

# Agent Based Trouble Ticketing System

SENG 609.22 Group Project

Adnan Ahmed  
Mehnaz Shams  
Yongxue Cai

Submitted on: 11/2/2004  
Submitted to: Dr. B. H Far

---

## Table of Contents

No	Topic	Page
1	Abstract & Introduction	3
2	Problem Statement	4
3	System Specification	5
4	Design Documents	11
5	Communication Specification	21
6	Data & knowledge Sharing Specification	29
7	Conclusion	29
8	Reference	31

## Abstract

Network fault management is currently one of the most actively researched areas. More and more network managers are opting for automated network fault management systems due to the speed and accuracy of fault management of such systems. Effective fault management includes identifying the faults, the prompt and efficient isolation and resolution of the faults and making sure those faults don't occur or at least trying to find out why the faults occurred and remembering how to resolve those faults. We use a multi agent based trouble ticketing system to generate such tickets. The agents are responsible for generation of tickets, communicating those problems to other agents around the network, correlating multiple alarms into a more comprehensive and meaningful faults message, keeping historical record of faults and procedure of resolution for future reference etc.

This project describes the design, specification and requirements of the multi-agent Trouble Ticketing System.

## Introduction

As computer networks increase in size, heterogeneity, complexity and pervasiveness effective management of such networks simultaneously becomes more important and more difficult. IT managers constantly face many challenges related to their work. Some of these challenges include staying abreast of the rapid advance of technology and the heterogeneous mix of vendor products that make up current networks, sustaining networks that are scalable, maintaining reliability, and anticipating customer demands, amongst other things. As networks increase in size and complexity, it becomes increasingly difficult for human operators to keep track of all data items, correlate and assign meaning to them in order to actually diagnose faults in the network systems, while avoiding cognitive overload. [1] Many tools and applications are being researched and developed for automating the process of identification, correlation, and resolution of faults, and these tools have capabilities for intelligent reasoning and learning. A very important aspect of the network management domain is the generation of trouble tickets. Currently majority of the network management systems do not include this functionality. Generally trouble tickets are generated or created by human operators. We are suggesting an automatic approach for this task that will make trouble ticketing more efficient and easier to manage. Such a system will in turn make the network management task more efficient.

## Problem Statement

In order for a large network to operate smoothly with minimum errors and maximum output, constant monitoring of network events is required. Network managers aim at minimizing the occurrence of network failure or troubles that cause networks to perform poorly. Some problems are indication of bigger and more serious problems. In most large scale networks when faults occur numerous alarms from various heterogeneous network elements are generated. Some are distinct that need separate attention others are indication of the same faults. Correlating a large number of alarms and faults to understand the core problem is very hard for a human operator. Also it is quite time consuming. Some operators ignore certain alarms that if consulted

with historical records indicate more serious problems with the network and may pinpoint the source of the problem. However in most cases such records are either not kept or never consulted. This leads to only “Fix-when-broken” approach, where instead of finding the core reason operators and organizations wait till network performance degrades to resolve the issue, sometimes only a temporary fix.

## System Overall Specification

An automated solution to the above problem would solve many of issues currently faced by many network management systems. Our solution of a multi-agent based system for trouble ticket generation can bring more speed, accuracy, historical data consultation etc to trouble ticket generation of network managements system. Before we discuss our system in details, a few definitions would be helpful to understand the system better.

### ❖ What is network fault or trouble?

Fault is an abnormal operation that disrupts communication or degrades performance of an active entity in a network. It is a deviation from the expected network performance. A *fault* is an abnormal operation that disrupts communication or degrades performance of an active entity in a network. The term *fault* and *error* are used in the sense that an error occurs when a system deviates from its specified *normal* behavior, and the error is caused by a fault [1]. A fault is assumed to be *atomic* (either happens fully or not) *non-intermittent* (i.e., lasts over a considerable long time, as opposed to transient faults) and *logical* (i.e., affecting the system behavior).

### ❖ What is Network Fault Management?

Fault management is the process of identifying and locating faults in the network. This could include discovering the existence of the problem, identifying the source, and possibly repairing (or at least isolating the rest of the network from) the problem.

### ❖ What is a Trouble Ticket?

Trouble ticket (sometimes called a trouble report) is a mechanism used in an organization to track the detection, reporting, and resolution of some type of problem. Trouble ticketing systems originated in manufacturing as a paper-based reporting system; now most are Web-based and associated with customer relationship management (CRM) environments, such as call centers or e-business Web sites, or with high-level technology environments such as network operations centers. Trouble ticket can be compared to a patient's hospital chart, because both define a problem and help to coordinate the work of several different people who will work on the problem at different times.

### ❖ General Description of the Agent based Trouble Ticketing System (TTS)

The trouble ticket system (TTS) will be part of the network fault management system. As the name suggests it consists of multiple autonomous and intelligent agents that handle the trouble ticketing procedure. The following two types of agents will be part of the TTS:

1. **Manager Agents:** These agents reside on a more stable device or machine and manage the service agent within its domain. There may be multiple manager agents in a large network that communicate with each other. Manager agents only communicate with the agents within their domain. Manager agent is in charge of generating the final trouble ticket.
2. **Service Agents:** Every service agent is under only one manager agent. They can communicate with other service agents within the domain as well with the manager agent in charge of the domain. Service agents correlate alarms to generate a more comprehensive fault message or temporary trouble ticket and communicate it to the manager agent that in turn based on its "Judgment" communicates it to the rest of the network and to the human operator.

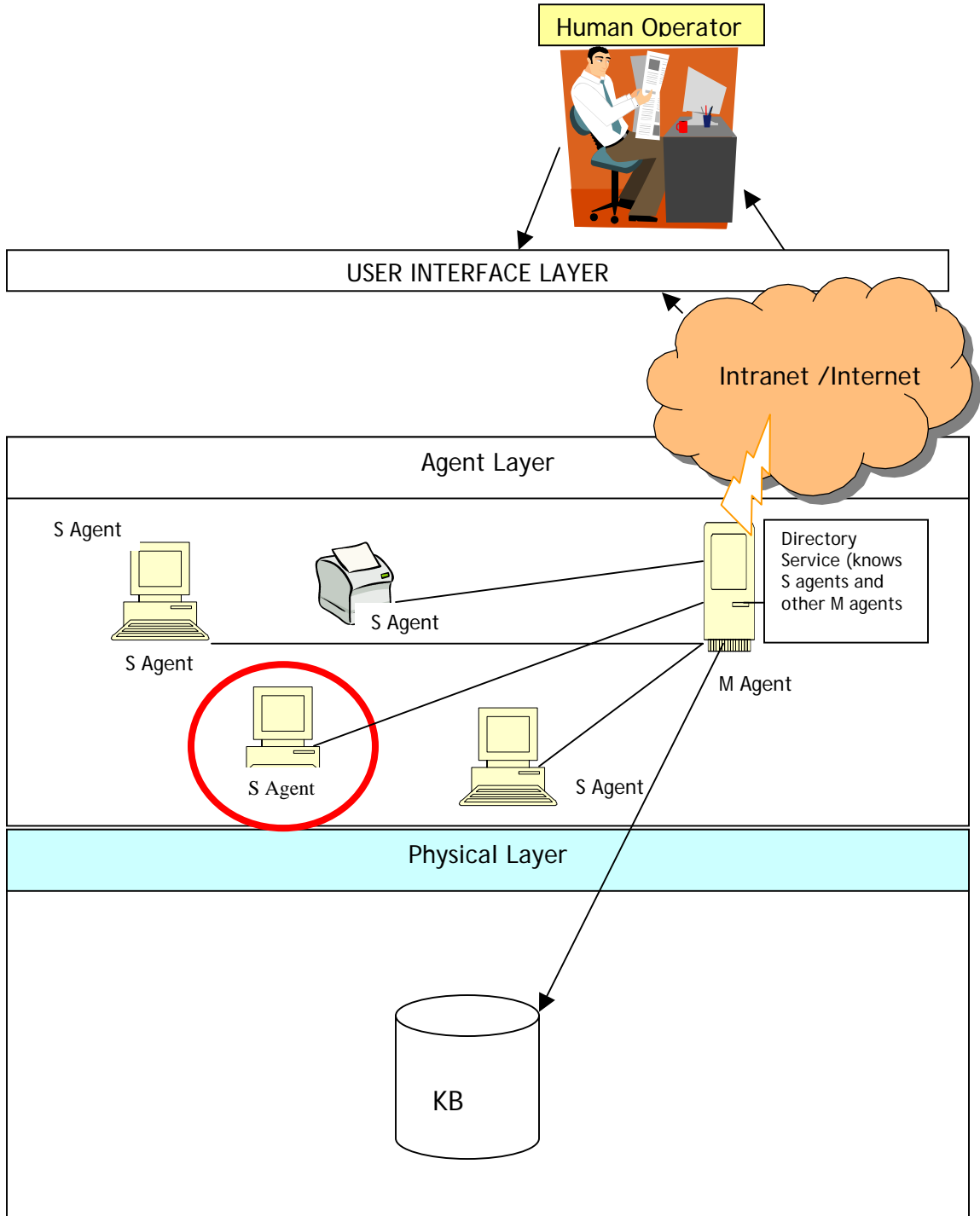


Figure 1. Shows an example scenario with one-manager agents with a few service agents when a fault occurs.

In the above figure there are four service agents that monitor the devices and one manager agent that oversees the Trouble ticketing work of the service agents. When there is a trouble the respective agent on the monitored device (in this figure marked with red circle) alerts the manager agent. And send the temporary trouble ticket to the manager. Once the manager agent receives the trouble ticket it verifies the trouble ticket using the KB in the physical layer to make sure others have not submitted the same trouble ticket. (Such a condition can occur when two or more nodes face the same problem and generally in this case the root cause is the same). After verifying the ticket, if it's a new ticket the manager agent creates the final trouble ticket and contact the other manger agent that are registered in the directory service. In the future the directory service may also be an agent as described in our wish list. Once the managers have been notified it is time to post the trouble ticket to the UI layer and notify the human operator. The human operator resolves the problem and enters the updated information into the UI layer and this in turn is communicated to the manger agent that generated the ticket. The ticket status is then updated. During the whole process the network resumes (maybe at a lower capacity in the domain where the error was generated). However its not a phantom error anymore because the ticket generation is instant and the status updating is also instantaneous. User of such networks are not kept in the dark of the status and they know that its being worked on, and who is working on it and the probable cause of the problem. This is also done through the same UI layer.

The layers of the TTS are described in details in the System Architecture section of this document.

#### ❖ Assumptions

1. The managers Agents reside on devices that are always available.
2. There is at least one communication link working properly from the manger agent to at least one service agent in the managed domain.
3. The agent's ability to successfully correlate alarms and classify faults has already been solved and implemented.

4. The agents use SNMP data (MIB Variables) to diagnose the health of the managed device.
5. A human operator takes prompt action when a manager agent submits a trouble ticket to the NMS (Network management system) website.
6. Human operators are technically proficient in handling the anomalies once it has been detected by the agents and notify the operator
7. Operators enter correctly the steps taken to resolve the issue

#### ❖ Requirements

1. The TTS will be able to correlate the alarms and generate a concise error message to be put on the trouble ticket.
2. The TTS will be able to focus on the source of the problem
3. The System has the ability to archive the steps taken
4. The TTS will have the ability to assign appropriate severity level to trouble tickets based on past experience (knowledge base)
5. The TTS is capable of communicating the troubles to the entire networks via manager agents
6. The system can operate in heterogeneous network with multiple platforms.
7. The network is wired and has fixed topology
8. The nodes in the network supports agents and had SNMP agents on top of which the TTS agents reside.
9. The parent NSM runs 24 x 7

#### ❖ Wish list

1. The TTS system will have "Operator" agents that can take steps to resolve problems based on past experience
2. The TTS system will be able to function in wireless ad-hoc networks
3. The parent NMS will support automatic topology discovery
4. Directory service is handled by "directory service" agent

## ❖ System Architecture

The trouble ticketing system adopts a distributive approach. Using this approach the manager agents are responsible for their own domain and communicate with each other (other manager agents). Whenever information from another domain is required, the corresponding manager is contacted and the corresponding domain managers collaborate to resolve inter-domain problems. [1] However within a domain there is a hierarchical structure between the service agents and the manager agent. The manager agent sits at the top level of the hierarchy and the service agents are right below that level.

Our TTS is divided into the following three layers:

### 1) User Interface layer:

The UI layer is mainly used by the operators to interfacing with the agent system or the conceptual layer. This interface will be a web based interface where the manager agent will post the trouble tickets and the operator records the resolution procedure as well as the status of the ticket.

### 2) Agent Layer:

As the name suggests this layer includes the two types of agents of our system, the Manager Agent and the Service Agent. In the future this layer will also include the Recovery Agent that we mentioned in our wish list.

### 3) Physical Layer:

The physical layer includes the knowledge base where the recovery procedures and past trouble tickets are archived. Once a service agent submits a temporary trouble ticket the manager agent consults this layer to find out if past knowledge is available about this trouble. And post the information to UI layer using XML documents, which is described in details in the communication specification section of this document.

## Detailed System Design

### ❖ Agent Architecture

An agent, in the TTS system, is the program that resides in a given network device. According to the system's specification, the TTS system contains 2 types of agent, Manager Agent and Service Agent. A network may contain different domains, and each domain has its own domain manager, hence force referred to as the Manager Agent. From the perspective of management for the whole network, Manager Agents can cooperate together to solve the problems among different domains. While inside a domain itself, the Manager Agent for this specific domain is an upper hierarchy agent, which controls a set of lower hierarchy Service Agents and those Service Agents reside on managed devices under this domain. The Manager Agent oversees correlation and diagnostic processes in its domain and exercise control over the service agents, while all the service agents perform monitoring, event correlation and data processing services for the managed device assigned to them.

As shown in Figure 2 and Figure 3, each agent is composed of five main components: The Manager Agent consists of the communication engine, the learning engine, the reasoning engine, the data processing engine and the Bayesian network engine; while the Service Agent is composed of the communication engine, the learning engine, the ticket generating engine, the data processing engine and the Bayesian network engine.  
[2]

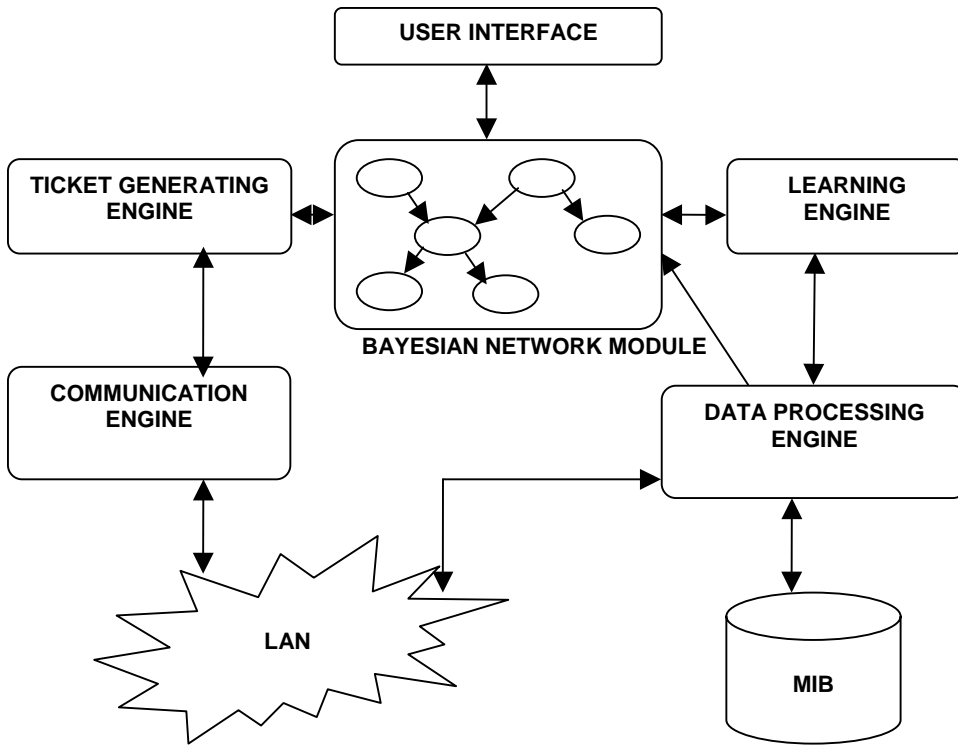


Figure 2 Internal Design of the Service Agent

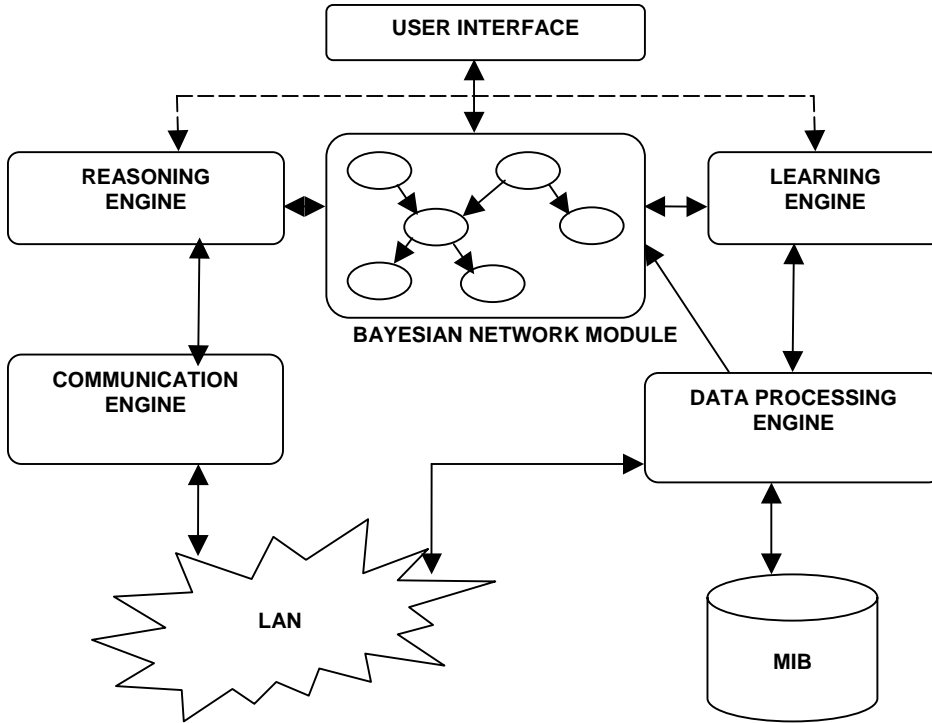


Figure 3 Internal Design of the Manager Agent

### **1) The Data Processing Engine:**

Service Agents receive information directly from the managed devices they monitor. The data is processed in the data processing engine to extract the meaningful information, and the results are passed to the Bayesian network model for situation analysis.

### **2) The Bayesian Network Module:**

Bayesian network is used as one of the major correlation techniques for the intelligent network agent. In this paper, we won't give a detail explanation on how Bayesian Network Module does the situation analysis and generate a fault situation report.

### **3) The Learning Engine:**

The Learning engine handles all learning processes. In Manager Agent, its results are sent to the reasoning engine; while in Service Agent, its results are used by the Bayesian Network Module to dynamically adjust the thresholds for the related MIB variables.

### **4) The Ticket Generating Engine:**

After the situation analysis, the output of Bayesian Network Module is a probability distribution over the set of faults that the Service Agent can detect. In the Service Agent, the Ticket Generating Engine encapsulates an algorithm, which using the entropy measure to quantify the degree of uncertainty of the Service Agent, and uses along with other useful information it can get to generate a trouble ticket.

### **5) The Reasoning Engine:**

In the Manager Agent, the Reasoning Engine receives the situation analysis report from the Bayesian Network Module, and after manipulate through its own reasoning processes, the results it generates will include: the recovery action it decides to take; the identification of the domain of the fault; and the identification of fault sources and causes.

### **6) The Communication Engine:**

The communication engine handles all communication between the agents. In the TTS system, the Service Agent doesn't carry out any recovery actions. It will simply report its findings, hereby, a trouble ticket, and any other useful information, such as current status, to the Manager Agent.[3] The communication between Manager Agent and Service Agent, as well as the communication among different Service Agents themselves are all handled by the Communication Engine.

❖ Use cases Diagram

Three use case diagrams are showed as below:

1) Use case 1:

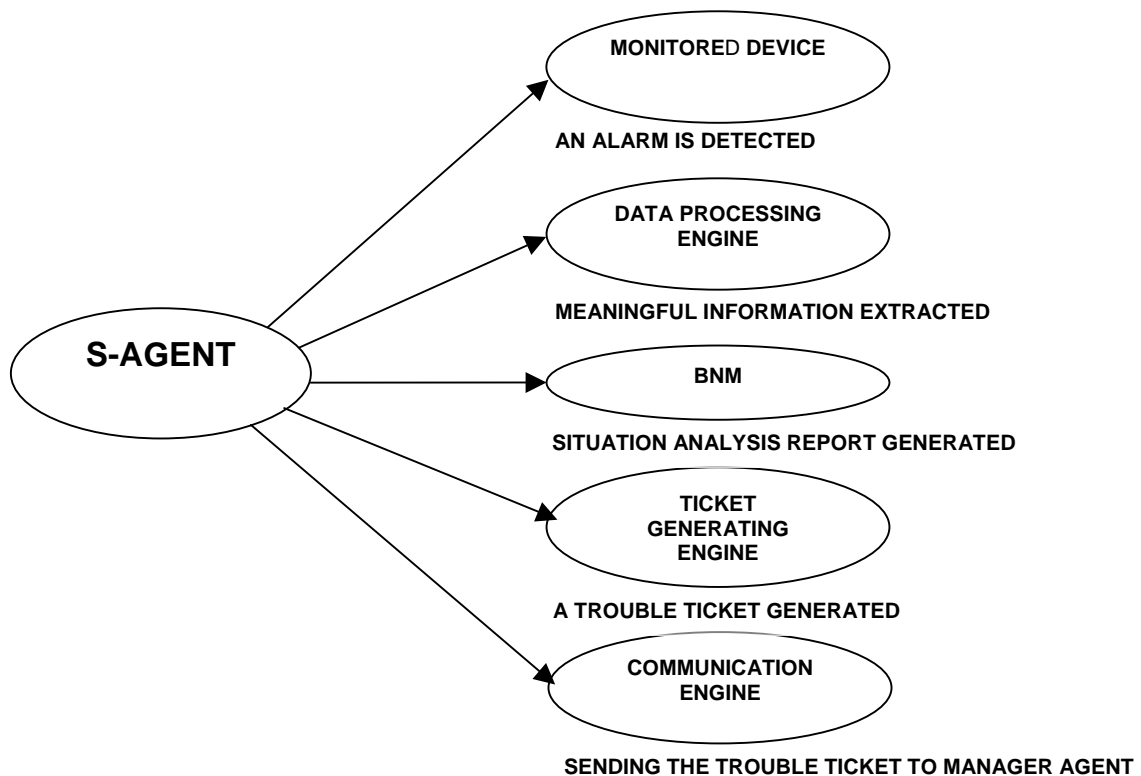
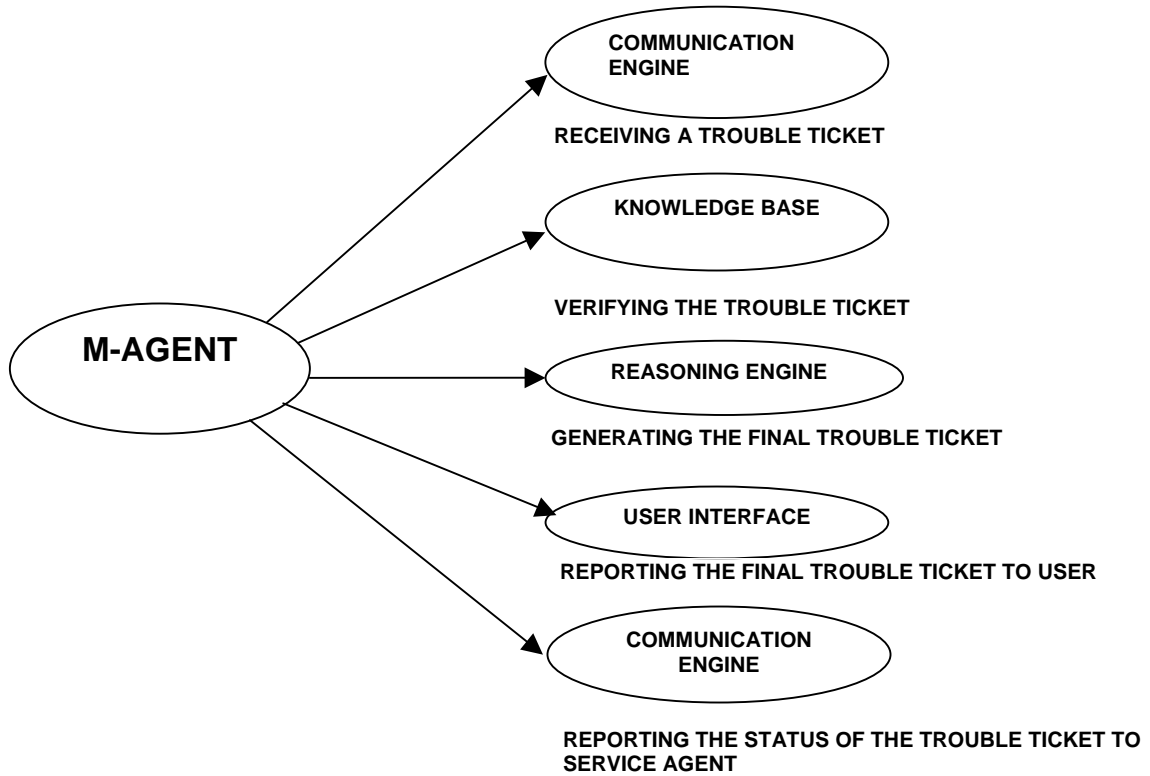


Figure 4 Use Case 1 -- A Service Agent sending a trouble ticket to the Manager Agent

Brief Descriptions:	A Service Agent sending a trouble ticket to the Manager Agent
Pre-condition(s):	Service Agent would skip minor alarms, and will only

	generate trouble tickets when needed.
Post-condition(s):	N/A
Process Steps:	
1	The Service Agent receiving alarms from its monitored device.
2	The alarms are sending to the Data Processing Engine to extract meaningful information.
3	The results from the Data Processing Engine are forwarded to the Bayesian Network Model for situation analysis.
4	The Ticket Generating Engine receives the situation analysis report generated by the Bayesian Network Model and applies into its own processes to do the observation selection. And then, using along with other useful information it can get to generate a trouble ticket.
5	The Communication Engine receives the trouble ticket, and sends it to the Manager Agent.
Exceptions:	
1	Skip the minor alarms, and not generating a trouble ticket.
Relationship:	
Initiating:	Service Agent
Collaborating:	Service Agent, Manager Agent
Data Required:	TicketID, StartTime, StartDate, ProblemSeverity, ProblemDescription, ReporterName, ReporterOrganization, InvolvedMachine

2) Use Case 2:



**Figure 5** Use Case 2 -- The Manager Agent receiving a trouble ticket from a Service Agent

Brief Descriptions:	The Manager Agent receiving a trouble ticket from a Service Agent
Pre-condition(s):	Service Agent would skip minor alarms, and will only generate trouble tickets when needed.
Post-condition(s):	N/A
Process Steps:	
1	Receiving a Trouble Ticket from a certain Service Agent.
2	Verifying the Trouble Ticket from the Knowledge Base.
3	Fault Source Identification from Reasoning Engine, then generating a final Trouble Ticket; if this is a new ticket, notifying to the other Manager Agents in the same network.

4	Reporting the final Trouble Ticket to the User Agent.
5	Sending the current status of the Trouble Ticket to all of the Service Agents under its control.
Exceptions:	
Relationship:	
Initiating:	Manager Agent
Collaborating:	Service Agent, Manager Agent, User Agent
Data Required:	TicketID, StartTime, StartDate, ProblemSeverity, ProblemDescription, ReporterName, ReporterOrganization, InvolvedMachine, TicketOwner

3) Use Case 3:

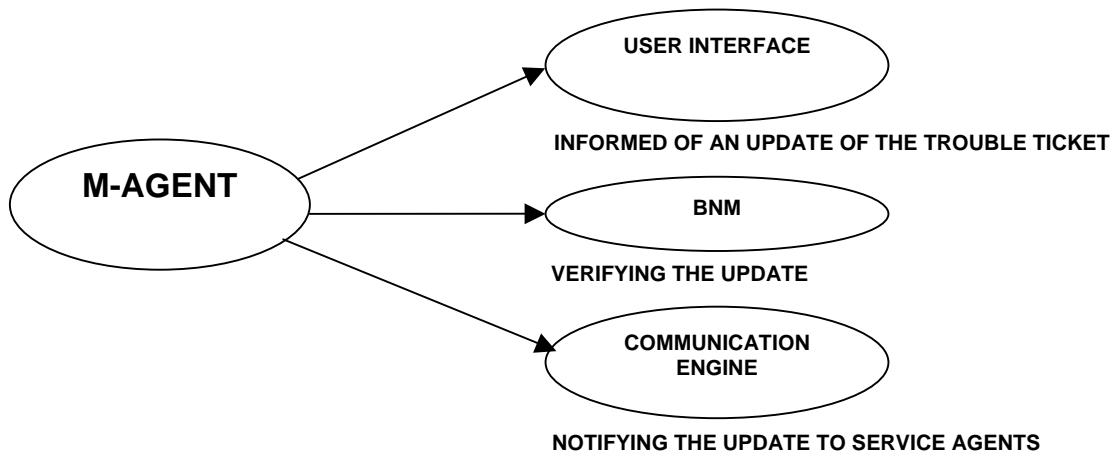


Figure 6 Use Case 3 - The Manager Agent sending out the updates of a trouble ticket

Brief Descriptions:	The Manager Agent sending out the updates of a trouble
---------------------	--

	ticket
Pre-condition(s):	User agent will notify the M-Agent the updates of a trouble ticket through User Interface
Post-condition(s):	N/A
Process Steps:	
1	Receiving a Trouble Ticket from a certain Service Agent.
2	Verifying the Trouble Ticket from the Knowledge Base, and generating a final Trouble Ticket; if this is a new ticket, notifying to the other Manager Agents in the same network.
3	Reporting the final Trouble Ticket to the User Agent.
4	Sending the current status of the Trouble Ticket to all of the Service Agents under its control.
Exceptions:	
Relationship:	
Initiating:	Manager Agent
Collaborating:	Service Agent, Manager Agent, User Agent
Data Required:	TicketID, StartTime, StartDate, Problem Severity, ProblemDescription, ReporterName, ReporterOrganization, InvolvedMachine, TicketOwner

### ❖ Sequence Diagram

The following sequence diagram shows the case where an alarm is received by the service agent, and the service agent informs the manager agent which in turn alerts the operator. The message trace after the resolution is also shown.

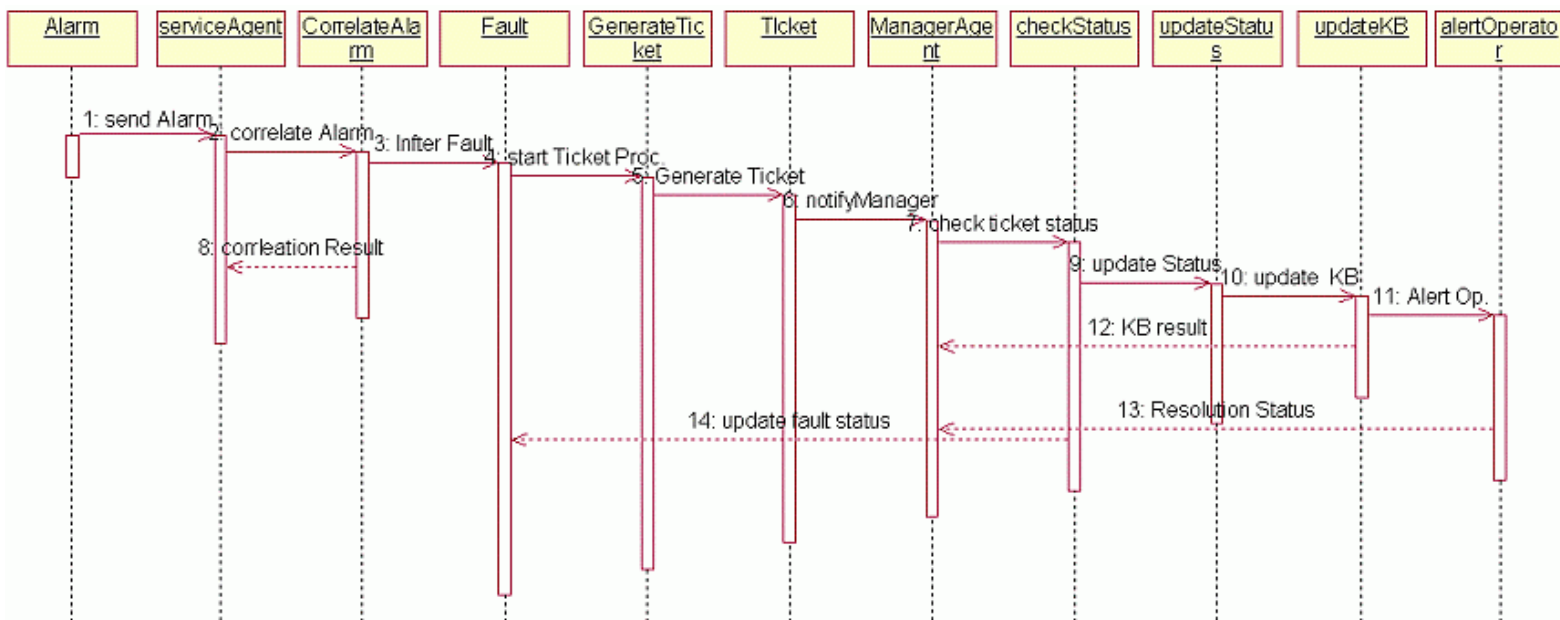


Figure 7 Sequence diagram for the TTS System

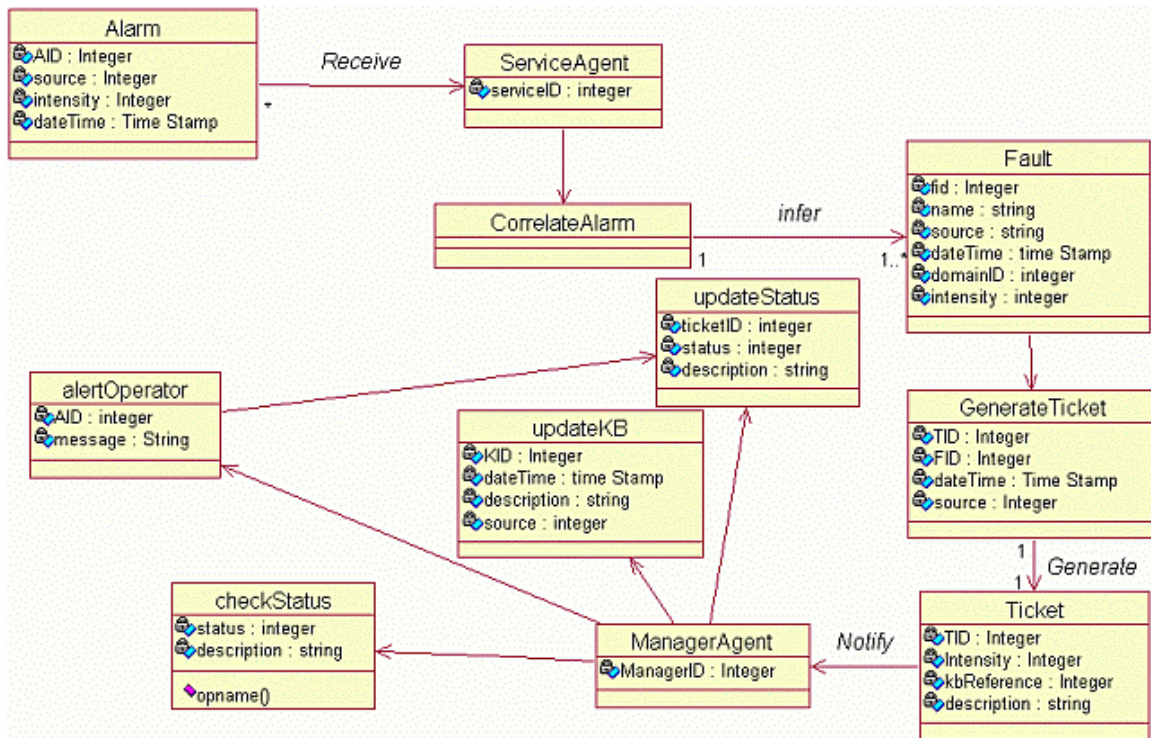


Figure 8 Class diagram for the Simplified TTS System

1. Alarm Class: The alarm class is generated for every alarm that the service agent gets. The class has a unique identifier (AID), a source that represents the node that generated the alarm. Intensity is used by service agents to determine whether it is worth of processing of the alarm. If more processing is needed the service agent gathers more information from the node.
2. Service Agent Class: This class represents the service agent resident on a node.
3. CorrelateAlarms class: This class is used by service agents to correlate the alarm classes that are received by the serviceAgent Class. Similar alarms are grouped by service agent into one alarm class for better handling and housekeeping.
4. Fault Class: The fault class is generated when an agent has a belief about a fault's existence. Fault classes can be uniquely identified by FID and has intensity level that is inferred by the service agent, source identifies the generating agent; description of the projected fault and the domain of the fault are also recorded in the fault class.

5. GenerateTicket: the fault class calls the generateTicket class to initiate the process of generating a ticket. This in turns calls the faultTicket class that is used to actually write the faults into XML formats.
6. The manager agent class is then notified of the fault which then creates the class updateKB to make sure that the fault has already not been reported. If not then it updates the knowledgebase and the status of the ticket and finally posts the trouble ticket to the alertOperator class.

## Communication Specification

The major communication between the agents is the trouble tickets in our TTS. The trouble tickets (both temporary and final) are generated in XML format. The manger agents post the final trouble ticket (XML document) to the web server, which updates the network view on the operator's screen. The resolution of the problem is done and the human operator updates the XML document and the manager agent is notified if the it is updated. This updated information is communicated with the rest of the agent community in the network and the information is verified to make sure that the resolution has been completed and no other problem has occurred due to the actions taken during the resolution.

Trouble ticketing system generally have a pre-set trouble ticket structure that is customized according to the available data and the kind of processing that is done on such data. The following section describes the structure of the trouble ticket in our system:

### ❖ **Trouble Ticket Structure**

Most trouble ticketing systems follow a fixed format or structure for the trouble ticket that it generates. Our trouble ticketing system will follow the structure defined below for its trouble tickets

### HEADERS:

These are the fixed fields that the ticket will start with:

1. Time and Date of problem start.
2. Initials or sign-on of the operator / Agent opening the ticket.
3. Severity of the problem
4. A description of the problem for use in reports.
5. Who reported the problem? (Name, organization, phone, email address)
6. Machine(s) involved.
7. Network involved (for multi-network fault management).
8. User's machine address.
9. Destination machine address. Who should the ticket be dispatched to?
10. Ticket "owner" (one person designated to be responsible overall).

### INCIDENT UPDATES:

The main body of trouble tickets is usually a series of freeform text fields. Optimally, each of these fields is automatically marked with the time and date of the update, and with the sign-on of the operator making the update. Since updates are frequently recorded sometime after the problem is fixed, however, it is useful to allow the operators to override the current time stamp with the time the update was actually made. (In some Implementations, both times will be kept internally). The first incident update usually is a description of the problem. Since the exact nature of the problem is usually not known when the ticket is first opened, this description may be complex and imprecise. For problems that are reported by electronic mail, it is useful to be able to paste the original message in the ticket, particularly if it contains cryptic or extensive information (such as a user's trace route output). At least one such arbitrarily-long freeform field seems necessary to contain this kind of output, although it is better to allow arbitrarily long messages at any stage (e.g., so future complex messages can also be archived in the Ticket). Subsequent update fields may be as simple as "Called site; no answer". Some systems allow these kinds of updates to be coded in fixed fields; most use freeform text.

There should always be an indication of what the next action for this ticket ought to be. Again, this may be implemented as a special fixed field, or by convention of using the last line of text.

Advanced systems may also need a facility to allocate the amount of time a ticket is open between multiple sources. A serious network management system will want to use its trouble ticket system to statistically track its performance on responding to problems. (E.g., Mean Time Between Failure and Mean Time To Repair reports). Frequently, though, repairs are stopped at the customer's request. In these cases the ticket needs to remain open. Each incident update may have a time and date of status change, and that these status changes can be read and aggregated by reporting programs.

#### **RESOLUTION DATA:**

1. Once a problem is resolved, it is useful to summarize the problem for future statistical analysis. The Following fields will cater to that need:
2. Time and Date of resolution (for outage duration).
3. Durations (can be calculated from time of resolution and incident report "customer/NOC time" stamps).
4. Resolution (one line of description of what happened, for reports).
5. Key component affected (for MTBF and similar reports).
6. Checked By -- a field for supervisors to sign off on ticket review.
7. Escalated to -- for reports on how many problems require non-NOC help.

#### **❖ Simple Object Access Protocol (SOAP)**

Simple Object Access Protocol (SOAP) is a protocol initially proposed by Microsoft, IBM, and others, including Don Box and Dave Winer. SOAP provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. SOAP does not itself define any application semantics such as a programming model or implementation specific semantics; rather it defines a simple mechanism for expressing application semantics by providing a modular packaging model and encoding mechanisms for encoding data within modules. The SOAP protocol supports XML document exchange and provides a

convention for Remote Procedure Call (RPC) and specifies a wire protocol for facilitating highly distributed applications. SOAP is similar to DCOM and CORBA in that it provides an RPC mechanism for invoking methods remotely. SOAP differs in that it is a protocol based on open XML standards and XML document exchange rather than being an object model relying on proprietary binary formats. Both DCOM and CORBA use binary formats for their payload. The SOAP gateway performs a similar function to DCOM and CORBA stubs - translating messages between the SOAP protocol and the language of choice. As a result, SOAP offers vendor, platform, and language independence. With SOAP, developers can easily bridge applications written with COM, CORBA, or Enterprise JavaBeans.

From the specification, SOAP is composed of three parts:

**1) A framework describing how SOAP messages should be constructed**

A SOAP message is an XML document that contains the following:

- The Envelope is the top element of the XML document representing the message.
- The Header is a generic mechanism for adding features to a SOAP message in a decentralized manner without prior agreement between the communicating parties. SOAP defines a few attributes that can be used to indicate who should deal with a feature and whether it is optional or mandatory
- The Body is a container for mandatory information intended for the ultimate recipient of the message. SOAP defines one element for the body, which is the Fault element used for reporting errors

**2) A set of encoding rules for exchanging data types**

The SOAP encoding style is based on a simple type system that is a generalization of the common features found in type systems in programming languages, databases and semi-structured data. A type either is a simple (scalar) type or is a compound type constructed as a composite of several parts, each with a type.

**3) A convention for representing remote procedure calls**

One of the design goals of SOAP is to encapsulate and exchange RPC calls using the extensibility and flexibility of XML. Using SOAP for RPC is orthogonal to the SOAP protocol binding. In the case of using HTTP as the protocol binding, an RPC call maps naturally to an HTTP request and an RPC response maps to an HTTP response. However, using SOAP for RPC is not limited to the HTTP protocol binding.

### CORBA

CORBA is an open standard of the Object Management Group. The goal of the OMG is to encourage the development and standardization of an Object Management Architecture (OMA) that provides easy interactions between object-oriented distributed components even in highly heterogeneous networks. Currently, CORBA seems to be the most widely used middleware product used in Windows-based PCs [[www.omg.com](http://www.omg.com)].

CORBA architecture supports object-oriented application developers by providing a distributed object environment where objects are made available independent of their location and implementation.

Users can create their own application objects and make them available to others through the CORBA architecture. The creation of distributed objects can facilitate the realization of applications in distributed and heterogeneous environments.

The architecture of CORBA consists of CORBA Objects, Object Request Broker (ORB) and Interface Definition Language (IDL) components.

CORBA objects represent the OMG object model. CORBA uses the remote-object model. In this model, the implementation of an object resides in the address space of a server. CORBA specifications never explicitly state that objects should be implemented only as remote objects. However, virtually all CORBA systems support only this model.

The Object Request Broker is responsible for the location transparency of distributed objects. The ORB forms the core of any CORBA distributed system; it is responsible for enabling communication between objects and their clients while hiding issues related to distribution and heterogeneity. In many systems, the ORB is implemented as

libraries that are linked with a client and server application, and that offers basic communication services.

IDL is used to specify the interface between the client and ORB and similar to other interface definition languages in that it provides a precise syntax for expressing methods and their parameters. It is not possible to describe semantics in CORBA IDL. An interface is a collection of methods, and objects specify which interfaces they implement. The International Standards Organization (ISO) has selected IDL as a “universal” standard. IDL is a declarative specification language that provides basic data types (such as integer, short, etc.), constructed types (structure, union), and template types (sequence, string).

## **TTS System messages**

In this system, communication architecture should be able to handle different agents that are implemented with different programming languages and may operate on different platforms. As the population of service agents is defined and each new service agent should register before being utilized, message-passing works fine in this setting. SOAP provides the simplest communication system that meets the requirements of this application as it enables different agents with different internal structures to share a common communication language. Other messaging systems like CORBA may also be used but the complexity of the Object Request Broker puts an unnecessary extra pressure on the system.

As mentioned before SOAP uses XML as the communication language. Input and output messages in this system is described:

## TicketGeneration

Every service agent generates a ticket when it finds a problem. These are called the temporary trouble tickets in our system. This ticket is communicated to the manager agent of the domain for verification.

<b>Request:</b>
<pre>&lt; TicketGeneration &gt;   &lt;StartTime&gt;time&lt;/StartTime&gt;   &lt;StartDate&gt;date&lt;/StartDate&gt;   &lt;OpeningAgent&gt;string&lt;/OpeningAgent&gt;   &lt;ProblemSeverity&gt;string&lt;/ ProblemSeverity&gt;   &lt;ProblemDescription&gt;string&lt;/ProblemDescription&gt;   &lt;ReporterName&gt;string&lt;/ReporterName&gt;   &lt;ReporterOrganization&gt;string&lt;/ReporterOrganization&gt;   &lt;ReporterPhone&gt;string&lt;/ReporterPhone&gt;   &lt;ReporterEmail&gt;string&lt;/ReporterEmail&gt;   &lt;InvolvedMachine&gt;string&lt;/InvolvedMachine&gt;   &lt;InvolvedNetwork&gt;string&lt;/InvolvedNetwork&gt;   &lt;UserMachineAdd&gt;string&lt;/UserMachineAdd&gt;   &lt;DestinationMachineAdd&gt;string&lt;/DestinationMachineAdd&gt;   &lt;TicketOwner&gt;string&lt;/TicketOwner&gt; &lt;/ TicketGeneration &gt;</pre>
<b>Response:</b>
<pre>&lt;Generate&gt;   &lt;TicketReceipt&gt;string&lt;/TicketReceipt&gt; &lt;/Generate&gt;</pre>

### *TicketNotification*

Manager agent notifies the other manager agents when it registers a new ticket

<b><i>Request:</i></b>
<i>&lt;TicketNotification&gt;</i> <i>&lt;MachineAdd&gt;string&lt;/MachineAdd&gt;</i> <i>&lt;GeneratedTicket&gt;string&lt;/GeneratedTicket&gt;</i> <i>&lt;TicketID&gt;integer&lt;/TicketID&gt;</i> <i>&lt;/ TicketNotification &gt;</i>
<b><i>Response:</i></b>

### *HumanNotification*

Manager agent communicate with human operator to inform him about the problem

<b><i>Request:</i></b>
<i>&lt; HumanNotification &gt;</i> <i>&lt;GeneratedTicket&gt;string&lt;/GeneratedTicket&gt;</i> <i>&lt;TicketID&gt;integer&lt;/TicketID&gt;</i> <i>&lt;/HumanNotification &gt;</i>
<b><i>Response:</i></b>

### *HumanModification*

Human operator orders to delete a ticket when the corresponding problem is solved

<b><i>Request:</i></b>
<i>&lt; HumanModification &gt;</i>  <i>&lt;TicketID&gt;integer&lt;/TicketID&gt;</i>  <i>&lt;/HumanModification &gt;</i>
<b><i>Response:</i></b>

## DeleteTicket

Manager agent informs service agents to delete a ticket

<b>Request:</b>
<pre>&lt; DeleteTicket &gt;    &lt;TicketID&gt;integer&lt;/TicketID&gt;  &lt;/DeleteTicket &gt;</pre>
<b>Response:</b>

## Data Specification

As it is mentioned before, in this architecture we use message passing for agent communications. To keep the communication as simple as possible and to maximize agents' access to information, we use the modified version of blackboard architecture in which every agent keeps and updates the same copy of ticket database.

If we used a central database, we could save some memory but we would have more communication requirement

Data Element	Description	Type
TicketID	Primary key (Defined by manager agent)	Integer
StartTime	The time of problem start	Time
StartDate	The date of problem start	Date
ProblemSeverity	The severity of the problem	Char (32)
ProblemDescription	A description of the problem for use in report	Varchar (256)
ReporterName	Name of the initial problem reporter	Char (32)
ReporterOrganization	Organization of the initial problem reporter	Char (32)
ReporterPhone	Phone# of the initial problem reporter	Varchar (24)
ReporterEmail	E-mail address of problem reporter	Char (32)
InvolvedMachine	Machines that are involved	Varchar(64)
InvolvedNetwork	Networks that are involved (for multi-network fault management)	Varchar(64)
UserMachineAdd	Address of user's machine	Char (32)

DestinationMachineAdd	Address of machine that the ticket should be dispatched to	Char (32)
TicketOwner	One person designated to be responsible overall	Char (32)

**Conclusion:**

We have proposed a multi agent based network trouble ticketing system in order to enhance the current trouble ticketing systems used in the networking industry. Such a system is not limited to only the network domain but can be extended to other areas as well. The primary difference between the current trouble ticketing systems available in the market today and our system is the use of a multi agent system that is responsible for the issuing, managing and recording of the trouble tickets. The research community is vigorously working towards achieving our ultimate goal of a self healing network. The automatic agent based trouble ticketing system is an important step in achieving this goal.

## Reference:

- [1] E. Ekaette, B.H Far *"Application of Intelligent Autonomous Agents to Network Fault Management"* Dec., 2003
- [2] Dr. B.H Far *"Distributed Software Agents For Network Fault Management"*, 2004
- [3] [http://www.geodise.org/useful\\_links/link\\_services.htm](http://www.geodise.org/useful_links/link_services.htm)
- [4] <http://www.w3.org/TR/2000/NOTE-SOAP-20000508>
- [5] W. Emmerich, *"Engineering Distributed Objects"* , 2000