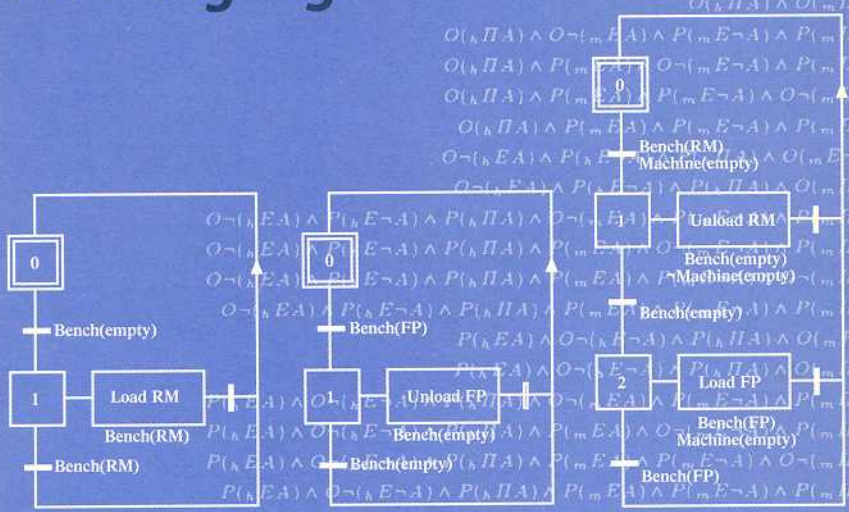
Munindar P. Singh Anand Rao
Michael J. Wooldridge (Eds.)

Intelligent Agents IV

Agent Theories, Architectures, and Languages



Springer

In the proceedings of the Fourth International Workshop on Agent Theory, Architecture, and Languages, 1997, Lecture Notes in Springer Verlag Series, LNAI 1365, pp. 77 – 92

Distributed Storage of Replicated Beliefs to Facilitate Recovery of Distributed Intelligent Agents

Arvind K. Bansal[*], Kotagiri Ramohanarao[†], and Anand Rao[††]

[*]Department of Mathematics and Computer Science
Kent State University, Kent, OH 44242, USA
arvind@mcs.kent.edu

[†]Department of Computer Science
University of Melbourne , Parkville, Victoria 3052, Australia
rao@cs.mu.oz.au

[††]Australian Artificial Intelligence Institute
Level 6, 171 La Trobe Street, Melbourne, Victoria 3000, Australia
anand@aaii.oz.au

Abstract

We address the problem of recovering the state of an agent after a hardware/software failure of the system. We address the replication and reincarnation sub-problems of agent recovery under certain assumptions. An algorithm for distributed storage of replicated beliefs is provided and its correctness is proved formally. This algorithm allows the reincarnation of multiple crashed agents in a system of distributed autonomous intelligent agents. The scheme uses replication and distributed storage in the immediate neighboring agents, and uses distributed logical clocks to preserve the causality and to terminate retransmission.

Key-words: Distributed fault tolerance, Multi-agent system, Recovery, Reliability

1 Introduction

The past few years has seen a rapid explosion of systems built using the agent-oriented paradigm [8, 9, 15]. Increasingly, these systems are being embedded into safety-critical applications, such as, air-traffic management, telecommunications network management, and intensive-care monitoring. Agent based systems either carry out critical tasks autonomously or assist humans in critical decision-making activity in these applications. In either case, the need for robustness and a quick recovery from hardware and/or software faults becomes critical. In spite of its importance, the complexity of the task has meant that the problem has not received sufficient attention in the agent-oriented research community. This paper redresses this imbalance by examining in greater detail the problem of *agent recovery*: recovering the state of an agent after a major crash or failure.

* Address for correspondence

Although the agent recovery problem shares similarities with other related problems of database recovery and process recovery in distributed systems, there are significant differences as well. We explore some of these differences below.

First, the agents continuously sense the external environment and based on their internal mental state take certain actions that affect the environment. The effects of these external actions cannot be undone. Also, agents lack a clear notion of a transaction that can be used to log changes to its internal state onto a permanent store. In other words, rolling back to a previous state is much harder in agents than in conventional databases.

Second, the agents themselves are embedded in a continuously changing environment and often interact with other agents and/or humans to carry out a variety of tasks. As a result, even if a failed agent was capable of being recovered in a finite time, the recovered agent would still be out-of-step with the external environment, as there is a high likelihood that the external environment and the other agents have changed significantly during this period. These differences require us to modify existing techniques and adapt it for agent recovery.

Given the complexity of the agent recovery problem it is useful to split the problem into its constituent sub-problems. The agent recovery problem can be split into three simpler sub-problems:

1. the *replication* problem or the problem of replicating the state of an agent at other nodes in a distributed network, so that the agent can be rebuilt when a system failure occurs;
2. the *reincarnation* problem or the problem of building the state of a crashed agent from the replicated information across the distributed network; and finally
3. the *assimilation* problem or the problem of the agent "catching-up" with the external environment, by reaching a state which could be considered a state where it would have been, had it not crashed.

In this paper, we consider the replication and reincarnation problems. Although we do not address the assimilation problem, an application developer can write agent-oriented programs in such a way that goals which are yet to be satisfied are re-established and tried under the new conditions. Depending on the time-criticality of the application such a solution may be adequate. However, the problem of providing fault tolerant recovery in a non-stop real-time system is quite complex. We believe that the research presented in this paper will certainly help in providing a practical fault-tolerant systems.

Before we proceed with the solution for the agent recovery problem, we briefly describe system of agents under consideration, assignment of agents to machines, and *crash* or *failure* of an agent.

The agents we have in mind are Belief-Desire-Intention (BDI) agents. BDI agents [8] are autonomous software entities that sense a continuously changing environment and based on their internal mental state take certain actions which affect their external environment. The primary distinguishing feature of a BDI agent is its mental state - comprising the mental attitudes of *beliefs*, *desires*, and *intentions* that represent information, motivational, and deliberative components, respectively. In addition, these agents are programmed by writing *plans* that specify how to achieve

desires in certain situations. A practical realization of such BDI agents is dMARS (distributed Multi-Agent Reasoning System) agent-oriented development environment that has been used to build a number of practical applications [15]. The multi-agent environment dMARS allows the specification and execution of multiple BDI agents which cooperate with each other to solve a problem. The architecture allows multiple agents to be assigned to a single operating system process, and spawns multiple processes to run on a single machine.

Given a generic agent architecture which allows multiple agents to run simultaneously on a single machine, one can envisage at least three different types of crashes or system failures (see Figure 1): single agent crash, multiple non-neighbor crashes, multiple neighbor crashes.

Single agent crash implies that a reincarnated agent fully recovers before another agent crashes. After the crash of an agent, its belief and vaults (needed to store the beliefs of the neighboring agents) are lost. To facilitate the recovery, the agent's beliefs are distributed in the vaults of the neighbors and the secondary storage of the neighbors. To recover the vaults, fault tolerance and check-pointing are needed during the storage of beliefs. After the full recovery, there is no loss in the belief system.

Multiple non-neighbor crash is treated as multiple isolated single agent crashes. Since none of the crashed agent store the beliefs of other crashed agents in their vaults, each one of them can be recovered fully in a concurrent manner.

Multiple neighbors crashes suffer from the problem of information loss since the neighbors share mutually part of the each other's beliefs. To avoid information loss, belief replication has to ensure that multiple neighboring vaults are sent the same belief.

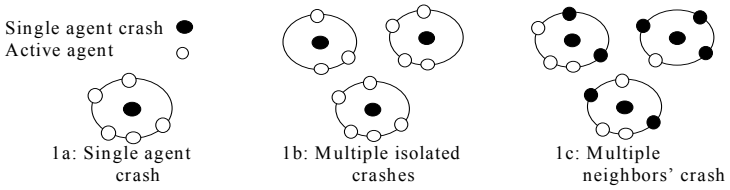


Fig. 1. Different types of crashes

In this paper, we consider single agent and multiple non-neighbor crashes. With such a crash the loss of an agent results in a temporary loss of some of the functionality of the overall system. As long as the situation does not deteriorate to a state where multiple neighboring agents are failing at the same time, the agents can be reincarnated at other nodes, and can be integrated with the rest of the system to facilitate the reliable execution of the overall system. After reincarnation, an agent has no knowledge of the tasks (or intentions) that were being executed at the time of crash, has no knowledge of the neighboring agents which initiated new tasks, and has no knowledge of the agents with whom it had to communicate. In addition, the state of agents would have changed between the time of crash and time of recovery: beliefs would have altered, and some of the tasks being executed at the time of crash

would not be needed. Under these restrictions, we examine the solutions for the problem of agent recovery.

A naive solution of periodically sending messages to neighboring agents informing them of one's own belief state will not provide the desired behavior. For example, assume that an agent A sent a message at Time 1 to its neighboring agent B that A's position is (10, 10) on an X-Y axis, and the agent A sent another message at Time 3 to the agent B that A's position is (11, 10). Depending on how the agent B processes the messages one might end-up with different results. To avoid this problem one can insist that the agent B updates based on its local clock, i.e., process the messages in the order that they arrive.

However, this solution is also inadequate since order of processing based on arrival does not preserve the order of events. Consider the situation where there are three neighboring agents A, B, and C, each sending their belief state (which includes the position of itself and other agents) to their neighbors. In such a situation it is possible that an agent, say B, receives a message from A that its new position is (11, 10) followed by another message from C that A's position is (10, 10). Processing these messages according to its local clock can result in agent B incorrectly recording that agent A's position is (10, 10).

To overcome these problems each agent needs to maintain a distributed logical clock [7] and update the beliefs based on their causal ordering, not just their local clock. In this paper, we discuss the distributed storage of replicated beliefs of a crashed agent to facilitate recovery. Beliefs are time-stamped and are replicated at the neighboring agents. In addition, to reduce the overhead of communication between agents, the time-stamped beliefs are piggybacked on existing messages. The scheme assumes that replicated beliefs are check-pointed periodically to facilitate fast recovery by the agents.

The major contributions of this paper are twofold. First, to the best of our knowledge, this is the first time that causality and replication used in the recovery of distributed transaction processing and fault tolerant computing systems [2] have been applied for the recovery of distributed intelligent agents. Compared to direct acknowledgment based schemes, this mechanism provides fault tolerance in the case of signal loss. Second, the agent recovery mechanism proposed here allows for reincarnation of an agent on any node in the network independent of the crash of a node. Third, we also use the messages at the application level for transmitting the belief updates, and use time-stamp comparison of the messages at application level to ascertain the storage of the belief updates in the neighbors.

2 Background

In this section, we describe distributed logical clocks and their application to ensure causality, and introduce the notations and definitions.

In a heterogeneous network different machines have different physical clocks. These clocks have different values at the same absolute instance. As a result communication delay causes the same event to be recorded at different times and the

causality of events can not be guaranteed. Logical clocks [5] are needed to maintain the order of distributed events. The schemes are based upon associating a logical clock - count of events - associated with each agent and transmitting a vector of logical clocks [7] among agents. The vector of clocks are piggybacked with every message from one agent to the another. Upon the receipt of a vector clock instance from another agent, the local vector clock is updated by taking the *least upper bound* of the common logical clocks from two vector-clock instances. Events are time stamped by the current value of the vector clock in an agent. Causality between the events is established by using partial order between the time-stamps. A vector clock instance precedes another vector clock instance if all the logical clocks in the first instance precede or equal the corresponding logical clocks in the second instance, and there exists at least one logical clock in the first instance which precedes the corresponding logical clock in the second instance. Two events are concurrent in the absence of partial ordering between the corresponding vector clock instances.

In this paper, we denote vector clock as a sequence of integers within angular brackets < … >. An unknown logical clock instance (or a vector clock instance) is denoted by the bottom symbol "⊥". We define two important notions: least upper bound of tuples (lub), and greatest lower bound of tuples (glb), which we use throughout this paper to update the value of vector clocks. The definition of *lub* and *glb* are as follows:

$$lub(\langle a_1, \ldots, a_n\rangle, \langle b_1, \ldots, b_n\rangle) = \langle max(a_1, b_1), \ldots, max(a_n, b_n)\rangle \quad (1)$$
$$lub(\langle a_1, \ldots, a_n\rangle, \perp) = \langle a_1, \ldots, a_n\rangle$$
$$lub(\perp, \langle a_1, \ldots, a_n\rangle) = \langle a_1, \ldots, a_n\rangle$$

$$glb(\langle a_1, \ldots, a_n\rangle, \langle b_1, \ldots, b_n\rangle) = \langle min(a_1, b_1), \ldots, min(a_n, b_n)\rangle \quad (2)$$
$$glb(\langle a_1, \ldots, a_i, \ldots, a_n\rangle, \perp) = \langle a_1, \ldots, a_i, \ldots, a_n\rangle$$
$$glb(\perp, \langle a_1, \ldots, a_i, \ldots, a_n\rangle) = \langle a_1, \ldots, a_i, \ldots, a_n\rangle$$

$$max(a, \perp) = a; \quad max(\perp, b) = b; \quad max(a, b) = a \text{ if } a > b \text{ otherwise } b \quad (3)$$

$$min(a, \perp) = a; \quad min(\perp, b) = b; \quad min(a, b) = a \text{ if } a < b \text{ otherwise } b \quad (4)$$

Example 1

Let us consider Figure 2. There are three communicating agents: A_1, A_2, and A_3. The belief updates are marked by filled circles, and the transmission of messages from a agent to another are marked by arrows. The direction of the arrows is from the originator of a message to the destination of the message. The triples inside the angular brackets denote the: I_{th} field in a vector-clock instance represents the number of preceding belief updates for the agent A_i ($0 < I < 4$). A message from the agent A_1 increments the vector clock of the agent A_2 from *<0, 1, 0>* to *<1, 1, 0>* (least upper bound of *<1, 0, 0>* and *<0, 1, 0>*); a message from the agent A_3 increments the vector clock of the agent A_2 from *<1, 1, 0>* to *<1, 1, 1>* (least upper bound of *<1, 1, 0>* and *<0, 0, 1>*); a message from the agent A_2 increment the vector

clock of the agent A_1 from *<2, 0, 0>* to *<2, 1, 1>* (least upper bound of *<2, 0, 0>* and *<1, 1, 1>*); a message from the agent A_2 increments the vector clock of the agent A_3 from *<0, 0, 2>* to *<1, 1, 2>* (least upper bound of *<0, 0, 2>* and *<1, 1, 1>*); and a message from the agent A_1 increments the vector clock of the agent A_3 from *<1, 1, 2>* to *<2, 1, 2>* (least upper bound of *<1, 1, 2>* and *<2, 1, 1>*). The partial ordering of instances of vector clock maintains the causality.

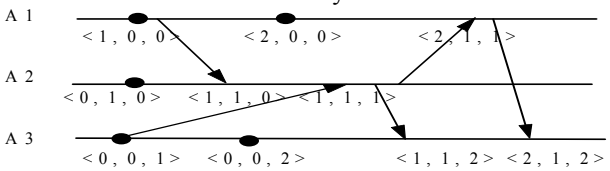


Fig. 2. Causality in logical clocks

2.1 Notations

We denote least upper bound by *lub;* greatest lower bound by *glb*; for all by $\forall$; existence by $\exists$; logical conjunction by $\wedge$, logical disjunction by $\vee$, composition of two functions by $\bullet$; a vector within a pair of angular brackets < … >; a tuple between a pair of parenthesis (…), and logical clock pair within the curly brackets { … }[1], a logical clock instance by t_i; projection of i_{th} sub-field of a tuple by Π_i; composition $\Pi_i \bullet \Pi_j$ by $\Pi_i\Pi_j$; agent-ids by italicized upper case English alphabet *A, B, C, D*; a vector clock by the capital Greek alphabet Γ, a vector clock of a generic agent D by Γ^D; an instance of a vector clock by Γ_I; a logical clock of a generic agent *A* by $\Pi_A(\Gamma)$; I_{th} element of vector clock by $\Pi_I(\Gamma)$; precedence by the symbol "$\prec$"; "precedes or equal to" by the symbol "$\preceq$"; and concurrency of two clocks by the symbol "||". We use natural language and syntax structure of C like languages to explain the algorithms.

3 Modeling Multi-agents System

In this section, we briefly describe the modeling of agents and the notion of distributed clocks in a multi-agents based system.

Although we are primarily interested in BDI agents [8, 9] the solutions proposed in this paper are generic enough to be applied for autonomous agents that communicate with each other using messages. The only requirement of these agents is that they capture the state of the external environment as a set of beliefs. We assume that agents communicate with each other using four primitive types of messages: *ask, reply, tell*, and *ack* (acknowledgment) messages. A *tell* message is unidirectional in nature; *ask* and *reply* messages are complementary; and *ack* message is a receipt of *ask, reply*, and *tell* messages. At any point in time, multiple

[1] We use curly brackets instead of pair of parenthesis just for better comprehension. Both have the same meaning.

agents are executing their programs; updating their beliefs; cooperating with each other through ask-reply message pairs, tell messages, and ack messages; and interacting with the environment through sensors to collect information and through actuators to control the environment.

The message connectivity between agents is modeled by an agent connectivity graph (A-graph). An A-graph is a weighted graph[2]: Agents are represented as nodes, and the possibility of a message transfer from one agent to the other is represented by an edge. The weight of an edge is the number of occurrences of message commands between two agents. The presence of an implicit ack message for every ask, reply, and tell implies a symmetrical edge A_iA_j for every edge A_jA_i. The total weight of an edge is given by (*occurrences of tell-messages from* A_i *to* A_j + *occurrences of ask-messages from* A_i *to* A_j + *occurrences of reply messages from* A_i *to* A_j + *occurrences of tell-messages from* A_j *to* A_i + *occurrences of ask messages from* A_j *to* A_i + *occurrences of reply messages from* A_j *to* A_i). An agent is a *neighbor* of another agent if there is an edge between the corresponding nodes.

In an agent based system, a *logical clock* is an incremental count of events within an agent. An event is either a *belief update event* or a *message event*. In a *belief update event*, a belief is updated, i.e., added, deleted, or modified. A *message event* is either an ask, tell, reply, or ack message. A *vector clock* Γ is of the form $<\gamma_1, \ldots, \gamma_i \ldots, \gamma_n>$ where each γ_i is a pair of the form (*agent-id, logical-clock*). Each logical-clock is a pair of the form {*belief update count*, *message count*}. *A belief update count* is incremented after each assertion, deletion, or modification of a belief in an agent. A *message count* is incremented before each message is sent to a neighboring agent. We assume that events in the same agent are ordered resulting into monotonicity in logical clocks. We separate the belief update count from the message count as they are handled differently at the time of recovery.

We denote the set of logical clocks that have been altered between *two consecutive instances* of a vector clock by Δ. We denote the agent-id of the I_{th} element in a vector clock instance Γ by $\Pi_I^1(\Gamma)$ and the I_{th} logical clock by $\Pi_I^2(\Gamma)$. From this we denote the belief update count of the logical clock of an agent I by $\Pi^b\Pi_I^2(\Gamma)$ and the message count of the logical clock of an agent I by $\Pi^m\Pi_I^2(\Gamma)$. We denote belief update precedence by the symbol $\prec_b$, and message precedence by the symbol $\prec_m$, "belief count precedes or equals to" by the symbol $\preceq_b$ and "message count precedes or equals to" by the symbol "$\preceq_m$".

There are two types of precedence relations between instances of the same logical clock: belief precedence and message precedence. A *belief precedence* is based on partial order between the belief update counts and *message precedence* is based on partial order between message counts.

---

[2] Although, we do not make use of these weights in this paper, they are useful in providing a probabilistic interpretation of agent failure. This is beyond the scope of this paper.

Given two instances t_i and t_j of the same logical clock, $t_i \prec_b t_j$ if $\Pi^b(t_i) < \Pi^b(t_j)$, and $t_i \prec_m t_j$ if $\Pi^m(t_i) < \Pi^m(t_j)$. For example, *{1, 3}* $\prec_b$ *{2, 3}* and *{1, 3}* $\prec_m$ *{1, 4}*. An instance of a logical clock precedes another instance of the same clock if $t_i \prec_b t_j \wedge t_i \preceq_m t_j$ or $t_i \prec_m t_j \wedge t_i \preceq_b t_j$. Due to the monotonicity property in logical clocks, it is impossible to have two instances t_i and t_j such that $t_i \prec_b t_j$ and $t_j \prec_m t_i$.

Given two instances Γ_i and Γ_j ($i \neq j$) of a vector clock, $\Gamma_i \prec \Gamma_j$ if $\exists k (\Pi_k^2(\Gamma_i) \prec \Pi_k^2(\Gamma_j)) \wedge \forall l(\Pi_l^2(\Gamma_i) \preceq \Pi_l^2(\Gamma_j))$. For example, consider an agent graph where an agent A is connected to another agent B, and the agent B is connected to the agents A and C. The vector clock has three fields: one field for every logical clock. For example, <(*A*, *{1, 1}*), (*B*, *{2, 4}*), (*C*, *{2, 5}*)> $\prec$ <(*A*, *{2, 2}*), (*B*, *{2, 4}*), (*C*, *{2, 5}*)> since *{1, 1}* $\prec$ *{2, 2}* in the agent A, and other two logical clocks in the agents B and C are equal.

Two events are concurrent if the above conditions are not satisfied. For example, <(*A*, *{4, 6}*), (*B*, *{3, 4}*), (*C*, *{1, 5}*), (*E*, *{4, 3}*)> $\|$ <(*A*, *{3, 5}*), (*C*, *{6, 7}*), (*D*, *{4, 5}*), (*E*, *{10*, *3}*)> since the logical clock instances {4, 6} (in the first vector) $\succ$ {3, 5} (in the second vector) for the agent A and {1, 5} (in the first vector) $\prec$ {6, 7} (in the second vector) for the agent C. This situation violates the condition that all the logical clock instances of a vector-clock precede or equal ($\preceq$) the corresponding logical clocks of the other vector-clock. Similarly, <(*A*, *{4, 6}*), (*B* *{3, 4}*), (*C*, *{6, 7}*), (*E*, *{4, 3}*)> $\|$ <(*A*, *{4, 6}*), (*C*, *{6, 7}*), (*D*, *{4, 5}*), (*F*, *{10*, *3}*)> since no logical clock instance in either of the vector clocks strictly precedes the corresponding logical clock instance in the other vector clock.

4 Distributed Storage and Recovery

In this section, we describe a scheme to distribute the replicated beliefs in the neighbors, using causality to control the transmission of replicated beliefs. We also describe an algorithm for the transmission of replicated beliefs to neighbors.

4.1 Distributed Storage of Replicated Beliefs

Each agent holds a secure vault to store the transmitted beliefs for each of its neighbors. These vaults are updated dynamically after a message from the corresponding neighbor is received. A *neighboring vault* is a set of 5-tuples of the form (*agent-id of origin, birth clock, old-belief, nature of update, new-belief*) where *agent-id* identifies the neighboring agent requesting the update; *birth clock* is an instance (of origination) of the vector clock of the originating agent; and nature of update specifies the action to take. An update could be *assert*, *retract*, or *modify*. The vault in an agent is periodically check-pointed. We require that the vault be check-pointed immediately after a recovery. The advantage of check pointing is that number of belief updates are reduced during recovery since check-pointing retains the latest belief updates.

4.2 A Brief Overview of the Belief Recovery Process

If the secondary storage used by the crashed agent is reachable during the recovery process to reincarnate an agent, then the check-pointed beliefs and the check-pointed vaults related to the neighbors are restored; the check pointed replicated beliefs of the crashed agent in the neighbors' secondary storage are restored; and the remaining subset of beliefs is restored from the distributed vaults. In the absence of secondary storage used by the crashed agent, the problem becomes complex since check pointed beliefs and the vaults in the crashed agent are lost. The use of fault tolerance to ensure that multiple neighbors get the same replicated beliefs in their vaults is beyond the scope of this paper.

4.3 Belief Replication and Transmission

A replicated belief update is stored in at least one of the vaults in the neighboring agents. Belief updates are time-stamped and retransmitted through the following messages to neighboring agents. Retransmission mechanism uses piggybacking on one of the implicit messages - *ask, reply, tell*, or *ack* - until piggybacked clock instance on one of the received messages ensures that the replicated beliefs have been received by one of the neighbors. The sending agent increments its message-count before sending the message; incrementing the clock and sending the message constitute an atomic action. Each agent keeps track of the last clock transmitted to the neighbor and the last clock received from each of its neighbors. It sends Δ to reduce the communication overhead [7]. The overall clock instance is re-built at the destination. The neighboring agent, receives the updated belief and stores the transmitted belief updates. The agent then increments its vector clock. We require that storing the belief and updating the vector clock constitute an atomic action. The value of a vector-clock instance and the notion of causality is used to control the retransmission of replicated beliefs as described in the following section.

4.4 Belief Storage and Acknowledgment

The belief updates are sent by any of the four types of messages: ask, reply, tell and ack which results in sixteen pair combinations for sending the belief updates and receiving back a vector-clock instance acknowledging the receipt of the update in the corresponding neighbors. However, the edges in the A-graph limit the use of different combinations. There are two scenarios for the transmission of belief updates and the acknowledgment that it has been received by one of the neighbors:

1. The agent A_i's neighbor A_j ($i \neq j$) which received the belief update sends one of the four messages to the agent A_i, and the agent A_i receives the message before receiving any other agent's message which received a message from A_j after A_j received A_i's message.

2. The agent A_i's neighbor A_j ($i \neq j$), received the belief update, and sends a message to a neighbor A_k ($i \neq k, j \neq k$). Before A_i receives a message from A_j, A_i receives a message which succeeds (by causality) the message from A_j to A_k.

The update in the neighboring vault is ascertained by comparing the received clock instance with a *last update acknowledged marker* (*LUA marker).* A *LUA marker* is an instance of an agent's vector clock which marks the birth of the last belief update which has been stored in one of the neighbors' vault. Every agent has its own LUA marker.

4.5 An Algorithm for Retransmission of Belief Updates

As shown in Figure 3, a queue of 5-tuples (*birth-time, agent-id, old-belief, update-type, new belief*) is maintained in each agent to retransmit the belief updates. At any time, all the beliefs updated between the interval of the current LUA marker and the current value of the agent's vector clock are transmitted to one of the neighbors (see Figure 3). A LUA marker is updated to received-clock instance if LUA marker $\prec$ received-clock instance, and all the messages with birth-time $\preceq$ the new LUA marker are removed from the queue.

Example 2

Let us consider a strongly connected A-graph with three agents *A, B* and *C*. Consider a case in the agent A when the P_A^2(LUA marker) of the agent A = {*1, 1*}. After the assertion of each belief the agent A increments its belief update count by *1*. The boxes in the queue in Figure 3 show the belief updates stored in the outgoing queue, and the pairs in the bottom line of Figures 3a and 3b and 3c give P_A^2(birth-time) associated with each belief update. Let us assume that the messages are transmitted to the agent *B* when the logical clock instance for the agent A is {*4, 1*}. Before transmission the message count of the logical clock is incremented by 1, and the logical clock instance of the agent A gets updated to {*4, 2*}, and the belief-updates (in the outgoing queue of the agent A) with P_A^2(birth time) {*2, 1*}, {*3, 1*}, {*4, 1*} are transmitted to *B*. The next re-transmission occurs when the logical clock instance is {*5, 2*}, and the agent A sends a message to the agent C. Before re-transmission the message-count is incremented by *1*, the logical clock instance for the agent A at the time of re-transmission becomes {*5, 3*}, and the belief updates with P_A^2(birth time) {*2, 1*}, {*3, 1*}, {*4, 1*}, {*5, 2*} are transmitted to the agent *C*. Note that the belief updates with P_A^2(birth time) {*2, 1*} and {*3, 1*} and {*4, 1*} are being re-transmitted to one of the neighbors. Figure 3a shows a snapshot of this instance.

Figure 3b shows a snapshot after the belief updates with P_A^2(birth-time) {*2, 1*}, {*3, 1*}, {*4, 1*}, {*5, 2*} have been transmitted to the agent C, and the current logical clock has been updated to {*7, 3*} after two more belief updates. Figure 3b also depicts a snapshot when an incoming tell-message is received from the agent B. The Δ (received-clock instance) sent from B to A is <(A, {*4, 2*}), (B, {*3, 2*})>. Upon

receipt of the message, the LUA marker of the Agent A has been updated to <(A, {*4, 2*}), (B, {3, 2})>, and the belief updates (in the agent A) with P_A^2(birth time) {*2, 1*}, {*3, 1*}, {*4, 1*} have been deleted from the outgoing messages queue. The snapshot after the deletion of the messages is shown in Figure 3c.

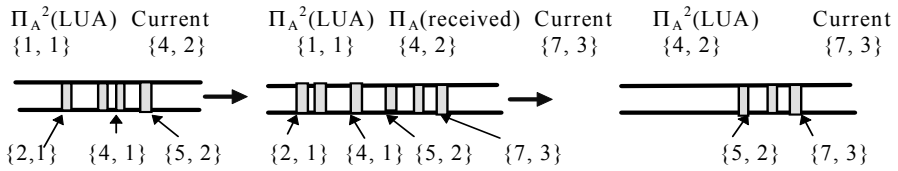


Fig. 3. Belief re-transmission and LUA marker update

From the above description it should be reasonably straightforward to arrive at an algorithm for retransmitting updated beliefs. The algorithm is given in Figure 4.

Algorithm Belief-retransmission-and-LUA-update;

Input: 1. A queue Q of replicated beliefs in an agent *A*;
2. A vector-clock instance C_i for an agent *A*;
3. The LUA marker for an agent *A*;

Output: 1. An updated queue Q of the replicated beliefs;
2. The vector clock instance C_{i+1} for the agent *A*;
3. The modified LUA marker for the agent A;

{1.Receive the next message. Build the full received-clock instance from Δ;
2.If ((Π_A^2(LUA-marker) $\prec_m$ Π_A^2(received-clock instance)) &&
(LUA-marker $\prec$ received-clock instance)) {
3. LUA-marker = received-clock instance;
4. Delete the beliefs from Q with birth-time $\prec$ LUA-marker;}
5. Agent's vector-clock C_{i+1} = *lub*(received-clock instance, C_i);}

Fig. 4. An algorithm for retransmission

To prove the correctness of the algorithm, namely, that the belief updates of an agent A will be stored in at least one of its neighbors we need to prove the following theorem.

Theorem: The transmitted beliefs between the interval LUA marker and the logical clock preceding the Π_A^2(received clock instance) of an agent *A* are stored in at least one of the neighbors iff (Π_A^2(*LUA marker*) $\prec_m$ Π_A^2(*received-clock instance*)) $\wedge$ (*LUA marker* $\prec$ *received-clock instance*).

Proof: Let us consider an A-graph, without loss of generality, which contains three agents A, B, and C such that the agents A and B are neighbors, the agents B and C

are neighbors, and there is a path from the agent C to the agent A other than through the agent B.

We first prove that if the vault update is done in a neighbor *B* of an agent *A* then the condition is true. After the vault update, the neighbor *B* updates its vector clock Γ^B to new value *lub*(current value of Γ^B, clock instance received from *A*). The neighbor *B* sends at least one message. The $\Pi^m\Pi_B{}^2(\Gamma^B)$ is incremented by *1*. If the message is directly sent back to the agent *A* then there is no problem since the logical clock of *B* has been updated and incremented such that LUA marker of A $\prec$ vector clock instance sent by A $\prec$ value of the current instance of Γ^B. If the message is sent from *B* to another agent *C* ($C \neq A$) and there is a path from C to *A* in the A-graph then all the agents in the path will at least increment the message-count in their logical clocks by at least one before *A* gets the acknowledgment.

We now prove that if the condition is true then vault update is done. The proof is by contradiction. Let us assume that there is a case ($\Pi_A{}^2$(*LUA marker*) $\prec_m$ $\Pi_A{}^2$(*received-clock instance*)) $\wedge$ (*LUA marker* $\preceq$ *received-clock instance*) but the corresponding beliefs with the belief update count $\Pi^b\Pi_A{}^2$(*birth-time*) $\leq$ $\Pi^b\Pi_A{}^2$(*received-clock instance*) have not been stored in any of the neighboring vaults. The logical-clock instance $\Pi_A{}^2$(*received-clock instance*) can only be incremented by *A*. Thus, at least one of the neighbor received a later value of the logical clock of *A*. A later instance of logical clock of *A* is only piggybacked in the messages sent after the *LUA* marker. Thus at least one of the neighbors got the message with the time-stamp $\Pi^m\Pi_A{}^2$(*received-clock instance*). Due to monotonic property in the logical clock instances, if $\Pi_A{}^2$(*birth-time*) $\prec_m$ $\Pi_A{}^2$(*received-clock instance*) then $\Pi_A{}^2$(*birth-time*) $\preceq_b$ $\Pi_A{}^2$(*received-clock instance*). Hence, all the beliefs with the belief update count $\Pi^b\Pi_A{}^2$(*birth-time*) $\leq$ $\Pi^b\Pi_A{}^2$(*received-clock*) have been received by one of the neighboring agents. The operation of receiving the message and storing the message in the vault are atomic. Hence the update has been done. A contradiction.∎ QED

Complexity: Let us say that the size of the message queue is M, the number of neighboring agents are upper bounded by N. Thus the clock-size of each agent is bounded by N. The overall complexity is guided by the binary-search of an N-tuple clock in a message queue of size M, and to delete all messages before the LUA marker. The overall complexity is O(NlogM + M). We delete the details due to the space limitation.

4.6 Complexity of Handling Multiple Neighbors Crash

Failure from a multiple neighbors crash is a dynamic property. and a probabilistic model can be used to estimate the overall loss of information for an agent in a multiple neighbor crash scenario. The estimated loss is:

$$\sum_{m=0}^{m=r} (m/k) \binom{K}{m} \Big/ \binom{n-1}{m} \times p^{(m+1)} (1-p)^{n-m-1}$$

where m is the number of failure, k is the average number of neighbors of an agent, r is the maximum number of failed agents at a time, n is the number of agents in the system, and p is the probability of failure of an agent.

It can be shown in the probabilistic model that providing fault tolerance by re-transmitting extra number of messages to different neighbors can alleviate the problem of multiple neighbor crashes. The upper bound on the number of re-transmissions to different neighbors can be derived for a specific failure rate. The need for providing fault tolerance increases with the failure rate. The need for extra fault tolerance increases the re-transmission and message logging overheads on the agents. However, the probabilistic model is outside the scope of this paper.

5 Related Works

The work on distributed fault tolerant computing, replication of facts for fault tolerant distributed transaction processing, and the use of vector of distributed logical clocks has been well researched [1, 2, 6, 7, 13]. However, to the best of our knowledge, this is the first attempt to apply distributed logical clocks to ensure reliability in intelligent agents. Indeed, this paper uses the results from distributed computing and database recovery. However, the problem is more complex in agent-based systems as the agents are autonomous and interact continuously (in a non-stop fashion) with a changing environment.

Our scheme of storing the replicated beliefs in immediate neighbors keeps the overhead low which is necessary to satisfy a realistic time interaction of humans with agent based systems. The proposed scheme does not use any additional message (other than *implicit ask, reply, tell*, and *ack* messages) between the agents, and make use of vector clock comparison to ascertain the belief-updates.

Specifically, this work is different from standard distributed operating and database system due to its use of application level messages: *ask, reply, and tell* messages instead of low level acknowledgments. Unlike other schemes, we use clock comparisons for the acknowledgment of messages; and message can come from any neighbor if clocks satisfy causality.

Related work is also being done in the application of group communications to the transaction level processing [10]. Our scheme of establishing reliability by storing the beliefs in neighbors is different from the scheme of establishing reliability of group communications. Group communication is at a lower level while our scheme is at the application level. We use neighborhood as a group and reincarnate crashed agents to provide reliability, while group communication approaches use majority scheme [2, 10] and group based integrity to provide fault tolerance. Group communication schemes have additional communication overhead due to the distribution of the processes in the same groups on different nodes.

Other work to provide message acknowledgment [14] suffer from the problem of synchronization and loss of acknowledgment due to loss in communication. Our scheme is free from any problem of synchronization and data loss since the acknowledgment is based upon clock comparison. Such a clock

comparison provides implicit fault tolerance against any data loss during communication. However, we believe that reliability will benefit from fault tolerance and group communication at lower level [10].

The work on social comparison of agents [3] uses comparison of trace of events of an agent with the trace of similar agents. This scheme is suitable to correct deviations when there are multiple similar agents, and does not address the issue of recovery when the agents crash.

The work on customizable coordination system [12] provides a high level declarative specification of interaction between agents. It will be very interesting to integrate the low level messages generated in this scheme with our scheme of distributed logical clock comparison to provide application level fault tolerance.

6 Conclusion and Future Work

In this paper, we have described a scheme for reliable intelligent distributed agents. The scheme benefits from previous research on fault tolerance distributed computing and recovery in distributed transaction processing, and the use of causality based on vector of logical clocks. The scheme uses partial ordering of clocks, comparison of vector clocks for the acknowledgment of update, piggybacking on existing messages, and storage of replicated beliefs in immediate neighbors to reduce the overhead of communication during storage and recovery.

The assumption of mapping one agent per processor will be relaxed in future. The notion of uniform connectivity will be relaxed using static and dynamic analysis of the behavior of agents. We aim to address the problem of non-neighborhood crashes, provide a probabilistic interpretation of failure, and enhance the scheme to address dynamic message transfer behavior between agents.

Acknowledgments

This first author was supported in part by the Australian Federal Government funded program of Cooperative Research Center for Intelligent Decision Systems during his sabbatical to The University of Melbourne. The authors acknowledge Michael Georgeff, Andrew Worsley, and Andrew Hodgson at the Australian Artificial Intelligence Institute for useful discussions.

References

[1] D. Agarwal and A. Malpani, “Efficient Dissemination of Information in Computer Networks,” *The Computer Journal*, 34:6, 1991, pp. 534 - 541.

[2] P. Jalote, “Fault Tolerant Distributed Computing,” *Prentice Hall*, 1993.

[3] G. Kalinka and M. Tambe, "Social Comparison for Failure Detection and Recovery," In this volume.

[4] D. Kinny, M. Georgeff, J. Bailey, D. B. Kemp, and K. Rammohanarao, "Active Databases and Agent Systems," *Proceedings of the Second International Rules in Database Systems Workshop, RIDS95,* Athens Greece, 1995.

[5] L. Lamport, "Time, Clock, and the ordering of Events in a Distributed Systems," *Communications of the ACM,* 21:7, 1978, pp. 558 - 565.

[6] H. V. Leong and D. Agrawal, "Using Message Semantics to Reduce Rollback in Optimistic Message Logging Recovery Schemes," *Proceedings of the 14th International Conference on Distributed Computing Systems*, 1995

[7] M. Ranyal and M. Singhal, "Capturing Causality in Distributed Systems," *Communications of the ACM,* February 1996, pp. 49 - 56.

[8] A. S. Rao and M. P. Georgeff, "Modeling Rational Agents Within a BDI-Architecture," *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, San Mateo, CA, USA, Morgan Kaufaman publishers, 1991.

[9] A. S. Rao, "AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language," in *Agents Breaking Away*, editors, Van de Velde, W. and Perram, J. W. Lecture Notes in Artificial Intelligence, LNAI 1038, Springer-Verlag, 1996

[10] A. Scheiper and M. Ranyal, "From Group Communications to Transactions in Distributed Systems," *Communications of the ACM,* 39:4, 1996, pp. 84 - 87.

[11] F. B. Schneider, "Implementing Fault Tolerant Services using the State Machine Approach, a tutorial," *ACM Computing Surveys* 22: 4, 1990, pp. 299-319.

[12] M. P. Singh, "A Customizable Coordination Service for Autonomous Agents," In this volume.

[13] J. Wuu and A. J. Bernstein, "Efficient Solutions to the Replicated Log and Dictionary Problems," *Proceedings of the 3rd ACM Symposium of Principles of Distributed Computing*, ACM Press, New York, 1984, pp. 233 - 242.

[14] A. R. Worsely and A. Hodgson, "dMARS Fault Tolerant Communications, Reliable Messaging Use Cases," *Internal Report*, The Australian AI Institute, Carlton, Victoria 3053, Australia, February 1995.

[15] M. Wooldridge and N. R. Jennings, " Intelligent Agents: Theory and Practice," *The Knowledge Engineering*, Publisher: Springer Verlag, Volume 890, 1995