An Approach to Rollback Recovery of Collaborating Mobile Agents

Taha Osman, Waleed Wagealla, Andrzej Bargiela
Department of Computing,
The Nottingham Trent University
Burton Street
Nottingham NG1 4BU
taha.osman@ntu.ac.uk, waleed.wagealla@ntu.ac.uk, andre@doc.ntu.ac.uk

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 analyse 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-one execution property.

1. Introduction

The past few years witnessed the emergence of mobile agents as the most promising technology to take advantage of the global connectivity of the Internet. Combining autonomy of execution and mobility, agents traverse the Internet intelligently performing tasks on behalf of the user and possibly migrating closer to the distributed services location rather than transferring multiple requests across congested network link.

However, 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 and reliability. These problems are critical for a large number of Internet-based applications such as e-commerce and GPS navigation systems [1].

Many academic and commercial systems offer agent-based computing platforms, such as Voyager from Object Space [2], aglets from IBM [3], grasshopper from IKV [4], FIPA-OS from Emorphia [5], and Jade from TILAB [6]. Most these platforms provide for the fundamental features of agents such as autonomy, intelligence, and mobility, but they all lack comprehensive support for fault-tolerance [7]. Fault-tolerance, albeit crucial, 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 [8].

Unfortunately, a lot of research work on agents' reliability focused 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 between interacting agents and the real-time constrains of potential agent applications, which affects the choice of the recovery strategy.

Our approach addresses these problems by initially studying the practical execution model of agent platforms. 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 rest of this paper is organized as follows: section 2 describes the motivation for this research and discusses related work. Section 3 analyses the agent execution and failure model. Sections 4 and 5 discuss the fault-tolerance framework for mobile agents. Section 6 presents a formal proof of correctness to the proposed algorithm, and finally section 7 concludes the paper and presents plans for further development.

2. 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 analyse the characteristics of potential agent applications to tailor-fit the fault-tolerance model to their reliability needs.

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.

Silva and Popescu in [9] 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 reliability of inter-agent communications and thus does not cater for duplicate and out-of-bound messages. The approach also doesn't guarantee the exactly-once execution property, which might compromise the integrity of services accessed by multiple agent replicas. This problem usually arises from imperfect detection of agent failure.

The later problem is handled in another replication-based approach [10]. Here the exactly-once problem is tackled by agreeing a consensus between simultaneously executing agent replicas. However, their model also doesn't cater for faults in inter-agent communications, which might be crucial for applications requiring agent interactivity.

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

However, we argue that agent computing is not suitable for high-performance applications with strictly constrained response time for which such 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 pre-compiled into native machine code. There is also the overhead of marshalling/unmarshalling the execution code (byte-code serialization) and agent load/start-up time as agents travel their itinerary to fulfil a user task [11]. The agent-computing paradigm was primarily designed to enhance the human-computer and computer-computer communication rather than delivering high performance.

Checkpointing offers a low-cost alternative to agent systems reliability, where live replicas running on redundant hardware are not required. Checkpointing is easer 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.

There is very few reported work on checkpointing-based agent fault-tolerance. One of the main contributions is the James platform [12]. 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 de-serializing and transporting the agent code. Weak migration significantly complicates the fault-tolerance protocol since the agents no longer have autonomous execution state. The protocol can be further complicated by inter-agent messaging, which is not taken into consideration in the James platform.

Mohindra et al [13] proposed a reliability scheme that exploits redundancy in non-deterministic constructs in the agent language to achieve agent's tolerance to failures. Their programming model presumes that there is more than way to arrive at the correct result, which 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 scripts paths are assumed to be *rollbackable*, but no mechanism is provided to undo the actions of non- idempotent 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 reliability of inter-agent communication is not considered.

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.

3. 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 [14] and FIPA [15]. 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 [17], where agents are not physically transferred to the remote host, but a proxy is set-up there to act on their behalf. This type of migration significantly complicates rollback recovery of agents because a valid agent state cannot be fully captured.

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 update is done exactly once [18]. 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 whether that is the computing node. Transient 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 environment. 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 [16] 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 HW/SW set-up or access, even within the same enterprise network.

4. 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 Figure 1 and comprises the following components:

- A place is the environment where agents perform steps of execution. Each place resides in a separate node and consists of the agent software platform and resource variables that the agents access during step execution.
- A region is a homogeneous pool of places, each capable of hosting agents to a particular agent service. One place within a region actively executes the agents while at least one more place 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 [19]. 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. Decentralised schemes relying on neighbouring nodes (places) for error detection and recovery are efficient for applications with completely *autonomous* agents as in [9]. Our design requires only one recovery manager for all the agent-based applications (services) per enterprise network, thus minimising 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.

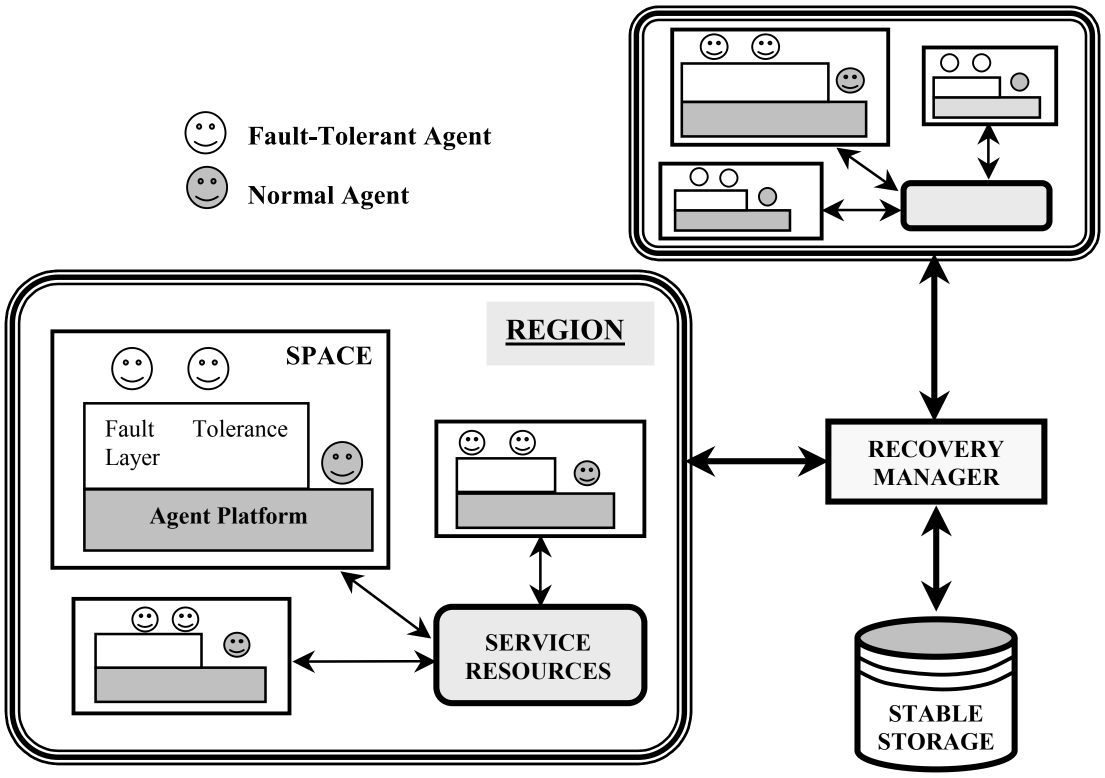

Figure 1: The Agents' Fault-Tolerant Environment

5. Enabling Fault-Tolerance

The design of our fault-tolerance algorithm 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 on agent fault-tolerance. Our results were published in [20], where we presented - consistently with this work, a decentralised 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.

5.1 Tolerating Faults Upon Agent's Execution

At each place, checkpoints of executing agents are taken periodically. The default checkpointing interval is set by the service provider, but can be overridden by the agent clients. Solutions presented in [21] and [12] 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 place by raising exceptions interrupting agent execution.

The checkpoints include the agent execution state image and agent state counter. 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.

During execution the fault-tolerant framework must maintain the exactly-once property. In [12], agents are allowed to execute in duplicate, and it is left to the application manager to determine which result should be used according to a best-effort and atomic execution schemes. In contrast, [10] proposes an application transparent solution is presented. However, the solution, based on solving the consensus agreement problem between concurrently executing replicas is complex and incurs a significant bookkeeping overhead.

In contrast to theses 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.

```
at regular checkpointing intervals do
  interrupt agent execution
  compile agent checkpoint [ exec. 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
```

5.2 Tolerating Faults During Agent Migration

Migration takes place when an agent decides to move to a region providing different services. We adopt a two-phase commit protocol to guarantee the reliable delivery of mobile agents, i.e. agents must be integrally migrated to the destination place and safely started or the operation is aborted.

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 place output queue. The place 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 place.

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.

```
flush transient messages to migrating agent
trigger an agent checkpoint
begin transaction
  send agent checkpoint to destination node
  await acknowledgment of restart success from destination
commit transaction

if migration transaction is successful then
  delete agent checkpoint at sender
  log received agent checkpoint at destination
endif
```

5.3 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 too against possible faults. In fact some of the major AP systems such as FIPA-OS [15] and Zeus [22] only support agent communication as means of agent collaboration.

There is very little published work on the reliability of inter-agent communications. One of the major contributions is the work described in [23]. 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 reliable messaging and execution checkpointing.

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 [16] will have a high coordination overhead for the collaborative processes (agents), which can be prohibitive in the dynamically changing environments of mobile agents.

Optimistic, sender-based logging [24] of inter-agent messaging allows for checkpoints of agents to be taken independently, providing complete decoupling of sending and receiving agents and transparency to the agent location. The following scheme is suggested for ensuring reliable inter-agent messaging:

Upon Sending Messages. Each message is augmented with a sequential message number and the agent state counter. 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 the sequence number and state counter of the last message received from each agent it engaged in a conversation (inter-agent communication) with. If the received message number is equal or less than the last recorded number, this means that the message is duplicated and can be safely ignored. Otherwise, the locally held message sequence number for the sending agent is updated and the message is consumed.

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.

```
upon sending messages do
  augment message with Send Sequence # (SSN) and Agent State (ASt)
  log message to stable storage
  send message to receiving agent
  increment SSN
end

upon receiving messages do
  parse received message
  if (received message SSN ≤ local SSN of same agent) then
    ignore received message
  else
    update local SSN with received SSN
    consume message
  endif
  if (received ASt > local ASt of same agent) then
    purge all logged messaged to sending agent
    update local ASt with received ASt
  endif
end
```

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.

Our scheme does not cater for the loss of messages due to faults in the communication channel. We believe that the overhead of managing an acknowledgment/retransmission protocol for every sent/received message transparently will be too high even though it should be relatively straightforward with our optimistic sender-based logging. Dealing with lost messages can be realised far more efficiently realised at the application level by developer's instructions.

5.4 Failure Recovery

All recovery operations are initiated by the regions recovery manager upon receiving fault notification from the error detection mechanism [20]. We discussed earlier that there are three faults that invalidate the execution of agent applications:

i) Transient Faults only affect the executing agents. Agents are restarted on the same place from the last checkpoint saved on stable storage and are replayed 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.

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

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

```
Upon notification of agent failure do
  restart agent from last checkpoint
  resend logged messages to agent
end
upon notification of node crash do
  restart failed agents on an operative node in the region
  resend logged messages to restarted agents
end
upon migration timeout do
  reattempt migration transaction
  if migration fault persists then
    attempt to migrate to an alternative node within destination region
end
```

6. Proof of Correctness

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

- The communication channels are assumed reliable. Loss of messages or transported checkpoints, except upon agent migration, cannot be recovered. The management retransmission cycles for every exchanged message would invariably complicate the algorithm and incur a huge overhead as pointed-out earlier.
- 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 algorithm, we have to assume that the multi-agent application is deterministic, i.e. given the same set of inputs, e.g. incoming messages, it will always produce the same output. Otherwise, the programmer must take the responsibility for managing inconsistencies resulting from difference in time frames for example.

‡ **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-place 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 algorithm deploys sender-based checkpointing scheme. The following conditions have to be met to prove the lemma:

i) 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.

ii) Restarted agent resumes collaboration correctly. From i) above, and taking into account that channels are reliable, the rolled back agent will eventually receive all messages lost because of its failure since last checkpoint was taken. Thus, since we assume the multi-agent application to be deterministic, the agent is expected to generate consistent (valid) output messages to collaborating agents after rollback.

iii) Duplicate messages are tolerated. All messages are stamped with a sequential number and the agent state number. This information is verified upon message receipt and messages received again as a result of sender's rollback are ignored.

iv) 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.

By inference from i), ii), iii), iv), we prove that a rolled back agent maintains the consistency of the multi-agent application.

‡ **Lemma 3:** *migrating agents resume execution into a consistent state at 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 nodes 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, then from lemma 2, the checkpoint taken at the new place 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 mutli-agent application execution. Finally, lemma "3" proves the later for migrated agents.

7. Conclusions and Future Work

This paper presented a novel framework for mobile agents fault-tolerance that is based on a pragmatic reading of the current status of the agent-computing model. Careful analysis of the agent execution and failure model verified issues often overlooked by agent fault-tolerance work such as the criticality of inter-agent communication reliability and the flexibility in agent-applications response time requirement, allowing the utilization of a low-cost, transparent agent reliability method based on checkpointing techniques.

The presented framework for agent reliability covers all possible faults that might invalidate reliable agent execution, migration and communication. We developed an optimistic sender-based logging checkpointing strategy 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. Our framework also offers a simple solution to exactly-once execution problem of recovered agents that integrates directly into our checkpointing strategy, without the need for complex consensus-management operations.

Owing to its comprehensive coverage to all aspects of agent systems fault-tolerance, the described framework represents a blueprint for introducing fault-tolerance to the 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 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

1. Yager, "Targeted E-commerce Marketing Using Fuzzy Intelligent Agents", *IEEE Intelligent Systems,* Vol. 15, No. 6, pp. 42—45, November/December 2000.
2. ObjectSpace Inc., "Voyager ORB 3.0. Developer Guide", 1999, Available: www.objectspace.com,
3. D. Lange D. and M. Oshima, "Programming and Deploying Java Mobile Agents with Aglets", *Addison Wesley,* 1998.
4. M. Breugst, I. Busse, S. Covaci, T. Magedanz, "Grasshopper - A Mobile Agent Platform for IN Based Service Environments", *IEEE Intelligent Networks Workshop,* Bordeaux, France, May 1998.
5. Poslad, P. Buckle, and R. Hadingham, "The FIPA-OS Agent Platform: Open Source for Open Standards", *In Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents,* pp. 355—368, 2000.
6. Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa, "JADE - A FIPA-Compliant Agent Framework", *Proceedings of PAAM'99,* pp. 97—108, London, April 1999.
7. Osman and A. Bargiela, "Fault-Tolerance for Mobile Agent Systems and Applications", *Workshop 2000 Agent-based Simulation,* pp. 211—221, Passau, Germany, 2000.
8. Tom Walsh, Noemi Paciorek, and David Wong, "Security and Reliability in Concordia", *In Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences,* volume VII, pp. 44—53, January 1998.
9. F. Silva and R. Popescu-Zeletin, "Mobile Agent-Based Transactions in Open Environments", *IEICE/IEEE Joint Special Issue on Autonomous Decentralized Systems,*IEICE Trans. Commun., Vol. E83-B, NO. 5, pp. 973—987, May 2000.
10. S. Pleisch and A. Schiper, "Modeling Fault-Tolerant Mobile Agent Execution as a Sequence of Agreement Problems", *In Proc. of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS),* pp. 11—20, Nuremberg, Germany, Oct. 2000.
11. A. Schill *et .al*., "An Agent Based Application for Personalized Vehicular Traffic Management", *Proceedings, Springer-Verlag.*, pp. 99-111. Berlin, Germany, 1998.
12. L. Silva, V. Batista, and J. Silva, "Fault-Tolerant Execution of Mobile Agents", *In Proc. of the International Conference on Dependable Systems and Networks,* pp. 135—143, New York, June 2000.
13. Mohindra A, Purakayastha A, Thati P, "Exploiting non-determinism for reliability of mobile agent systems", *Proceeding International Conference on Dependable Systems and Networks - DSN 2000,* pp. 144—53, Los Alamitos, CA, USA, 2000.
14. D. Milojicic et al, "MASIF: The OMG Mobile Agent System Interoperability Facility", *Lecture Notes in Computer Science,* LNCS 1477, pp. 50—ff, 1998.
15. P.D. O'Brian, R.C. Nicol, "FIPA -- Towards a Sstandard for Software Agents", *BT Technology Journal*, Vol. 16 No3, July 1998.

16. T. Osman and A. Bargiela, “FADI: A Fault-Tolerant Environment for Open Distributed Computing”, *IEE Proceedings on Software,* Vol. 147(3), pp. 91—99, June 2000.

17. K. Rothermel and M. Schwehm, “Mobile Agents”, *Encyclopedia for Computer Science and Technology,* Vol. 40 - Supplement 25, pp. 155-176, New York: M. Dekker Inc., 1999.

18. M. Strasser M., K. Rothermel, “Providing reliable agents for electronic commerce”, *Proc. of the Int. Conference on Trends in Distributed Systems for Electronic Commerce (TREC),* Springer Verlag, pp. 241—253, Hamburg, Germany, June 1998.

19. R. Guerraouim and A. Schiper, “Software-Based Replication for Fault Tolerance”, *IEEE Computer Journal,* Vol. 30, No. 4, pp. 68—74, April 1997.

20. W. Wagealla T. Osman, A. Bargiela, “Error Detection Algorithm for Agent-Based Distributed Applications”, *Second workshop on Agent-based Simulation,* pp. 106-110, Passau, Germany, 2001.

21. G. Eugene, F. Lubomir, M. Dillencourt, “An Application-Transparent, Platform-Independent Approach to Rollback-Recovery for Mobile-Agent Systems”, *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, Taipei, Taiwan 2000

22. H. Nwana, D. Ndumu, L. Lee, J. Collis, “ZEUS: A Toolkit for Building Distributed Multi-Agent Systems”, *Applied Artificial Intelligence Journal*, Vol. 13 (1/2), pp. 129—185, 1999.

23. Amy L. Murphy and Gian Pietro Picco, “Reliable Communication for Highly Mobile Agents”, *In First International Symposium on Agent Systems and Applications/Third International Symposium on Mobile Agents (ASA/MA'99),* October 1999

24. B. Bhargava and S. Lian: Independen, “Checkpointing and Concurrent Rollback Recovery for Distributed Systems - An Optimistic Approach”, *Seventh Symposium on Reliable Distributed Systems IEEE*, pp. 3—12, 1988.