Fault-Management in MAS

Peng Xu
University of Saskatchewan
57 Campus Drive
Saskatoon, Saskatchewan, S7N 5A9
pex066@mail.usask.ca

ABSTRACT

Despite the considerable efforts researching and developing Multi-Agent Systems (MAS) there is a noticeable absence of deployed systems. In the past the MAS research community ignored this problem - arguing that it is not a genuine MAS problem and consequently of lesser importance than other unsolved issues like cooperation, coordination, negotiation and communication. However, as the field matures, empirical evaluations of techniques and systems are more commonly used and deployment issues like the management of a MAS become increasingly important.

This paper has two aims; firstly it introduces and structures the area of fault-management in MAS by identifying the key issues and providing an overview of the existing approaches from MAS and related areas. Secondly it introduces a generic framework for fault-management in MAS that has been successfully tested in a large scale MAS.

General Terms

Multi-Agent Systems, Management

Keywords

Fault-Management, Event-Streams, Fault Identification

1. Motivation

MAS are decentralized self-organizing systems consisting of autonomous entities called agents that are designed to solve tasks by cooperating with each other. Using structured messages for communication the agents negotiate and coordinate in a decentralized manner the actions required to solve a common task.

Decentralization and self-organization ensure that there is no need for central coordination and consequently no single point of failure resulting in a more robust design. But with the absence of a centralized coordination it becomes difficult to determine the current state of the system or to predict the effects of actions. This difficulty concerning the determination and prediction of states is made worse by the fact that the functional dependencies between the agents change over time as a result of negotiations at run-time. As a result the MAS appears to behave chaotically. This in turn contributes to the lack of successful deployments.

To overcome these difficulties it is necessary to introduce management functions that will support the monitoring and controlling of individual agents as well as the MAS. ISO [1] defines five basic management functions for any distributed system namely: accounting management, configuration management, performance management, security management and ***fault management*** (Figure 1).

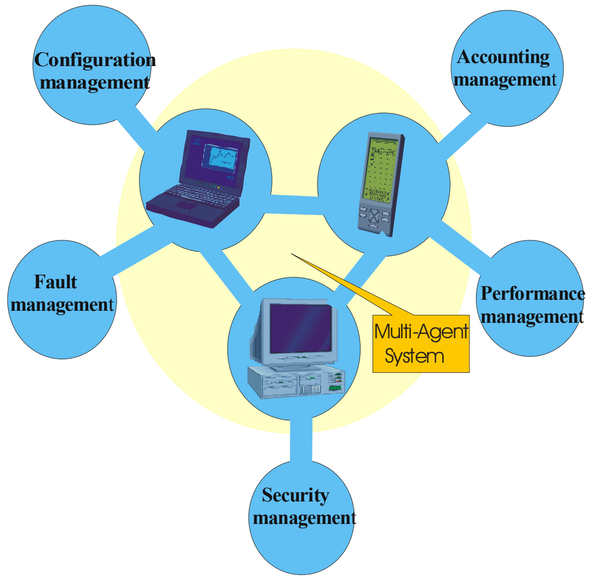


Figure 1: MAS Management Functions

While all five are important management functions I view fault management as most relevant in the deployment of a MAS based on deployment experiences [2]. Since agents are autonomous entities capable of constantly changing their functional interdependencies it is difficult to detect agent failures and to prevent them from spreading (fault propagation) which often leads to a complete MAS failure. Especially in environments which demand minimal downtimes e.g. information systems the early detection, isolation and clearance of faults becomes essential for a successful deployment.

The focus of this paper is on fault management (fault detection) in MAS and is organized as follows. Section 2 provides an introduction into MAS and fault-management and is followed by a review of existing approaches. Section 4 introduces an event-stream framework that has been developed as a means for providing basic fault-management functions. In section 5 a test-bed MAS is presented and used to evaluated the current version of the event-stream framework. The paper concludes with a summary and an outlook on future work.

2. Introduction in Faults, Failures & MAS

This section gives an overview of the key concepts in fault-management and MAS.

2.1 Faults & Failures

Before discussing fault management it is important to distinguish the concepts of *software failure* and *fault*. A *software failure* is the departure of the external results of program operation from requirements. A *fault* is the defect in the program that, when executed under particular conditions, causes a failure [3]. A multi-agent system is a complex distributed system. All possible faults in distributed systems may take place in multi-agent systems, such as processor faults, network faults and software bugs. All these faults can impact the performance of the system and lead to a system failure [4]. From the application point of view, even the most carefully crafted code has been estimated to include an average of three *bugs*, mostly intermittent ones, per 1000 lines of code [5] (individual numbers may vary depending on programming language and skill level of programmers).

In the context of MAS it is useful to view faults and failures predominantly from an agent perspective. Individual agents can encounter partial or total failures as a result of internal or external events. A *partial failure* will result in the degradation or loss of some agent functionality while a *total failure* will result in the complete loss of all agent functionality. Due to the cooperative nature of agents a single agent failure can often result in a phenomenon called *fault propagation* in which the single fault (root cause) of one agent starts a chain-reaction of agent failures with often-catastrophic results.

2.2 Fault-Management

Fault-management can typically be broken down into three basic steps namely:

1) ***Fault Detection (FD)***

   Registering failures of individual system components e.g. agents.

2) Fault Isolation (FI)

   Identifying the cause/fault that lead to the detected failure - in case of fault-propagation the determination of the root cause.

3) Fault Clearance (FC)

   Fixing the determined cause by launching recovery or compensating action.

Ideally, fault-management will include all three steps starting with the detection but most often only the fault-detection is implemented due to the complexity in providing general fault-isolation and fault-clearance procedures.

As mentioned above, fault-management is a complex process and consists of two diagnosis steps (FD, FI) and one planning step (FC). While FD can often be achieved by using rather simple techniques e.g. correlation rules the FI requires the use of an expert system. Consequently it is easier to achieve FD than FI, which is therefore often left to a human expert. FC is a relative simple planning problem that can be solved by standard planners like GraphPlan [6].

2.3 Agent Platforms

As the area of MAS began to mature standards regarding the agent environments and communication emerged. The *F*oundation for *I*ntelligent *P*hysical *A*gents (FIPA) [7] is the current standardization body regarding multi-agent systems. FIPA tries to ensure MAS interoperability by defining standards for architectures, communication languages, content languages and interaction protocols.

As part of its specifications FIPA defines the agent runtime environment (agent platform) in terms of its mandatory services. According to FIPA every agent platform must offer the following three mandatory components/services:

- Agent Management System (AMS)

  The AMS (white pages) provides limited configuration management functionality by storing the agent profiles of all registered agents for that platform. An agent profile contains the agent's owner, state and id. Agents can use the AMS to register, deregister, list known agents or change their profile.

- Directory Facilitator (DF)

  The DF (yellow pages) is the second service of the agent platform and like the AMS designed to provide basic configuration functionality regarding the services offered by the agents. Agents can use the DF to make their services known to other agents on that platform by registering, modifying or deregistering the service profiles. In addition to manipulation their own service data they can also use the DF as a means for locating agents that offer a particular service.

- Agent Communication Channel (ACC)

  The ACC is a basic communication component used for routing the messages (see FIPA for details).

Many FIPA compliant and publicly available agent platforms - Agent Development Kit, April Agent Platform, Comtec Agent Platform, *FIPA-OS* , Grasshopper, JACK ,*JADE* , JAS, LEAP and ZEUS - have been developed. Despite the fact that all platforms offer similar functionality only FIPA-OS and JADE managed to establish themselves as widely used development platforms. JADE and FIPA-OS are both java-based platforms and differ mainly in their agent process mapping. While FIPA-OS enforces a one agent per process model, JADE allows a N agent per process model. As a result FIPA-OS offers a more secure design by encapsulating agents in separate processes at the price of increased resource-consumption. JADE however allows that multiple agents share the same address space resulting in a smaller footprint and faster inter-agent communication at the risk of increased resource and address conflicts between agents.

FIPA does not address the issues of fault-management assuming that this is best left to the MAS designers. But as a result of the open design favored by FIPA it is fairly easy to add new services by simply using an agent as a service wrapper and registering it with the DF.

3. Existing Fault-Management Approaches

Since fault-management is a key management function in any complex system it has long been an area of intensive research. Unfortunately the approaches for dealing with faults are often highly domain specific. This paper therefore limits the discussion of fault-management approaches to domains that are similar to that of MAS. In addition the discussion will be limited to fault-detection since it is the most basic step in fault-management.

3.1 Resource Monitoring in Grid

Grid computing [8] is one of the latest additions to the universe of distributed computing approaches. Like P2P, the Grid aims at offering uniform access to heterogeneous and highly autonomous resource providing nodes. By allowing the users transparent access to the idle resources of remote machines it becomes possible to achieve higher resource utilization. To ensure that users can access the remote resources it is paramount that the resources and their usage is subject to continuous monitoring. Grids and MAS are similar in the sense that both consist of highly autonomous components (agents, locally managed computers) and that failures of their components can easily lead to unwanted fault-propagation. The grid monitoring architecture [9] is used as a performance prediction service. It takes monitoring data (event messages) as input into a prediction model, which is in turn used by a scheduler to determine which resources are assigned to which user/job.

The grid monitoring uses a simple event model consisting of event sources (producers), event sinks (consumers) and a lookup (directory) service.

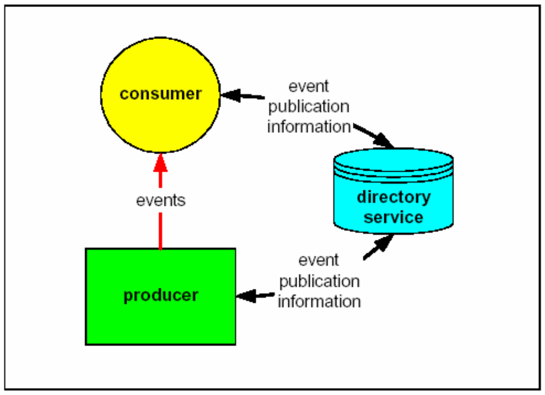


Figure 1: Monitoring in the Grid

- Directory Service

The directory service is used as a basic look-up service that enables the communication between consumer and producers. Consumers can use the directory service to discover producers of interest, and producers can use the directory service to discover consumers of interest.

- Producer

The event producers emit performance data regarding a resource in form of structured event messages. These event messages are sent to subscribing event sources (consumers). Performance data can be gathered from various sources such as hardware and software sensors or specific sub-systems.

- Consumer

The Consumers are the event sinks that analyze the data. The grid monitoring approach defines the following three basic categories of consumers:

- Archiver: aggregate and store event data in long-term storage for later retrieval or analysis.
- Real-time monitor: collect monitoring data in real time for use by online analysis tools.
- Overview monitor: collect events from several sources and use the combined information to make a decision that could not be made on the basis of data from only one producer.

One consumer can also collect event data from several producers, use that data to generate a new derived event data type, and make that available to other consumers. Such consumers can be called intermediaries.

3.2 Fault Management in Telecommunication

Due to the importance of telecommunication networks significant efforts are spent on ensuring that these systems have minimal downtimes. As a result of their size, heterogeneity and dynamics they tend to be difficult to manage which has lead to the development of extensive management support infrastructure.

The widely accepted TMN standard [10] provides a uniform view on the network components and enables the development of generic management functions. The TMN model is based on the concept of representing the various network entities (NE) e.g. switches, routers, links in form of logical objects called managed objects (MO). The MOs are wrapper objects that enable a uniform view to the NE by offering standardized interfaces (e.g. Q, Q3).

The fault-management in a TMN network is based on event messages. A MO can emit event messages in form of notifications e.g. periodic reports or alarms e.g. error reports. Event messages are sent from the MO to predefined event forwarding entities that allow for a decentralized event processing. The event forwarding entities send the pre-processed events in form of meta-event to an operator.

In the management of a TMN network the pre-processing of event messages is of great importance since operators have difficulties handling large numbers of event messages. Obtaining the knowledge on how to preprocess (e.g. correlate, filter) event messages is by no means an easy task especially since the telecommunication networks are subject to constant reconfigurations and modifications.

Sterritt [11] presents a three-tier architecture for discovery and learning of event processing rules.

- Tier 1 – Visualization of Event Messages

The visualization correlation tier allows visualization of the data in several forms. It provides data interpretation and evaluation throughout the knowledge discovery process, from data cleaning to data mining.

- Tier 2 - Managing the Correlation Rules

The second tier supports the definition of correlation rules that are discovered by experienced operators.

- Tier 3 – Discovering Correlation Rules

The third tier mines the TMN (Telecommunications Management Network) messages to produce more complex correlation rules.

This three-tier architecture enables both computer-aided human discovery and human-aided computer discovery and shows how an integrated solution consisting of such different components as visualization tool, rule-tool and machine-learning tool can form a very useful fault-management solution.

3.3 Agent Tracker

Tambe [12] introduced the non-intrusive concept of agent/team *tracking*. By monitoring the agents' actions and communication it is possible to infer their goals, plans and intentions of the agents.

Agent tracking uses an approach based on *model tracing*, which involves executing an agent's run-able model, and matching the model's predictions with actual observations. The main difficulty in agent tracking is that the tracker has to resolve ambiguities in real-time. Tracking can also be used to monitor teams of agents with the aim of identifying the team's joint goals and intentions.

RESC (REal-time Situated Commitments) is an example of an agent/team tracker. A tracker executes a model of the trackee (the agent being tracked), matching the model's predictions with observations of the trackee's action. Due to ambiguities in the trackee's actions, there are often multiple matching execution paths through the model. Given real-time constraints and resource-bounds, it is often difficult to execute all paths or wait so a trackee may disambiguate its actions (delay analysis). Therefore, RESC commits to one, heuristically selected, execution path through the model, which provides a constraining context for its continued interpretations. When tracking teams, team models are used to track a team's joint goals and intentions.

A team model consists of a team state and team operators. The team state is used to represent the team's joint state whereas the operators are the agents in the team which are used to represent the team's commitment to a joint activity.

RESC tracks an agent team as follows:

1. Execute the team model in the tracker - commit to a team operator hierarchy and apply it to a team state to generate predictions of a team's action. In doing so, if alternative applicable operators are available (ambiguity):
    a. Prefer ones where the number of sub-teams equals the number of roles.
    b. If multiple operators are still applicable, heuristically select one.
2. Check any tracking failures, specifically, match or role failures; if none, go to step 1.
3. If this fails, determine if there is a failure in tracking the entire team or just one sub-team. In case of team failure, repair the team operator hierarchy. If it is a sub-team's failure, remove the sub-team assignment to role in team operator, repair the sub-team hierarchy. Go to step 1.

3.4 Exception Handling

Klein [13] pioneered the now widely used Exception Handling (EH) method, a domain-independent fault-management approach. It focuses on providing an exception handling service that is "plugged", with little or no customization, into an existing MAS. EH can be viewed as a coordination "physician" that knows about the many different ways a MAS can get "sick" and actively looks (system-wide) for symptoms and is capable of invoking selected interventions strategies to "cure".

The Exception Handling service communicates with agents using pre-defined languages for learning about the exceptions (exception query language) and for describing exception resolution actions (action language). The query language represent the medium by which the exception handling service interacts with the problem solving agents to detect, diagnose and resolve exceptions. While the query language is used to get agent state information, the action language is used to modify it which includes changing the process model (re-ordering, deleting or adding new tasks; changing the resources allocated to a task; canceling tasks) and changing the work package content. At run-time the EH service actively request information about the activities from the agents to detect exceptions (ante failure treatment). If an exception is found the EH service will select a series of compensation actions and execute them by use of the action language.

The agents have to implement the interfaces for the question and action language. The key to the success of EH in a MAS is that highly reusable, *domain-independent* exception handling expertise is separated from the knowledge used by agents to do their "normal" work.

This Exception Handling service has been used successfully [14] for total agent failures in MAS that use the Contract Net protocol (CNET). CNET is a market-based protocol for allocating tasks to agents. An agent (contractor) identifies a task that it cannot or chooses not to do locally and attempts to find another agent (subcontractor) to perform the task. If a CNET agent dies there are several immediate consequences: the customers of the dead agent cannot receive any results; if dead agent has subcontracted out subtasks, the subtasks will become "orphaned"; if the system uses a matchmaker, it will continue to offer the now dead agent as a candidate, which will create confusion in the system.

With the EH service, when an agent joins the MAS, the EH service begins periodic polling of the agent. If an agent dies (does not respond to polling in a timely way), the EH service takes a series of coordinated actions to resolve the problem:

- It notifies the matchmaker that this agent is dead and should therefore be removed from the list of available subcontractors.

- If the dead agent was performing tasks for some customer(s), the EH service immediately asks these customers to re-allocate the tasks assigned to the dead agent.
- If the dead agent had allocated tasks to other agents, the EH service tries to find new customers for these orphaned tasks by acting in effect as a proxy.
- An agent reliability database is notified so it can keep up to date information about the mean time between failures for each agent type.

The EH service makes two assumptions about agents in order to provide these capabilities. One is that it can transparently monitor and if necessary, modify the domain-independent aspects (message types as well as task and agent IDs) of all inter-agent messages. To achieve this, sentinels are added to the system [15] (Figure 2). Every agent (including the matchmaker if any) is "wrapped" with a sentinel through which all in- and out-going message traffic are routed.

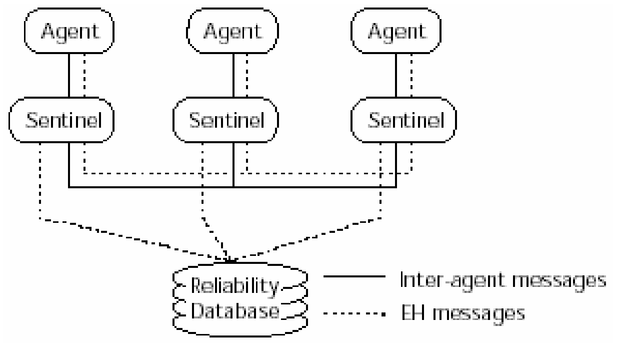


Figure 2: Exception Handling

In addition EH requires that when agents enter a MAS they indicate the kinds of exception handling behavior they can support. This "EH signature" specifies for that agent how agent death can be detected, how dead subcontractor problems are resolved, how dead customer problems are resolved, and how dead subcontractor problems are avoided.

3.5 Conversation pattern

Conversation patterns are the second most often used approach for dealing with faults in MAS. Conversation patterns are defined by a set of conversation actions and policies associates with them. With the conversation patterns, agents can provide services to other agents or even non-agent components in the network that implement the same patterns. Conversation patterns can also be used by management modules to enforce policies, such as obligation and authorization, and measure response time to obligation triggered events.

Whenever an agent offers a service that requires an interchange of messages, the agent can "offer" one or more conversation patterns to its agent client [16]. When the agents agree on using a certain pattern, they can start the interaction by following the policies of that pattern. The conversation pattern can be used as an information source for fault-detection since patterns contain information on correct communication acts between agents. The agents' activities can be monitored and checked without constraining their basic autonomous behaviors.

Patterns can also be reused for other conversation scenarios with some modification, which can cut down on the time spent for their development. An example for adopting conversation patterns can be found in the travel-agency demo [see section 5]. In this demo, the client agents are obligated to initialize the action by providing destination, date and other information to the service provider. The service provider agent is authorized to query clients for additional information or reply with the search result. The obligation and authorization in this conversation can be enforced by predefined patterns. Similar to the sentinel approach, observers are created and used to monitor messages between agents.

3.6 Discussion

The above-mentioned approaches for fault-management showed a variety of ways to deal with faults. In the following a discussion of the approaches in regards to:

- Type of Fault-Management
- Required Knowledge
- Invasiveness
- Overhead
- Openness

The grid monitor [9] uses generic event messages as a means to obtain the state of the resources. Resource providers have sensor units assigned that emit performance data in form of event messages that are sent to analyzers that detect the state. Unfortunately the approach leaves it up to the developer of the analyzer how to analyze the data, which is in indication that hard coded a priori knowledge is being used. The approach is aimed at offering basic fault-detection functionality, in an open non-invasive way and ensures by decoupling the monitoring data flow from the actual usage of the resource that the overhead is kept minimal.

The work of Sterritt [11] for managing faults in TMN networks can be seen as an extension of the approach used in the grid monitor. TMN is based on a event message approach in which the network elements (NE) are monitored by managed objects (MO) that are responsible for emitting messages regarding the state and performance of the NE. The MOs are therefore similar to the event producers in [9]. The contribution of Sterritt is that he structures the analysis of the messages by offering 3 components, a visualization tool, a rule editor and a machine learning tool. With the help of the visualizing tool the various event messages can be analyzed using different views e.g. structured by originator, time, sub-network. In addition it is possible to use the rule-editor to define basic correlation rules to make sure that simple situations can be detected automatically reliving the operator from simple and repetitive tasks. To support the operator in defining the rules a machine-learning component is added which tries to identify non-trivial rules.

The RESC approach is an example of a completely non-invasive approach that relies on eavesdropping and the use of behavioral models. The "Achilles heal" of this intriguing approach is the quality and complexity of the behavioral models. With increased agent complexity more sophisticated models are required which leads to the problem of how to obtain these models and how to deal with the increased computing costs of using the models. Consequently this approach can be regarded as not useful for real-world MAS deployments.

Exception Handling (EH) is the only fault-management approach that supports fault detection, isolation and clearance. This is achieved by introducing two additional languages (exception language, action language) that have to be handled by the agents. In addition EH deals with each agent separately - it lacks the ability to provide a bird's eye view of the system such as traffic load of the system, relationships among agents. It is therefore limited to dealing with single failures and is useless when dealing with fault propagation. Nevertheless its straightforward approach of deploying specialized sentinels is relatively easy to implement which makes it still an attractive choice.

The drawback of the conversation pattern approach is that it can only manage the faults that result in a different behavior during conversions. In addition total failures of agents that have not engaged in a conversation are undetectable with this approach. Like the EH service, conversation patterns are limited to individual or a small group of agents; and cannot provide state information regarding the whole MAS.

Name	Model	Type	Knowledge	Invasiveness	Overhead	Openness
Grid	Events analysis	n/a	n/a	Median	Low	High
Tele-com.	Rule discovery	FD	n/a	Median	Media	High
Agent tracker	Plan recognition	FD	Agent model	Low	Media	Low
EH service	Diagnose	FD/FI/FC	Agent service	High	High	Low
Conv.	Pattern matching	FD	Agent service	Low	Media	Media

Table 1: Evaluation

All of the above mentioned approaches seem to focus only on some issues in the complex process of fault-management. This paper will focus on an event-based approach that will allow the combination of the above-mentioned approaches thus allowing for an open, extendable yet non-intrusive approach.

4. Event-Stream based Fault-Management

In this approach agents are required to report about changes regarding themselves or their environment by emitting event messages to an events manager (EM). By providing the EM with correct and faulty event sequences (events pattern) it can classify the incoming events and react accordingly. If the events of an agent or agent-group follow the "standard" events pattern then this agent/agent-group is performing correct. If a deviation from a "standard" pattern is detected or a match with a failure pattern is found diagnostic and compensating actions can be launched.

4.1 Events

Agent actions are reported as events to the events manager. Possible events are "Creation", "Migration", "Deletion", "State Change", "I/O". Dependent on the agents, any other events are possible [17]. Agents can also report another agent's failure as events. Each event has attributes like agent ID, timestamp, events type, task, transaction ID and etc. An example of a state change event is as follows using CORBA struct.

```
struct StateEvent
{
        string Name;     //name of the event
        string AID;      //agent id
        string From;     //from state
        string To;       //to state
        string TID       //transaction id
        string Time;     //timestamp
};
```

4.2 Events Manager (EM)

The events manage(s) is the central component in the events-oriented approach. It doesn't have to be an agent and can be plugged as a service into the MAS. The EM can support the following tasks:

- Organize events by event types, sender, task or transactions.
- Display events to human, such that allows events viewer to query EM to get desired views.
- Analyze events for fault-detection e.g. matches events sequence with events pattern.
- Correlate events, such that new events can be generated based on different management perspectives.
- Interpret the events to determine agent's state.

An overview of such approach is shown in Figure 3. Agents (colored circles) are distributed on different stations. Every agent should implement a minimum interface for being able to send events to the events manager. There can be multiple events managers existing in the system. These events managers can be equally capable of doing all the above tasks, thus each EM can be put *near* to the agents; works can also be distributed among EMs such that different EM can take different events and provide different tasks. The system may also need to provide duplicated EMs to store events so it can avoid single point failure.

4.3 Fault-Detection

Every agent reports their activities by sending various events to the events manager. To accomplish any tasks, the agent will follow specific steps while executing. Therefore, there are fixed sequence of events associated with each task and agent. These sequences can be predefined as events patterns. For example, the events pattern for individual agent can be the state transaction events sequences. The management module can match the received the events from an agent with the state transaction pattern which is predefined. It will be detected if the agent made an illegal state transaction which is likely to result a fault. The management module can also match the predefined task events pattern with the events sent by multiple agents that cooperating on a task. If a mismatch is detected the management module can discovery which agent caused such mismatch (sent incorrect events or no events), thus detect the faulty behaviour of the agent.

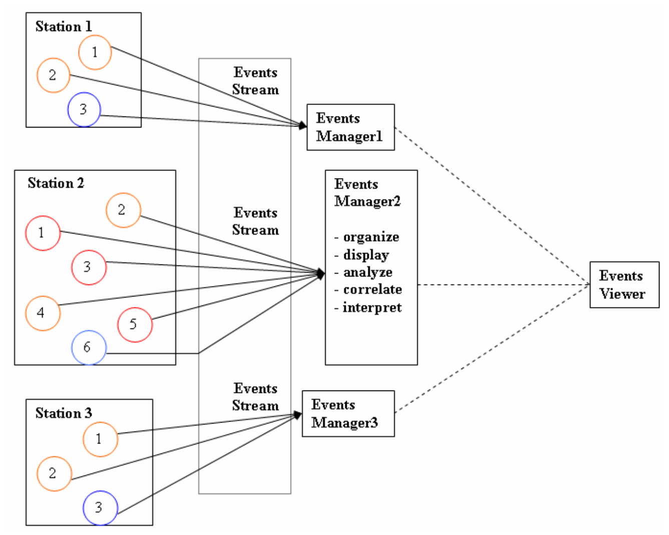


Figure 3: Event-Stream based Fault-Management

4.4 Fault-Isolation and Fault-Clearance

Fault-detection enables the registration of a fault but fails to provide information on the cause (fault isolation) or how to deal with it (fault clearance).

The fault-isolation process will study the faulty behaviours observed from the agent's events. With case-based reasoning the process can match the agent's symptom to existing case. Depending on the case the fault-clearance process will carry out pre-stored treatment strategy onto the agent.

5. Agent DEMO

A *Travel Agency* demo has been developed as test-bed to apply the event-oriented approach using the JADE development platform. JADE was chosen due to the fact that it is inherently more perceptible to fault-propagation as a result of the having multiple agents share the same address.

5.1 Structure and Complexity

This demo consists of a large number of agents with complex dependencies including client, agency, agency clerk, airline and airline clerk agent. There are frequent interactions among these agents such as services registration, task distribution, searching for destination and comparing results (Figure 4).

For making each reservation in the *Travel Agency* demo 39 ACL messages are sent among the agents and 45 event messages are sent from agents to EMs including creation, deletion, state change and IO events. This demo is tested with up to 200 agents running concurrently on Linux-clusters. Up to 15 processes are created with each process containing 10-15 agents during the run time. There are approximately 350 lines of codes per agent.

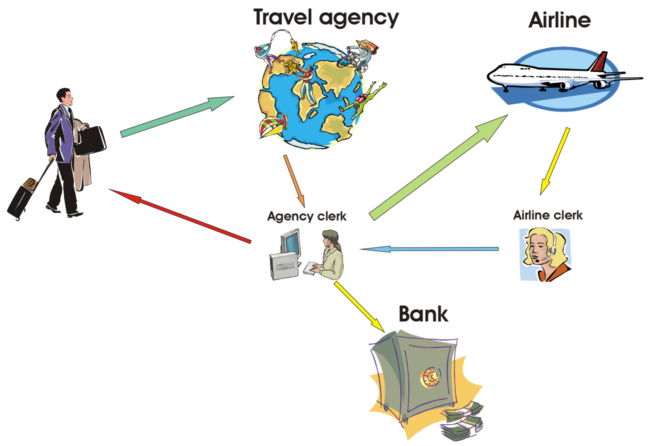


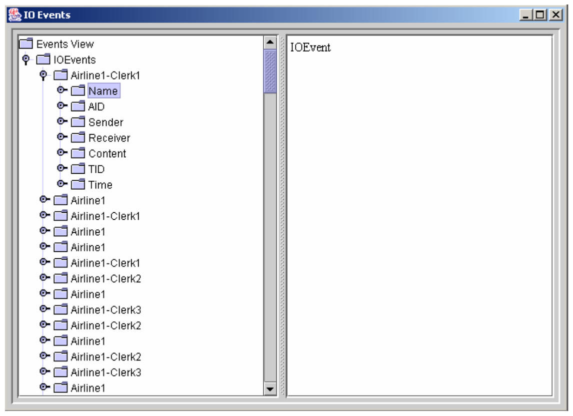
Figure 4: Example MAS

5.2 Events Display

The events-oriented approach provides four types of views for the *Travel Agency* demo so that different management focuses can be satisfied. To create this views, the events manage will group all four types of events by different attributes e.g. events type, sender, task and transaction.

5.2.1 Events View

In the events view, the events manager provides four GUI frames to display four types of events separately. For example, all the IO events are put together by its timestamps with all the information (attributes) including sender, receiver, task, transaction id and content (Figure 5).



IO Events

```
Events View
  IOEvents
    Airline1-Clerk1
      Name
      AID
      Sender
      Receiver
      Content
      TID
      Time
    Airline1
    Airline1-Clerk1
    Airline1
    Airline1
    Airline1-Clerk1
    Airline1-Clerk2
    Airline1
    Airline1-Clerk3
    Airline1-Clerk2
    Airline1
    Airline1-Clerk2
    Airline1-Clerk3
    Airline1
```

IOEvent

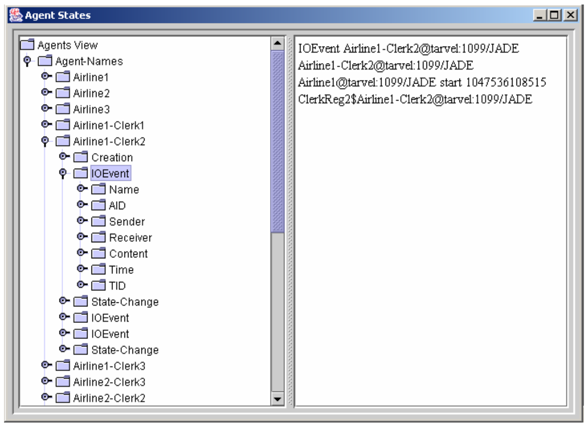
Figure 5: Events View

This events view helps user to focus on one type of event. For example, keep track of the traffic load of the system by looking at the IO events such as how many messages are sent in the system in a certain time period or which agent is sending or receiving the most amount of messages; keep track of the state changes of agents by looking at the state change event such as the agent who generated a lot of state changes tends to be busy (heavy loaded) for this time period and agent that generated no state changes tends to be free; it can also keep track how many agents are created and deleted from the system by looking at creation and deletion events.

5.2.2 Agents View

In the agents view, the events manager provides a GUI to display events by their sender rather than event type (Figure 6).

This agent view focuses on each agent's activities. It collects all the events generated by an agent and lists them by the generated time. In this way, it acts as a record of an agent's life. As each agent works on its tasks, it will perform state transactions from time to time. The correct state transactions for doing one task are predefined as the agent is implemented. The state changes reveal whether the agent is carrying its task correctly. For example, a missing state change or an illegal state transaction can indicate failures. The IO events of an agent reveal which agents are closely related to each other. One agent must have frequent interaction with some other agents if a large number of IO events are generated among them. Thus this agent's failure is likely to affect those agents too. Because the agent will be unable to response to other agents' requests as it failed.



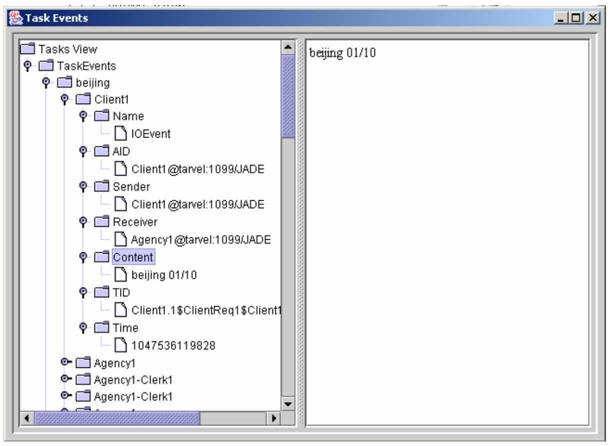
```
IOEvent Airline1-Clerk2@tarvel:1099/JADE
Airline1-Clerk2@tarvel:1099/JADE
Airline1@tarvel:1099/JADE start 1047536108515
ClerkReg2$Airline1-Clerk2@tarvel:1099/JADE
```

Figure 6: Agent View

5.2.3 Task View

In the task view, the events manager provides a GUI to display the IO events by the task the events associated with (Figure 7). Each task is defined as a reservation in the current system.

Since agents are task-driven so it makes sense to provide a view of all the tasks in the system. All the subsequence events of task are listed below by their timestamps. As mentioned in early part of the paper, events patterns for correctly carrying out a task or incorrectly carrying out a task are defined in advance. Therefore, the events of a task can reveal whether the task is carried out correctly or not. If the events match the events pattern for correctly carrying out a task then the agents are doing well; otherwise, if it falls into any incorrect pattern, the management module can discover this, thus prevent the failure form happening and also know which agent is making mistake. The tasks view also provides other meaningful information of the system. By showing the tasks, it can be revealed which task is most frequently executed and which agents are involved in this task, i.e. Beijing might be a place that many people want to go to.



```
beijing 01/10
```

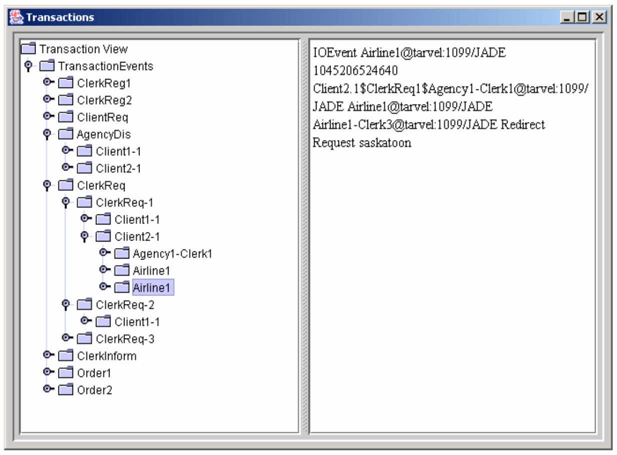
Figure 7: Task View

5.2.4 Transaction View

One drawback of the task view is that each task may be too complex, which involves too many events (45 events in this case). So it's very difficult to match this number of events to the events pattern. One solution is to view the complex task as the parent task that includes many transactions. Each transaction accomplishes one step of the parent task. If the parent task includes a large group of agents' interactions then each transaction is typically the interaction between two or a small group of agents. And each transaction can be further divided to sub-transactions also.

Thus, when the agents report their activities via events they indicate which transaction they are working in. The events associated with the same transaction are grouped together for analysis. However, the agents must be made *transaction aware.* A transaction ID is introduced to accomplish this: the initiating agent in each transaction creates the transaction ID, it passes the transaction ID in ACL messages when it interacts with other agents; the subsequent agents in the transaction merely adopt the ID from the sender agent and include it in the events message. The agents will pass the transaction ID till the end of the transaction, then a new transaction ID will be generated by the first agent in the next transaction. Each transaction ID includes four parts: name of the agent which initiates the parent task, the number of times the task is executed, name of the current transaction, name of the agent who initiates this transaction.

A transaction view is showed next (Figure 8) where AgencyDis and ClerkReg are both the name of the transactions and Client1, Client2 are the name of the agent that initiated two parent tasks correspondingly. In this case, each ClerkReg transaction includes three events, which are from Agency-Clerk1 and Airline1. The transaction view provides a step-by-step monitoring of the agents' execution. It releases the burden of the management component in examining large amount of events and provides the ability to focus on each step separately. With the transaction view, the viewer can clearly visualize the execution of the tasks in MAS step-by-step, and decide at which step the execution failed. In case of large scale system the viewer can always chooses what kind of tasks to monitor, e.g. choose by the name of the task or the agent that performing the task.



```
IOEvent Airline1@tarvel:1099/JADE
1045206524640
Client2.1$ClerkReq1$Agency1-Clerk1@tarvel:1099/
JADE Airline1@tarvel:1099/JADE
Airline1-Clerk3@tarvel:1099/JADE Redirect
Request saskatoon
```

Figure 8: Transaction View

5.3 Fault generator

There are two kinds of failures considered in the multi-agent system with respect to their causes:

- Hard failures. Software or hardware crashes that cause suddenly disappear of agent or agent container or the whole platform.
- Soft failures. Caused by the faulty behaviors of the agent such as slow down execution, incorrect response to request or unexpected death.

To exam the performance of the events-oriented approach it is necessary to simulate some failure scenarios in the system. Hard failures tend to be easy to generate: i.e. deactivate agents. But it needs some effort to generate soft failures. To accomplish this, a fault generator is implemented. It is an agent that possesses several faulty behaviors. It can send the faulty behaviors to other agents and other agent will start to execute these behaviors. Depends on the nature of the behaviors the system will run into different failures. The fault generator provides three common faulty behaviors in MAS:

- Unexpected death of agent.
- Unreasonable delay in answering requests.
- Incorrect responses to requests or any other unexpected behaviors

6. Results

To evaluate the visualization layer of the event-stream approach several experiments were conducted with the following questions:

1. How quickly the operator realizes the failure with the event-oriented approach.
2. How quickly the operator locates the agent that causes the failure.
3. Whether the operator can figure out what causes that agent to fail.

Two graduate students in computer science with experience in MAS and JADE participated in this experiment. After an initial learning phase in which the MAS and the faults were explained they were asked to manage the system. While they were managing the system the fault generator was used to:

- Manually kill one or more agents (hard failure).
- Inject faulty behavior to causes unexpected death (generate deletion event before death).
- Inject faulty behavior to cause delay in responding request.
- Inject faulty behavior to cause incorrect response to requests.

The students could immediately realize the failure of the system given an incomplete transaction view. It took up to 2 minutes for the students to locate the first hard failure. The other hard failures took much less time due to experience. Unexpected deaths of agents (soft failure) could be located in 30 seconds providing the deletion events that presented in the agent view. The agent delay in responding request could take any length of time to discovery depending on the knowledge of the students possess about the system. Because there is no timing mechanism presenting in the system so responding delay could be considered as agent death unless the transaction view or agent view are refreshed. Incorrect responses to requests could be discovered in 30 seconds to 1 minute by matching the received events with the standard events pattern.

7. Conclusion

The complexity and all other features of multi-agent system determined the importance of a fault management infrastructure for MAS. An event-oriented approach provides a domain-independent solution for fault management in MAS. It only requires each agent to report its activities in events but not affect any designed behaviors of the agents. The low invasiveness of this approach makes it easy to be applied to any MAS. Because the functionality of events managers can be distributed to multiple ones so this approach can also support large-scale system and allow easy access to any other applications.

8. Summery

This paper presents the motivation for fault management in multi-agent system and briefly discusses the concepts of agent, FIPA and JADE. It provides a literature view of related works in grid system, telecommunication system, agent tracker, exception handling service and conversation pattern. An event-oriented approached is introduced and tested that utilizes the advantages of the other works. In the event-oriented approach, the agents are required to report their activities by sending events to events manager(s). The events are organized, analyzed and interpreted at the events manager(s). Other application or human can query the events manager to get different events views or state information about the MAS. Several events views are discussed in the paper. A fault generator is created for injecting faulty behaviors to the system and tests how event-approach can help human to do fault management. This is examined by an experiment with several graduate students.

9. Future Work

The current work provides the ability to discovery the failures caused by the faulty behavior of an agent. But what faulty behavior the agent is executing is unclear. Therefore, no resolution can be launched to solve the problem.

The future work will focus on fault-isolation, which is to find the cause of agent's unexpected behavior. Case-based reasoning will be adopted in this process. After the fault-isolation process the work will focus on fault-clearance which will develop various treatment plans for resolving agent's fault.

10. REFERENCES

[1] International Organization for Standardization. http://www.iso.ch/iso/en/ISOOnline.frontpage, 2003

[2] I-Help. http://www.cs.usask.ca/i-help, 2003

[3] John D. Musa, Anthony Iannino and Kazuhira Okumoto. "Software Reliability: Measurement, Prediction, Application". ISBN 0-07-044093-X

[4] Alan Fedeoruk. "Agent Replication in Multi-agent System".2002

[5] J, Gray and A. Reuter "Transaction Processing: Concepts and Techniques" San Mateo, Calif. USA: Morgan Kaufmann Publisher, 1993

[6] Graphplan Home Page. http://www2.cs.cmu.edu/~avrim/graphplan.html, 2003

[7] Foundation for Intelligent Physical Agents. FIPA agent management specification. http://www.fipa.org, 2000.

[8] Global Grid Forum. http://www.gridforum.org/, 2003

[9] Ruth Aydt, Warren Smith, Martin Swany, Valerie Taylor, Brian Tierney, Rich Wolski. "A Grid Monitoring Architecture". July, 2001

[10] Telecommunication Management Network. http://www.tmn.pt/, 2003

[11] Roy Sterritt, "Discovering Rules for Fault Management". Proceedings of Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01)

[12] Milind Tambe, "Tracking Dynamic Team Activity", 1996. National Conference on Artificial Intelligence(AAAI96)

[13] Mark Klein and Chrysanthos Dellarocas. "Exception Handling in Agent Systems". Proceedings of the Third International Conference on Autonomous Agents, Seattle, WA, May 1-5, 1999.

[14] Mark Klein, Juan Antonio Rodriguez-Aguilar and Chrysanthos Dellarocas. "Domain-Independent Exception Handling Services to Enable Robust Open Multi-Agent Systems: The case of Agent Death". Appears in the Journal of Autonomous Agents and Multi-Agent Systems, 2001

[15] S. H?gg, "A Sentinel Approach to Fault Handling in Multi-Agent Systems,"presented at Proceedings of the Second Australian Workshop on Distributed AI, in conjunction with Fourth Pacific Rim International Conference on Artificial Intelligence (PRICAI'96), Cairns, Australia, 1996

[16] Christos Stergiou and Geert Arys, "Policy Based Agent Management using Conversation Patterns". *AGENTS'01*, May 28-June 1, 2001, Montr?al, Qu?bec, Canada.

[17] Sebastian Abeck, Andreas Koppel, Jochen Seitz. "A Management Architecture for Multi-Agent Systems". 1998 IEEE