocol can be further complicated by inter-agent messaging, which is not taken into consideration in the James platform.

Mohindra et al. [20] proposed a reliability scheme that exploits redundancy in nondeterministic constructs in the agent language to achieve agent’s tolerance to failures. Their programming model presumes that there is more than one way (path) to arrive at the correct results; the paths are connected by choice points. If a taken path (choice) fails, a rollback service is used to reset the script’s local state to what it was just before the choice was made. The script’s paths are assumed to be rollbackable, but no mechanism is provided to undo the actions of nondeterminant operations, which would be the case with financial transactions with an external database for example. While this approach might be attractive to applications with inherent redundancy in the programming model, it is too restrictive to adopt as a generic approach agent systems’ fault-tolerance. Here also the state inter-agent communication channel is not considered in the rollback recovery scheme.

All the discussed papers offer a partial solution to agent systems fault-tolerance. We intend to provide a comprehensive solution that maps to a realistic execution model of agent applications and advocates low fault-management cost and ease of integration.

III. ANALYSIS OF THE AGENT EXECUTION AND FAILURE MODEL

The utopian concept of agents freely roaming Internet sites performing tasks on behalf of the user is clearly unrealistic. Attempts to standardize agent platforms have resulted in the establishment of two main standards, MASIF [21] and FIPA [22]. Agents belonging to specific platforms affiliating to these standards can collaborate to achieve a common goal via inter-agent messaging. However, agents can only migrate to and execute in remote sites if the site’s hosts run a compatible agent platform and the agent has the credentials to surpass the site security firewalls.

Both replication and rollback recovery techniques necessitate the availability of redundant hardware nodes that can take-over the agent-execution upon failures. Hence, these additional nodes must not only run the same agent platform, but also crucially provide identical access to the resources/services that the failed node maintained.

Some agent platforms support weak migration [24], where only the agent data information (i.e. the values of the internal variables) is transferred. The advantage of this migration scheme is the small size of transferred information as opposed to that of strong migration, where the code and execution state (i.e., the stack, heap, program counter, etc.) is transported. The size of the transferred data can be further compressed by selecting the variables making up the agent state. However, weak migration schemes do not map into reliability models because migrated agents cannot resume execution from the point prior to migration, which constitutes a valid checkpoint to rollback to that is consistent with the state of other collaborating agents.

Another important point to consider when designing fault-tolerant agent systems is the state of services that the agents interact with. Although overlooked by many researchers, it is absolutely essential for reliability-critical applications to ensure that an update is done exactly once [25]. An example of such service can be crediting or debiting an e-account. The duplication of the financial transaction because of probable imperfect failure detection or failure to agree a consensus is intolerable.

The fault-recovery protocol should guard against faults occurring under the influence of the application external environment, i.e. errors caused by faults in the underlying hardware platform. Transient hardware failures, caused by a temporary memory fault for example, might affect the executing agent only, while permanent faults caused by host failure will crash the running agent platform and all the executing agents. The integrity of inter-agent messaging can also be violated by the failure of the sending or receiving agent. Host failures can also disrupt agent migration.

An often-overlooked issue is the coordination of recovering the agents in collaborative environments. Here we must take into consideration how the failure of a single agent or a communication transaction can affect the consistency of the global state of collaborative agents applications. Classic distributed fault-tolerance issues such as domino effect and duplicated messages [23] are also relevant for such agent applications.

The conclusion is that fault-tolerance solutions for agent systems must build on a realistic execution model. In this model, agent service providers have to grant the availability of a homogeneous pool of hosts, running the same agent platform and maintaining uniform access to resources that the agents might interact with. Fault-tolerant agents must be able to recover from permanent and transient node failures, whilst maintaining the exactly-once property of accessing these resources and a consistent global state of the agent application. This conclusion is true for each service that requires distinctive hardware/software set-up or access, even within the same enterprise network.

IV. OVERVIEW OF THE FAULT-TOLERANCE FRAMEWORK

While the hardware environment on which agents execute is no different to that hosting traditional distributed applications, the agent middleware and its distribution fundamentally differ primarily because of its need to support execution and collaboration of mobile programs. Many configurations were suggested for agent-based systems, but we base our configuration on the practical requirements for agent fault-tolerance, i.e., a homogeneous pool of hosts where agents can recover, and how this configuration logically maps into enterprise networks supporting several agent-assisted services.

The framework environment is illustrated in Fig. 1 and comprises the following components.

• A space is the environment where agents perform steps of execution. Each space resides in a separate node and con-
consists of the agent software platform and resource variables that the agents access during step execution.

- A region is a homogeneous pool of spaces, each capable of hosting agents to a particular agent service. One space within a region actively executes the agents while at least one more space is a potential stand-by for rollback recovery of failed agents.
- Each group of regions within the same enterprise network providing agent-based application services has a recovery manager, which executes in a fail-safe node running a fault detection mechanism and initiating agent recovery. The recovery manager’s reliability can be achieved using well-published replication and election voting mechanisms [26]. The recovery manager also maintains access to a persistent data storage, where the agents checkpoints and recovery bookkeeping is held.

The need to manage the rollback recovery of the collaborating agents imposed the need for an autonomous, central recovery manager. Decentralized schemes relying on neighboring nodes (spaces) for error detection and recovery are efficient for applications with completely autonomous agents as in [14]. Our design requires only one recovery manager for all the agent-based applications (services) per enterprise network, thus minimizing the cost of the fault-tolerance scheme. The disadvantage of the centralised entity is that it is a potential communication bottleneck for the system that can incur an overhead on the execution and recovery time of the agent-based application. This overhead will be discussed when we explain the dynamics of our fault-tolerance protocol in the following sections.

V. ENABLING FAULT-TOLERANCE

The design of our fault-tolerance protocol follows the realistic view of the agent execution model emphasized throughout this paper. To achieve their goal, agents execute, communicate, and migrate. Here we consider how to safeguard agent-based applications while the agent is performing each of these operations. The important issue of detecting faults affecting the agent’s operation was treated comprehensively in our earlier work on agent fault-tolerance. Our results were published in [27], where we presented—consistently with this work, a decentralized error-detection structure that divided the agent dynamic distributed system into network-partitioning proof spaces that provided for the transparent detection of faults that might affect agent execution and migration.

A. Tolerating Faults Upon Agent’s Execution

The pseudo-code for fault-management during agent’s execution within a space is shown in Fig. 2. At each space, checkpoints of executing agents are taken periodically. The default checkpointing interval is set by the service provider, but can

---

at regular checkpointing intervals do
interrupt agent execution
compile agent checkpoint [execution image, Agent State # (AST)]
log checkpoint to stable storage
increment AST
delete previous checkpoint
end

upon update to external service do
begin transaction
modify external service state
trigger an agent checkpoint
commit transaction
end

Fig. 2. Agent fault-tolerance during in-space execution.
be overridden by the agent clients. Solutions presented in [28] and [19] force the application developer to include checkpoint method calls within the agent code. This approach compromises the transparency of the fault-tolerance framework and should be avoided. Instead, checkpoints should be triggered by the fault-tolerance layer at each space by raising exceptions interrupting agent execution.

The checkpoints include the agent execution state image and agent state counter (ASt). Upon successful completion of the checkpointing process, the tentative checkpoint is sent to stable storage and the previous cycle (state) message log and checkpoint are cleared. Finally the global state of the agent is incremented.

All checkpointing and rollback recovery techniques assume piecewise deterministic execution of the distributed processes (agents) [29], i.e., given an initial state and an ordered collection of incoming messages, the application always behaves in a consistent manner. We relax this assumption by ensuring that critical updates to the external environment are executed exactly-once, rendering unnecessary the maintenance of the agents’ deterministic global state at that point.

Maintaining the exactly-once property is the responsibility of the application developer in [19], where agents are allowed to execute in duplicate, and the application developer must determine which result should be used according to a best-effort and atomic execution schemes. In contrast, [16] proposes an application transparent solution. However, the solution is based on a complex approach solving the consensus agreement problem between concurrently executing replicas and incurs a significant bookkeeping overhead.

In contrast to these solutions, our approach is checkpointing-based and we can offer a simple solution to the exactly-once problem by forcing agents to take a checkpoint when an update to an external service state is committed. The update to external state and taking a checkpoint must be joined in an atomic transaction to prevent interleaving the two operations. Thus we guarantee that a restarted agent will not re-execute the update after rollback.

B. Tolerating Faults During Agent Migration

Migration takes place when an agent decides to move to another region providing different services, for load-balancing, or for seeking a better quality of service (QoS) as common for agents of mobile devices in wireless networks [30]. As we mentioned earlier, our design only supports full-migration schemes, i.e., given an initial state and an ordered collection of incoming messages, the application always behaves in a consistent manner.

Before starting the migration process, all transient messages in the communication channel to the migrating agent must be flushed. This avoids complex and costly inter-region message forwarding scheme for transient messages at migration point. Flushing can be achieved by sending an “is channel empty?” acknowledgment request to collaborating agents.

Next a checkpoint is taken of the agent state and is placed on the space output queue as explained in Fig. 3. The space then enters an atomic, two-phase commit transaction to guarantee agent delivery. The checkpoint-to-transport is only deleted once the agent is safely restarted at the destination space.

On the destination side, the committed checkpoint of the agent is added to the destination region’s stable storage as an initial point of recovery.

C. Tolerating Faults Affecting Inter-Agent Communication

Inter-agent messaging is an important aspect of the agents computing paradigm. Agent migration can only have an advantage if the application requires intensive communication between remote hosts, but then intra-agent messaging has to be guarded against possible faults. In fact, some of the major agent platforms such as FIPA-OS [22] and Zeus [31] only support agent communication as means of agent collaboration.

To guarantee the successful recovery of any distributed system, the system must be recovered into a consistent state, i.e., a state beyond which rolling back any of the system’s processes (agents) is not necessary. A consistent state can only be achieved by considering the state of the communication channel between the agents as well as the execution state of the agents themselves.

There is very little published work on fault-tolerant techniques that take into account the state of the inter-agent communication channels. One of the major contributions is the work described in [32]. Their work focuses on reliable delivery of agent messages to highly mobile agents and heavily depends on strict management of FIFO-assumed channels between agent spaces. It is not clear how the reliable message delivery is coordinated with the distributed snapshot during checkpointing and rollback. Our pragmatic insight into the structure and capabilities of the agent execution model allows us to present a simpler approach that clearly defines the correlation between inter-agent messaging and execution checkpointing to maintain a consistent state of the agent application upon recovery.

Keeping in mind the objective of maintaining low overhead of introducing fault-tolerance, we opted for sender-based message logging strategy. Any variation of consistent checkpointing [23]—which requires synchronizing the checkpointing of all the collaborative processes (agents), will have a high coordination overhead, which can be prohibitive in the dynamically changing environment of mobile agents.

Optimistic, sender-based logging [33] of inter-agent messaging allows for checkpoints of agents to be taken in-
**Fig. 4. Fault-tolerance for collaborating agents.**

<table>
<thead>
<tr>
<th>upon sending messages do</th>
</tr>
</thead>
<tbody>
<tr>
<td>augment message with Send Sequence # (SSN) and Agent State (AS)</td>
</tr>
<tr>
<td>log message to stable storage</td>
</tr>
<tr>
<td>send message to receiving agent</td>
</tr>
<tr>
<td>increment SSN</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>upon receiving messages do</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse received message</td>
</tr>
<tr>
<td>if (received message SSN (RSSN) &gt; local SSN of same agent (LSSN) + 1) then</td>
</tr>
<tr>
<td>put message on local consumption queue</td>
</tr>
<tr>
<td>else if (RSSN = LSSN +1) then</td>
</tr>
<tr>
<td>forward message for consumption</td>
</tr>
<tr>
<td>increment LSSN</td>
</tr>
<tr>
<td>while (exists queued message with RSSN = LSSN+1)</td>
</tr>
<tr>
<td>forward message for consumption</td>
</tr>
<tr>
<td>increment LSSN</td>
</tr>
<tr>
<td>endwhile</td>
</tr>
<tr>
<td>endif</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>ignore received message</td>
</tr>
<tr>
<td>endif</td>
</tr>
</tbody>
</table>

| if (received AS > local AS of same agent) then |
| purge all logged messages to sending agent |
| update local AS with received AS |
| end |

Dependently, providing complete decoupling of sending and receiving agents and transparency to the agent location.

Most mobile agent systems are built on top of TCP/IP which is a reliable transmission protocol, so it is fair to assume that communication channels are reliable [34]. However, in checkpointing-based techniques a rolled back agent might undo the receipt of a message, therefore, we need to coordinate the processes of checkpointing the agent execution state and message logging to bring the system to a consistent state upon recovery.

We also need to consider that messages might not be received in the order they were sent in, because the agent platform message transport system might implement asynchronous messaging. For instance, the FIPA-OS agent platform uses RMI, CORBA, and HTTP for inter-agent communications, which are all built on TCP. However, the messaging in FIPA-OS is asynchronous, therefore it offers no guarantee that two messages sent to a single destination will arrive in any particular order.

The following scheme (see Fig. 4) is suggested for ensuring reliable inter-agent messaging in our fault-tolerance scheme.

**Upon Sending Messages.** Each message is augmented with a sequential message send number (SSN) and the agent state counter (AS). A copy of the message is broadcasted to the recovery manager to be saved in the stable storage, then the message is sent to the destination agent and the message number is incremented. Tagging messages with sequential numbers and the agent state counter allows the scheme to accurately relate messages to the execution state of agents, i.e., taken checkpoints.

**Upon Receiving Messages.** Each receiver agent holds a local sequence number (LSSN) and local agent state counter of the last message received from each agent it engaged in a conversation (inter-agent communication) with.

Received messages are forwarded for local consumption only if they are received in the correct sequence, i.e. received SSN (RSSN) is equal to LSSN + 1. Out-of-order messages with RSSN > LSSN + 1 are queued locally and only forwarded for consumption if the delayed preceding message in sequence, i.e. RSSN − 1 is received. The local sequence number (LSSN) is incremented following message consumption. If the above two conditions are not met, this means that the message is duplicated and can be safely ignored. A message sequence scenario illustrating how our fault-tolerance protocol handles inter-agent messaging is illustrated in Table I.

The receiving agent also checks if the sending agent state counter has been incremented indicating that it took a checkpoint recently. Then, the recovery manager is signalled to purge all receiver-side logged messages during earlier exchanges with the sending agent.

Although the recovery manager can potentially represent a bottleneck for communication-intensive applications, we argue...
that this burden is minimal because the coordination of the message checkpointing process is carried out by the communicating pairs, while the recovery manager’s role is limited to the log and replay of messages into/from the regions stable storage.

If the application reliability requirements are severe and necessitate guarding against incidents of complete breakdown of the physical communication network, we suggest that an application-directed approach is deployed to guarantee the reliable delivery of messages. For instance, an acknowledgment/retransmission mechanism can be deployed. We believe that the time overhead of managing such mechanism for every sent/received message transparently will be too high especially for communication intensive applications, thus defeating our objective of low overhead and cost-effective recovery mechanism. Dealing with such faults can be far more efficiently realized at the application level by developer’s instructions, where for example retransmission can be arranged for a subset of critical communication transactions.

D. Failure Recovery

All recovery operations are initiated by the regions recovery manager (see Fig. 5) upon receiving fault notification from the error detection mechanism [27]. We discussed earlier that there are three faults that invalidate the execution of agent applications.

1) Transient faults only affect the executing agents. Agents are restarted on the same space from the last checkpoint saved on stable storage and are re-sent all messages logged since their last checkpoint was taken by the recovery manager. Depending on the agent platform set-up, it might need to be notified about the new location of the agent. As mentioned earlier, rolled-back agents will not violate the exactly-once property because checkpointing is atomic to modifications to external environment state.

2) If the space completely crashes because of a permanent node failure, then the above recovery procedure needs to be repeated for all agents executing in the space at the time of the crash. The difference is that agents will be restarted on a space running on another node within the same region.

3) If an acknowledgment about agent commit during a migration process times-out, the sending space terminates the transaction and engages in a new one. If the failure persists, the origin space attempts to send the agent to an alternative space at the destination region.

VI. PROOF OF CORRECTNESS

The following assumptions need to be re-emphasised before presenting the formal proof of correctness.

- The communication channels are assumed reliable, but asynchronous agent communication is allowed.
The recovery manager and stable storage are assumed fail-safe. This requirement was argued when we discussed the hardware bed for our framework earlier.

To maintain the transparent characteristic of our protocol, agent-based applications are assumed to be deterministic. However, by ensuring that critical updates to the external environment are executed exactly-once we can tolerate nondeterministic interactivity with external services.

**Lemma 1**: For each failed agent, there always exists a valid checkpoint to rollback to.

*Proof*: The initial agent program code represents the very first checkpoint. Thereafter, the successful termination condition of the migration protocol results in taking a checkpoint every time the agent migrates to a new node in its itinerary. Checkpoints taken during in-space execution do not overwrite a previous checkpoint until they are safely stored, and since the recovery manager and stable storage are assumed reliable, we prove the lemma.

**Lemma 2**: A rolled-back agent maintains the consistency of inter-agent collaboration and external services state.

*Proof*: Our protocol deploys sender-based checkpointing scheme. The following conditions have to be met to prove the lemma.

1) All messages are logged for possible replay. Since communication channels and stable storage are assumed reliable, only the failure of the sending agent can prevent message log, but unsent messages will be resent once the agent restarts from the last checkpoint. Messages are only deleted from stable storage if the recipient has progressed to a new execution interval and the log is no longer required.

2) Rolled-back agent will not repeat a transaction with external service. Committing an update to external service state is atomically locked with taking a checkpoint, therefore the update can only be done exactly-once.

3) Restarted agent resumes collaboration correctly. From 1) above, and taking into account that channels are reliable, the rolled back agent will eventually receive all messages that were lost due to the agent failure since last checkpoint was taken. Thus, since we assume the agent application to be deterministic and critical updates to the external environment are not reversible from ii) above, the agent is expected to generate consistent (valid) output messages to collaborating agents after rollback.

4) Out-of-order and duplicate messages are tolerated. All messages are stamped with a sequential number and the agent state number. This information is compared against the history of received messages from the same agent upon message receipt. Out-of-order messages, probably caused by communication delays, are queued and only released for consumption in the correct sequence. Messages re-received as a result of sender’s rollback are ignored.

By inference from 1)-4), we prove that a rolled back agent maintains the consistency of the multiagent collaborative application.

**Lemma 3**: Migrating agents resume execution into a consistent state at the new node.

*Proof*: If the destination node fails, or an error occurs during transmission, the migration protocol will be aborted and repeated until it eventually terminates provided a destination node eventually becomes available. If the originator node fails, the agent will be restarted on another node and migration re-attempted. Since the message logs are kept in the fail-safe stable storage and are accessible to all the regions within the enterprise network, then from lemma “2” the checkpoint taken at the new space will be consistent.

**Theorem 1**: All the agents of the multi-agent application can be recovered to a consistent state after failures. This assertion holds for agents restarted after migration as well as after failure.

*Proof*: From lemma “1,” there will always be a checkpoint to rollback to at any stage of the agent execution. Lemma “2” proves that agents rolled back to these checkpoints, together with the safely logged inter-agent collaboration messages, maintain the consistency of the overall multiagent application execution. Finally, lemma “3” proves the latter for migrated agents.

**VII. CONCLUSIONS AND FUTURE WORK**

This paper presented a novel framework for mobile agents’ fault-tolerance that is based on a pragmatic outlook to agent-based computing. Our analysis established that for each agent-based service requiring high availability, service providers must grant a homogeneous pool of hosts, running the same agent platform and maintaining uniform access to resources that the agents might interact with. Further study of the agent execution and failure model raised frequently overlooked issues such as the criticality of inter-agent communications and the flexibility in agent-applications response time requirement that allow the utilization of a low-cost, transparent agent reliability methods based on checkpointing techniques.

The presented framework for agent reliability covers all possible faults that might invalidate reliable agent execution, migration and inter-agent communication. We proposed an application-transparent, cost-effective alternative to replication-based approaches based on checkpointing and rollback recovery. Our checkpointing strategy provides for the complete decoupling of sending and receiving agents by building on an optimistic sender-based logging approach that relies on the collaborative agents to manage a global consistent state of the application, thus avoiding the heavy costs of deploying a central coordination policy. The suggested protocol supports strong (full) agent migration schemes because weak migration schemes don’t transfer the agent execution state, thus preventing resuming the agent execution from the checkpoint prior to migration. The proposed framework also offers a simple solution to the exactly-once execution problem of recovered agents that integrates directly into our checkpointing strategy, without the need for complex consensus-management operations.

We recognize that the option of asynchronous inter-agent messaging is adopted by agent platforms such as FIPA-OS, hence, a strategy was included in our fault-tolerance protocol to manage out-of-order and duplicate messages in addition to
the main task of coordinating the process message logging with checkpointing the agent execution state and to bring the system to a consistent state upon recovery.

We need to emphasise that it is unfeasible to impose a rigid generic solution to agent systems fault-tolerance because implementing the reliability layer will significantly depend on the agent management polices of the targeted agent platform. For instance, some agent platforms, such as Grasshopper [5], do not implement a peer-to-peer agent communication, but inter-agent communications are executed by remote invocation of methods on a proxy representing the destination agent, which will require a reconsideration of the explicit message logging approach adopted in this work. Similar issues can arise for platforms implementing weak-mobility schemes. Nevertheless, owing to the comprehensive coverage to all aspects of agent systems fault-tolerance, we believe that the described framework represents a customisable blueprint for introducing fault-tolerance to agent computing platforms, making them more attractive for developers of reliability-critical Internet-based distributed services. A formal proof of our solution claim was presented.

Our plans for further research include implementing a prototype system based on the designed framework to provide a fault-tolerant wrapper for one of the existing agent platforms. The implemented prototype should allow us to practically study the overheads of fault-tolerance on the running time of agent applications and equally important the recovery time of failed agents, and develop fault-management tuning tools to adjust the reliability/performance balance to the varied fault-tolerance and raw throughput requirements of distributed applications.

REFERENCES

Taha Osman received the B.Sc. honors degree in computing from Donetsk Polytechnical Institute, Ukraine in 1992, the M.Sc. degree in real-time systems from Nottingham Trent University (NTU), Nottingham, U.K., in 1993, and the Ph.D. degree developing fault-tolerant distributed processing environments in 1998.

He is a Lecturer in the School of Computing and Mathematics at NTU. His current research interest is focused on the utilization of mobile agent systems for developing robust, open-access wireless computing environments.

Waleed Wagealla received the B.Sc. degree in computer science from University of Khartoum, Khartoum, Sudan, and the Ph.D. degree in computer science from Nottingham Trent University, Nottingham, U.K.

He is currently a Research Fellow at the department of Computer and Information Sciences, the University of Strathclyde, Glasgow, U.K. His recent work is in mobile agents technology and trust management in pervasive computing systems.

Andrzej Bargiela (M’96) is Professor and Director of the Intelligent Simulation and Modeling Lab at the Nottingham Trent University, United Kingdom.

Since 1978, his main research focus has been the processing of uncertainty in the context of modeling and simulation of various physical and engineering systems. This research involves the development of algorithms for processing uncertain information, investigation of computer architectures for such processing, and the study of information reduction through visualization. He published numerous papers and is a co-author of a research monograph on Granular Computing (see http://www.doc.ntu.ac.uk/RTTS). He is a Visiting Professor at the University of Alberta, Edmonton, AB, Canada. In 2002 he was elected to serve as Chairman of the European Council of the Society for Computer Simulation (SCS).

Dr. Bargiela represented the Nottingham Trent University on the U.K. Engineering Professors Council between 1996 and 2002, and is a member of the U.K. Conference of Professors and Heads of Computing. He is a member of the Institute of Measurement and Control (IMC), British Computer Society (BCS), and the Institute of Electrical Engineers (IEE).