Incorporating Fault Tolerance in Distributed Agent Based Systems by Simulating Bio-computing Model of Stress Pathways

Arvind K. Bansal
Department of Computer Science
Kent State University, Kent, OH 44242, USA
E-mail: arvind@cs.kent.edu

ABSTRACT

Bio-computing model of 'Distributed Multiple Intelligent Agents Systems' (BDMIAS) models agents as genes, a cooperating group of agents as operons ─ commonly regulated groups of genes, and the complex task as a set of interacting pathways such that the pathways involve multiple cooperating operons. The agents (or groups of agents) interact with each other using message passing and pattern based bindings that may reconfigure agent's function temporarily. In this paper, a technique has been described for incorporating fault tolerance in BDMIAS. The scheme is based upon simulating BDMIAS, exploiting the modeling of biological stress pathways, integration of fault avoidance, and distributed fault recovery of the crashed agents. Stress pathways are latent pathways in biological system that gets triggered very quickly, regulate the complex biological system by temporarily regulating or inactivating the undesirable pathways, and are essential to avoid catastrophic failures. Pattern based interaction between messages and agents allow multiple agents to react concurrently in response to single condition change represented by a message broadcast. The fault avoidance exploits the integration of the intelligent processing rate control using message based loop feedback and temporary reconfiguration that alters the data flow between functional modules within an agent, and may alter. The fault recovery exploits the concept of semi passive shadow agents – one on the local machine and other on the remote machine, dynamic polling of machines, logically time stamped messages to avoid message losses, and distributed archiving of volatile part of agent state on distributed machines. Various algorithms have been described.

1. INTRODUCTION

With the growing popularity and complexity of Internet based systems, the need for autonomous software has become essential. An autonomous software has functionality and agenda, and reacts to environmental conditions. A set of cooperating distributed agents may reconfigure dynamically based upon external environment and get activated based upon external messages or environment. With the increase in the complexity of agent based software systems and the presence of extensive knowledge based resources over the Internet, the need for 'Distributed Multi Intelligent Agent Based Systems' to assist humans has increased significantly. However, the use of such systems is still limited due to the use of (1) point to point messaging (2) limitations in the dynamic adaptation of agent functionality, and (3) the lack of fault avoidance, (4) the lack of robust schemes of fault tolerance and recovery in multi agent systems. In a distributed system there could be multiple types of faults such as process failure, node failure, communication failure, unexpected communication delay, agent database corruption, agent functionality corruption, agent interaction group corruption, intentional introduction of malignant agents. Many of these failures have a precursor such as process overload, communication overload, or frequent link failures between two nodes.

In contrast biological systems[(1)] provide robustness by integrating multiple level fault avoidance and fault recovery at the same time. At the genomic level, the genome provides fault tolerance at multiple levels such as double complementary strand for continuous gene-repair, evolution to create similar but more robust strains, the use of stress pathways to stabilize pathways, promoter based activation and inactivation of genes, and protein-DNA interaction to change the gene configuration. Biological systems continuously monitor the system parameters: any change in the concentration of a type of message starts a counter mechanism to balance the system. Any imbalance triggers stress pathways that identifies the fault causing agents, and takes counter measures to correct it. For example, immune system

Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense V, edited by Edward M. Carapezza, Proc. of SPIE Vol. 6201, 620108, (2006)
0277-786X/06/$15 · doi: 10.1117/12.669505

identifies and tags the foreign bodies by binding, and sending proteins as messages to trigger pathways to kill and clean up the tagged bodies. Those proteins bind to proteins (or gene promoters) altering their configuration resulting into change in functionality.

There are five major aspects of fault tolerance: authenticating messages, continuous agent code repair, fault avoidance, graceful degradation, and fault recovery. Most of the research in the past six years is focused on fault recovery. In the recent years, multiple attempts of incorporating fault recovery in distributed agent based systems and mobile agent based systems have been developed. The schemes can be classified as active agent replication based [13, 14], exception handling, agent recovery using integration of check-pointing and distributed replication of beliefs and change of state between two check-points [4], and adaptation based high level heuristics [10]. Active replication based systems keep a hot replicated agent that substitutes in the case of the corresponding agent failure until the agent is recovered again. While process replication and check pointing and distributed memory replication can handle process recovery at the cost of computation and memory overhead, little attempts have been done to avoid failures caused by increased overload. Many agent crashes can be avoided by timely sensing the overload problem, and triggering exception handlers that can either reduce the overload or temporarily shutdown that part of the agent system until the cause of overload has been removed.

The focus of work in this paper is in the development of an integrated approach — an approach that integrates load balancing by changing the data processing rate, process migration for load balancing, hot replicated local and heterogeneous agents, pattern matching to select the latent stress pathway activation, and finite change in configuration of agent based upon message binding — to handle failures of multi-agents caused by process failure, node crashes and failure avoidance.

This paper extends the author's bio-computing model [5] by incorporating fault tolerance using stress pathways and code repair. In the proposed bio-computing model, a bio-computing agent lives in one of the finite configurations that are selected by the messages. Configuration decides the dataflow between various function modules within an agent including partial activation (or varying the processing power) of a subset of function modules within an agent. In addition bio-computing agents use pattern matching of the broadcasted copies of the same message to allow multiple agents to be triggered simultaneously.

Stress pathways are exception handlers that are latent under normal condition. However, stress pathways are triggered once the multi-agent system goes beyond a threshold of normal activity to avoid instability. This abnormal system behavior may be caused either by changed external environment such as communication link failure, or sudden increase in data processing overhead, computation overhead, and process failure. The use of stress pathways is very different from slow statistical learning based adaptation models such as 'Ant Colony' [8] due to (1) the fast switching activation of stress pathways, (2) switching between finite number of configurations, (3) pattern based matching that allows the messages to be broadcasted and allow multiple agents to be activated concurrently.

The overall paper is organized as follows. Section 2 describes the background of agent replication and belief replication, the notion of heterogeneous partitions, and new definitions related to overloads and various schemes of fault recovery used in Multi Agent Systems (MAS). Section 3 describes the Bio-computing model. Section 4 describes the various schemes of fault tolerance. Section 5 describes algorithms for dynamic failure detection. Section 6 describes the related works, and the last section concludes the paper.

2. BACKGROUND AND DEFINITIONS

An *adaptive agent* is a reactive autonomous software system that can adapt to the external conditions or unfavorable internal changes by selectively varying its functionality: repressed, enhanced, suspended temporarily or permanently in response to a valid message or internal computation or sensing of an external condition or unfavorable internal changes such as buffer build up. A transmitted message is maintained in the queue until an acknowledgement (direct or indirect) has been received that indicates the receipt of the message at the destination or intermediate nodes that can transmit the message to the destination(s) in finite time directly or indirectly. A *distributed logical clock* [10] is associated with every message to maintain the sequentiality of events and *exactly once execution property* in response to a message. A

distributed-clock $\Pi = <\pi_1, \pi_2 \ldots \pi_N>$ (is a vector of the logical-clocks π_I ($1 \le I \le N$) of the agents where each π_I is a pair of the form (*belief-update-count, message-count*) [4]). A *belief-update-count* is incremented if the agent asserts a new assertion, and the *message-count* is incremented when the agent sends a new message. A distributed clock Π_1 is less than another distributed clock if every π_I ($1 \le I \le N$) in $\Pi_1 \le$ the corresponding clock in Π_2, and there exists at least one π_J in Π_1 that is strictly less than the corresponding logical clock in Π_2.

An agent can crash either due to process failure, indefinite delay caused by overload or node crash. Node crash results in the failure of all the executing agents. After a node crash, all the active agents on that node are lost, and need to be recovered. An agent recovery includes the recovery of unprocessed input messages and unacknowledged outgoing messages. *Replication* is the process to create a copy of the agent state for the purpose of recovering agent. Replication can be done within the same node, or on a remote node. The advantage of replication on the same node is that failure detection and recovery is faster. The disadvantage of replication on the same node is that in the event of a node crash, the replica crashes along with the agent. A *shadow agent* is a semi active (or semi-passive) replica that gets all the updates of the agent. A *local shadow agent* is located on the same computer node as the active agent. A *remote shadow agent* is a shadow agent that is located on a node other than the local node. The state of the shadow agent is maintained by the corresponding monitor of an active agent using messages tagged with distributed logical clocks.

Check pointing and agent replication based systems need roll back or pessimistic commit mechanism to ensure *execute-once* property of agents. *Execute-once property* is quite important in case of agent replications as an action executed more than once (first by the original agent and then by the replicated agent) will change the overall state of the multi agent system.

A *partition* is a set of nodes in a network that is strongly connected that is there is a path between any two nodes in the partition. Two partitions are heterogeneous if there is no common edge between them. A network is a union of all such heterogeneous partitions.

Detection of an agent failure needs the presence of monitoring agents that monitor the worker agents, and worker agents are registered with them. In the presently implemented systems, monitor agents reside on the same node thus having access to the archive of beliefs, plans, and state of the worker's agent. Upon the failure of worker agents, monitor agent communicate the agent's state to the shadow agents or other monitor agents.

3. BIOCOMPUTING MODEL OF INTELLIGENT AGENTS

Bio-computing model of multi agent based systems [5] model systems as set of interacting genes and the dataflow modeled as a set of interacting pathways. The multi agent system is organized at five levels: domain, bio-agent, message, bio-agent-team, pathway, and monitor. The domains, genes, and pathways can be in three stages: latent, suspended, and active. Suspension (temporary or permanent) may be caused periodically or may be based upon an event conveyed through a message.

A *domain* is the smallest unit of function in a bio-agent. A domain is modeled as a string, and acts as a key to invoke a function when the corresponding domain is activated. A *bio-agent* is a triple of the form (*promoter, set of domains, domain interactivity graph*) where a promoter is a composite binding pattern that is used for the interaction of an external or a local message with the bio-agent, and controls the activity of one or more domain(s) and dataflow by modifying the interactivity graph. A bio-agent can live in finite multiple configurations decided by the interactivity graph. The current path from the input domain to the output domain is given by the sequence of edges that give the maximum cumulative weight. By dynamically changing the weight of interactivity graph the data flow between the agents can be altered. The interactivity graph can be changed by external messages, and by internal programs those are triggered periodically and in the response to external messages [5].

A *promoter* is a quadruple of the form (composite binding pattern, action, delay time, reset time). The composite binding pattern is modeled as a concatenation of multiple binding patterns where each binding pattern controls the corresponding domain activity. A message is a triple of the form (*composite binding pattern, actions*, and *domains*). A message is passive despite having domain definitions. A message interacts when its binding pattern matches with the

binding patterns of different domains of a bio-agent. The matching is done by checking complement of the binding pattern of the message with the binding pattern of the promoter region of the bio-agent. Action on the corresponding domain is taken if the match is above a certain threshold. The message binding score is a combination of amount of matching binding, and the priority of the message. In the presence of multiple competing messages, the message with maximum binding score decides the action. There are many actions provided by messages such as domain suspension, *enhance data processing rate, decrease data processing rate, and the integration of message domains that may lead to a new configuration matrix altering the data flow*. A message may contain one or more actions.

Every domain updates the blackboard stored in the monitor [5]. A group of genes form an operon. Each operon has one input and one output, and agents within an operon are involved cooperatively in a single overall subtask. The genes within the same operon communicate to each other using the blackboard located in the monitor. *Monitors* are the meta-agents created at the time of initialization, and are not part of the agents' data processing system. The monitor within the operon is entrusted with the communication within the operon, communication between different operons in the same pathways, and the operons across different pathways. The basic tasks of the monitors is to (1) keep the registry of the created agents, (2) keep track of messages transmitted from the agents (3) sending messages to agent's replica in the same node and in the remote node (4) keeping track of liveliness of the neighboring operon's monitor, (5) distribution of the agents in case of the agent or node failure, (6) creation of the replica agents, (7) keeping the blackboard where agents within an operon communicate, and (8) managing I/O. Two different nodes communicate using a limited number of communication agents or through hot links between the top level monitors. The advantage of having the communication agent is that the number of redundant transmitted messages between agent-to-agent communications is reduced avoiding communication overhead. Monitors and regulatory pathways are for the meta-control and fault tolerance of the pathways. The overall structure of the bio-agent based model is given in Fig 1. The leaf nodes represent the bio-agents involved in data processing, and the non-leaf nodes are the monitor nodes.

4. FAULT TOLERANCE IN BIOCOMPUTING MODEL

Fault tolerance in the distributed bio-computing model is provided at multiple levels. The model supports both fault avoidance as well as fault recovery.

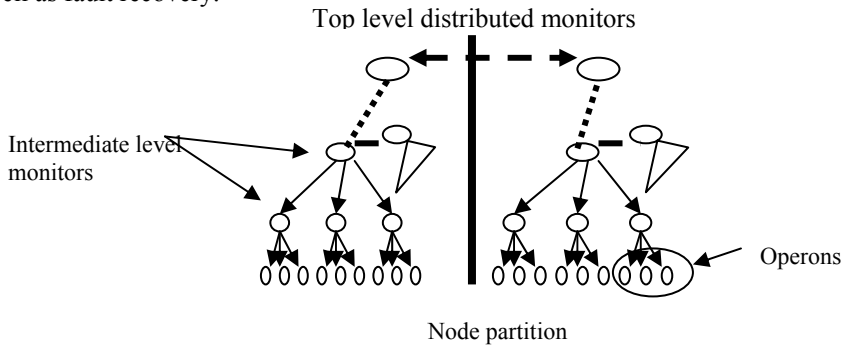

Figure 1. The distributed bio-agent system

Pathways are mapped initially statically and later dynamically to balance cumulatively computational and transmission load. During fault recovery, multiple pathways are migrated and distributed to other active processors based on load balancing. Fault avoidance is done by monitoring and maintaining the rate of data processing and the queue length at various operons and by activating stress pathways if (1) a particular pathway is considered more important at a particular time either by external message or (2) if there is a build up in the queue. Failure detection is achieved using hierarchical structure of monitors such that the lowest level monitor monitors each operon. Higher level monitor monitors children monitors, and the root monitors have hot links to communicate to root level monitors on other nodes. Fault recovery is done by a variant of semi passive replicated shadow agent and by distributing the agent state on multiple neighboring agents using piggybacking on outgoing messages to avoid loss of state during node failures. The details and mechanism are explained in the following subsections.

4.1 Genomic level code protection

The genomic description of agent mimics complementary strands of the genes. The use of complementary strand provides genome repair. The complimentary strand has been modeled by the use of four digit numbers {*1001, 1100, 0011*, and *0110*} that model the four nucleotides {*A, C, G, T*}. The numbers {*1001, 0110*} are pair-wise complementary to each other, and are stored in the same byte as *10010110* to model the base pair A←→ T. Similarly the byte *11000011* models the base-pair G ←→ C. Each symbol is at least three hamming distance providing single bit error correction capability at high level. The genome coding has three copies of the code at the domain level making to ensure code correction by consensus.

4.2 Mapping pathways on distributed processors

Multiple pathways are mapped on a set of M distributed machines. The smallest level of communication unit is an operon. The operons in the same pathways are mapped on the single node as much as possible to avoid communication overhead caused by large amount of data transfer between the agents within the same pathway. The system has been partitioned into multiple *heterogeneous pathways*: data intensive similar functions are grouped together in the same pathway. For example, in image processing pathway, color segmentation is an operon, texture analysis is second operon, and shape analysis is the third operon.

The major concern in developing the partition is to have (1) quick agent migration in case of impending failure or (2) to recover from a node crash, and (3) to perform graceful degradation. Initially M pathways in the multiagent systems are mapped to N processors based upon the balancing of the computational load. The computational load for a pathway is the product of the number of data and the computational time C_I needed to process one piece of data from one end of the pathway to another end of the pathway. This mapping is handled by making a connectivity graph between the pathways such that each pathways is represented as a node and an edge represents at least one communication links between any pair of agents in the two pathways. The issue is to partition the graph in a set of subgraphs with the constraint that the neighboring pathways are in the neighboring nodes, and the sum of the computation value of the nodes in the subgraphs in each cluster is nearly the same. The algorithm is based upon sorting (in descending order) the computational loads of all the pathways, and then all the pathways are allocated to the processors such that the next pathway is allocated to a processor that is topologically adjacent to processors that contain the allocated neighboring pathways. The algorithm for such a partition is given in Figure 2.

Algorithm pathway system partition
Input: 1. A pathway load set L = {l_1, …, l_n}; 2. A set of computer nodes C such that $||C|| = M$ ($M > 0$);
Output: A partition set S = {s_1, …, s_M}
{ Perform a descending sort on L. **Let** L_1 be the set of sorted nodes of L;
while L_1 is not empty {
Let $P_i \in L_1$ be the next pathway with highest computational load; $L_1 = L_1 - P_i$;
let N_1 be the set of neighboring allocated pathways of Pi, and C_1 be the subset of computing nodes containing N_1;
if $(C - C_1)$ is not empty {
pick a node $n \in (C - C_1)$ that is topologically adjacent to the nodes in C_1 and has minimum cumulative load value among nodes in $(C - C_1)$; $C_1 = C_1 \cup \{n\}$; sum(n) = sum(n) + load(p_i);}
else {pick a node n ∈ C1 that has minimum cumulative load value among C1; sum(n) = sum(n) + load(p);}
}
}

Figure 2. The initial partitioning of the system

4.3 Shadow pathways for agent recovery

The monitors keep track of the children agents, and keep two shadow agents – one locally and the other remotely for every pathway. The remote shadow agent is placed in the topologically neighboring computer node of the pathway such that the set {node} ∪ neighbors(node) is a subset of the neighboring nodes of the node harboring the shadow pathway. The rationale for such a constraint is that in case of the node crash of the original pathway, the second shadow pathway has to immediately take over. Since the distribution of pathways resident on the crashed computing node will

require registry and failure detection overhead, the overhead of pathway migration has to be kept to the minimum. The state of shadow agents is altered based upon the message from the monitor of the corresponding active agent. The communication agent of the corresponding active agent piggybacks limited number of redundant messages also to the other agents that are hosted by the node hosting the remote shadow agent to provide robustness against the message loss or delay. The message transmission takes place in two steps:

(i) In the first step, the communication agent sends all the belief and state change message to the communication agent harboring the shadow agent. To identify the computing nodes of the neighboring pathways a reachability table is kept in the global directory of agents present in every computing node. The rationale for this is that that belief changes and state changes of an agent from the last checkpoint is stored in a distributed way in at least one of the neighboring nodes so that crash of any node does not remove the beliefs. Broadcasting messages to all the nodes neighboring the crashed pathways also ensures robustness against message loss.

(ii) In the second step every communication agent sends the received message to the communication agent harboring the shadow agent. Every communication agent also maintains a partially ordered distributed logical clock, and stops transmitting the message to the neighboring nodes if the distributed logical clock in the outgoing message precedes the distributed logical clock of the incoming message. It has been shown that this condition of distributed clock ensures receipt of the message by the intended agent [(4)].

(iii) In the third step the communication agent harboring the shadow agent writes the message on a log file, and transmits the message to the monitor of the intended agent. The monitor updates its distributed logical clock, and sends an acknowledgement back to the communication agent, and deletes the message from its volatile memory.

The overall scheme of updating remote shadow agent is shown in Figure 3.

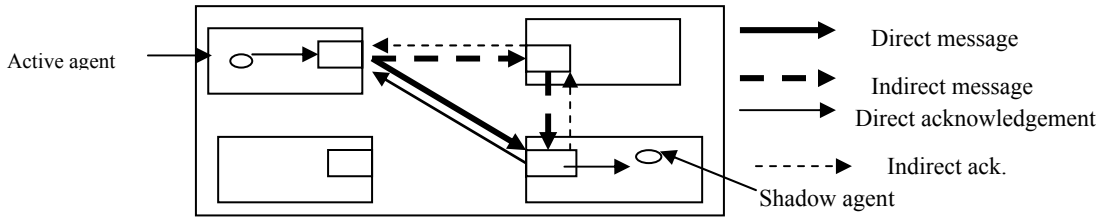

Figure 3. Replicated message based update of remote shadow

The distributed clock tagged (after being incremented) to the next outgoing message will confirm the receipt of the message by the agent. At the same time, a message update is sent directly to the shadow agent. After the shadow agent gets the message, and updates its state, it increments its clock counter. A new message is identified by comparing the archived clock of the active agent and the clock of the active agent included in the message. After updating the state of the shadow agent, the monitor of the shadow agent also sends a direct acknowledgement to the active agent through the communication agent. There are two ways for the monitor of the active agent to know that the message has been received: receiving the direct acknowledgement or the distributed clock comparison between the transmitted and received messages by the monitor of the active agent. If the archived distributed clock for the corresponding message is less than the received distributed clock with any message, the transmitted message has been received. To improve robustness against possible loss of message, the corresponding communication agent of the active agent transmits redundantly the message limited number of times to the node(s) that can transmit the messages (in finite hops) to the communication agents connected to the node harboring the remote shadow agent. These messages are removed from the intermediate nodes as soon as the distributed clock comparison ascertains that the message has been received by the intended agent.

The communication agents have a local shadow to handle their own failure. Any communication agent is connected to communication agents of the other nodes that archive all the messages received. Thus the failure of the

communication agents are protected at two levels: due to the presence of a local shadow agent and due to the distributed message archival that can be pooled in case of the processor node failure.

5. FAILURE DETECTION AND RECOVERY

A major issue is to detect dynamically the failure or impending failure of active agents, shadow agents, monitors at various levels, computing nodes, and communication agents. Failure can occur at many levels as follows: (1) failure of an active agent, (2) failure of multiple active connected agents in the same pathway, (3) failure of an intermediate level monitor, (4) failure of a top level monitor, (5) failure of a local shadow agent, (6) failure of a remote shadow agent, (7) failure of a communication agent of a computing node, and (8) failure of a computing node. Failure may be caused due to many reasons such as computational overload, communication overload, process failure, power outage, and processor failure. Higher level monitors have the responsibilities of detecting failure of the corresponding descendant monitors, and the lowest level monitors have the responsibilities of detecting failure of the corresponding active agents and shadow agents. The root level monitors from different nodes are connected to each other, and they periodically keep pinging each other to check their status using hot channels. Top level monitor is also monitored by a watchdog that is created by the top level monitor at the time of startup. The task of the watchdog is to switch to the local shadow top level root monitor in the case of root monitor failure, inform other distributed top level monitors about the failure.

5.1 Detecting failure of an active agent

Since all the communications to the external agents and shadow agents are done through the monitor and the state changes are logged in the blackboard within the corresponding monitor, the monitor has access to the latest changes in the logical clocks of the corresponding active agent. The monitor keeps an account of the average time t taken by the agent to send next update. If the agent does not update the blackboard within a time m * t (m > 2) then the monitor writes on a log file that agent checks periodically. If the agent is active then it communicates to the agent using a log file to send its latest logical clock within another limited time m * t (m > 2). The value of m and t is adaptive and varies with the computational load. If the message is not received by that time, the status of the active agent is considered sleeping, and a stress pathway is activated by sending a SoS message to the top level monitor. The stress pathway transmits a message to the corresponding shadow agent that activates the shadow agent through a promoter binding. The stress pathway builds the shadow agent by taking the latest information that is still not transmitted to shadow agent, updating the shadow agent, and activating the shadow agent. Meanwhile the stress pathway also keeps monitoring the monitor of the corresponding agent for the latest update. If the active agent responds before the registry changes are made, the activation of the shadow agent is suspended. The stress pathway suspends itself after activating the shadow agent.

5.2 Detecting failure of a local shadow agent

A semi-passive shadow agent has probably failed if the shadow agent does not send the acknowledgement (directly or indirectly through distributed clock comparison) of the state change information to the monitor of the active agent after a time m * t (where t is the average time to receive the acknowledgement and m > 2). The monitoring node retransmits the information to the shadow agent. If the shadow agent fails to respond again, the shadow agent has been blocked. The monitor then initiates a stress pathway. The stress pathway starts a new local shadow agent, and transfers the complete state of the active agent stored in the monitor to the shadow agent. The old shadow agent is not terminated until the new shadow agent is formed. If the old shadow agent responds before the new shadow agent is formed, the formation of new shadow agent is abandoned.

5.3 Detecting failure of a communication agent

Communication agents keep a log file of messages, communicate with top level monitors as well as the communication agents of the other computing nodes. The communication agent can fail individually due to process failure or due to computing node failure. The process failure of the communication agent is ascertained locally by the top level monitor and by the peer communication agents on other nodes. The top level monitor keeps analyzing the communication log created by the communication agent. If the communication agent does not register any message within a time m * t (m > 2, and t is the average time of recording a log), the top level monitor starts a stress pathway. The stress pathway sends

a specific message to the communication agent using a log file that communication agent is supposed to check regularly. If the communication agent does not reply again in a time m * t, then the stress pathway broadcasts the tell-me-status message to all other communication agents on the distributed communication agents on other nodes using the hot channel between top level monitors, and polls their response. In response the distributed communication agents transmit their distributed logical clocks. If the distributed logical clocks show no change in the logical clock of the communication agent for the last 2 * m * t seconds, then the shadow communication agent is activated, and the top level monitor is informed. The stress pathway suspends itself after the shadow communication agent is registered with the top level monitor and other communication agents from different nodes.

5.4 Detecting failure of the computing node

The top level monitors have the responsibility to check the computing node failure, the final determination about the communication agent failure, setting up the shadow communications manager, and keeping track of the status of the computing nodes. The top level monitors continuously broadcast the distributed-logical clock to their peers on other computing nodes. They also exchange the status of the computing nodes, and the liveliness of their agents and monitors to their peers. At any time, if any active agent is overloaded or is dead, the information is transmitted to the top level monitor through the tree of agent monitor. The top level monitors also inform their peers if they are going to shut down or are going to suspend temporarily due to temporary stress.

When the communication agent informs the top level monitor about the possible loss of communication agent residing in another node along with the last distributed logical clock received from the communication agent, the top level monitor broadcasts a request to all the peers to inform the status of the corresponding communication agent. The top level monitor waits for a period $K' \times \tau$ ($K' >> 2$) where τ is the worst time for the top level monitors to respond, and then collects the responses from all the top level monitors. If the top level monitor of the suspected node N does not respond and other top level monitors also do not report response from N, the node N is declared failed, and the corresponding node recovery is started for all the pathways residing on the node N.

During recovery each top level monitor starts the stress pathways. The role of the stress pathways is to (1) lower the rate of the data processing by broadcasting the messages to the appropriate agents, (2) to increase the buffer size of the data buffers to handle any bottleneck caused by the slowdown of the processing of the pathways feeding the data to the pathways of the crashed nodes, and (3) to inform the communication agents to hold on to the data to be transferred to node N until the reorganization of pathways is completed.

The top level monitors from different nodes activate the remote shadow agents of the pathways that were residing in N. After activating the shadow agents, the corresponding local shadow agents are built first by cloning the newly activated agents, the system is started, and then a new dynamic load balancing analysis is done to analyze the load of the computing nodes of the pathways so that new remote shadow agents can be allocated on the remaining nodes.

6. RELATED WORKS

This work has been influenced by many previous works on fault tolerance such as the use of monitors [9], the use of semi-passive replicated agents [13], the author's own work on the use of distributed logical clocks and distributed replication of state change to augment recovery by check-pointing that started fault tolerance in distributed multi agent systems [4]. However, the major thrust of this work has been to simulate stress pathways using a bio-computing model of distributed multi agent systems. ANT based models [8], use heuristics to evolve to do a task efficiently based upon statistical analysis and heuristics. However, they are not suitable for providing fault tolerance in the bio-computing models as it requires quick stress response that may include quick variation, suspension, and activation of new pathways. Incorporating fault tolerance by the use of genomic model has not been addressed before. The author is unaware of any work that addresses the issue of code repair and pathway based load balancing, the use of stress pathways to control the data processing rate to avoid the system failure., and the integration of monitor based model with semi-passive agent replication based upon distributed clock comparison to provide fault tolerance in any known model of distributed multi agent system.

7. CONCLUSION

In this research, the author has extended the genomic bio-computing model of distributed multi agent system to incorporate fault tolerance. The model simulates stress pathways to provide fault tolerance. The fault tolerance itself is an integration distributed logical clock comparison, distributed replicated belief archival to recover the state changes after the last check pointing, semi-passive replication of active agents, and the use of tree of monitors. The stress pathways also provide fault avoidance by reducing or enhancing the data processing rate during stress and during recovery. The dynamic load balancing provides graceful degradation.

ACKNOWLEDGEMENTS

This research was supported in part by Wright Brother's Institute/ U. S. Dept. of Defense through Subcontract # SKM-T1/F33615-97-D-1138 under 'Secure Knowledge Management' project. I thank Bill McQuay and Nikolaos Bourbakis for many useful suggestions and discussions.

REFERENCES

[1] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, J. D. Watson, "Molecular Biology of the Cell," Third Edition, Publisher: Garland Publishing Inc., 1994.

[2] P. J. Angeline, "Multiple Interacting Programs: A Representation for Evolving Complex Behaviors," *Cybernetics and Systems*, 29 (8), 1998, pp.779-806

[3] J. Balthrop, F. Esponda, S. Forrest and M. Glickman. "Coverage and Generalization in an Artificial Immune System," *in Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, Morgan Kaufmann. New York, pp. 3-10 (2002).

[4] A. K. Bansal, K. Rammohanarao, A. Rao, "A Distributed Storage Scheme for Replicated Beliefs to Facilitate Recovery in Distributed System of Cooperating Agents," *in Proceedings of the Fourth International AAAI Workshop on Agent Theory, Architecture, and Languages*, Lecture Notes in Springer Verlag Series, (1998), LNAI 1365, 77 - 92.

[5] A. K. Bansal, "Exploiting Systemic Biological Modeling for Trigger Based Adaptation in Networked Intelligent Multi-Agent Systems," *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004),* Boca Raton, USA, Nov. 2004, pp. 761-768.

[6] I. Burleigh, G. Suen, and C. Jacob, "DNA in Action! A 3D Swarm-based Model of a Gene Regulatory System," *in Proceedings of the First Australian Conference on Artificial Life*. Lecture Notes in Computer Science. Springer-Verlag: Berlin, 2003

[7] D. B. Fogel "The Advantages of Evolutionary Computation," *Bio-Computing and Emergent Computation,* D. Lundh, B. Olsson, and A. Narayanan (eds.), Sköve, Sweden, World Scientific Press, Singapore, 1997, pp. 1-11.

[8] M.J.B. Krieger, J.-B. Billeter, and L. Keller, "Ant-like task allocation and recruitment in cooperative robots," *Nature* 406(Aug. 31), pp. 992-995.

[9] S. Kumar, P. R. Cohen, "Towards a Fault-Tolerant Multi-Agent System Architecture," *In Proceedings of the Fourth International Conference on Autonomous Agents (Agents 2000)*, ACM Press, Barcelona, Spain, June 3-7, 2000, pages 459-466.

[10] S. Kumar, P. R. Cohen, H. J. Levesque, "The Adaptive Agent Architecture: Achieving Fault-Tolerance Using Persistent Broker Teams," *In Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS 2000)*, Boston, MA, USA, July 7-12, 2000, pages 159-166.

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

[12] L. E. Moser, P. M. Melliar Smith, D. A. Agarwal, R. K. Budhia, and C. A. Langley-Papadopoulos, "Totem: A Fault Tolerant Multicast Group Communication System," *Communications of the ACM,* Vol. 39, No. 4, pp. 54-63

[13] S. Pears, J. Xu and C. Boldyreff, *"*A Dynamic Shadow Approach for Mobile Agents to Survive Crash Failures*," in Proc. of the Sixth IEEE International Symposium on Object-Oriented Real time Distributed Computing (ISORC'03),* Hakodate, Hokkaido, Japan, *2003,*pp. 32-38

[14] S. Pears, J. Xu and C. Boldyreff, "Mobile Agent Fault Tolerance for Information Retrieval Applications: An Exception Handling Approach," in Proc. of the International Symposium on Autonomous Decentralized Systems, Pisa, April 2003

[15] S. Pleisch and A. Schiper, "Modeling Fault Tolerant Mobile Agents as a Sequence of Agreement Problems," *in Proceedings of 19th Symposium on Reliable Distributed Systems (SRDS), Nuremberg,* October 2000, pp. 11-20

[16] L. M. Silva, V. Batista, and J. G. Silva, "Fault Tolerant Execution of Mobile Agents," *in Proc. of the International Conference on Dependable Systems and Networks*, New York, June 2000, pp. 144-153

[17] S. Stephney, J. A. Clark, C. G. Johnson, D. Partridge, and R. E. Smith, "Artificial immune systems and the grand challenge for non-classical computation," *Proceedings of the 2003 International Conference on Artificial Immune Systems*, LNCS 2787, Springer, September 2003, pp. 204-216.

[18] K. Sycara, J.A. Giampapa, B.K. Langley, and M. Paolucci, The RETSINA MAS, a Case Study, Software Engineering for Large-Scale Multi-Agent Systems: Research Issues and Practical Applications, Alessandro Garcia, Carlos Lucena, Franco Zambonelli, Andrea Omici, Jaelson Castro, ed., Springer-Verlag, Berlin Heidelberg, Vol. LNCS 2603, July, 2003, pp. 232--250.