# EXHIBIT A

# J. TIPTON COLE RESUME

President, Tipton Cole + Company 1983—Present

Tipton Cole + Company is a software development and consulting company. We write commercial database applications and provide both technical and management consulting services to large and small businesses throughout North America and in the UK

Representative Clients:

Exxon Company USA
Exxon Chemical Americas
Exxon Company International
Wells Fargo
Cigna
IBM
Pacific Bell
Medical Innovations
Dun & Bradstreet
Logistics Data Systems
GTE
Akin, Gump, Strauss, Hauer & Feld

Chairman and Vice President, Cole + Van Sickle 1977 – 1983

- Custom software development for HP3000 MPE systems
- Co-author of PROTOS 4GL product for HP3000
- Developed Step-by-Step implementation methodology

Representative Clients:

Coca-Cola
Texas Mill Supply
Karl Rove + Company
Weyerhaeuser
Hewlett-Packard

Certifications:

ETI*EXTRACT Level III Master User
SAP ABAP/4

Programming Languages:

ABAP/4, Access Basic, ALGOL, Basic, C, C#, COBOL, Compass, DataEase, FORTRAN, HTML, LISP, PAL, Pascal, Perl, PL/1, PROTOS, Snobol, SPL, SQL, Tcl, VB.NET, Visual Basic, xBase, XML-XSL

Operating Systems:

AIX, CP/M, HPUX, Linux, MPE II - MPE IV, MSDOS 2.0 - 6.0, MVS, Novell 2.12 - 4.1, OS/360, UNIX, UT2D, Windows XP, Windows NT 3.0 – 4.0, Windows 3.0 - 2003

Education:

Juris Doctorate, University of Texas at Austin School of Law
Master of Arts in Computer Science, University of Texas at Austin
Bachelor of Arts in Mathematics, University of Texas at Austin

Teaching:

Adjunct Faculty, St. Edwards University, Austin, Texas, 1998.
Instructor in senior-level Database Management Systems course
University of Texas at Austin, 1982, Instructor in Data Structures

Deposition Testimony:

*BCE Emergis Corporation v. Community Health Solutions of America, et al* United States District Court, Western District of Texas, Austin Division, Case No. A-02-CA-741-JN, January, 2004.

*Kermit Aguayo and Khan N. Tran v. Universal Instruments Corporation* United States District Court, Southern District of Texas, Houston Division C.A. NO. H-02-1747, April, 2004

*The Chamberlain Group, Inc. and Johnson Controls Interiors, LLC v. Lear Corporation and Ford Motor Company* United States District Court, Northern District of Illinois, Eastern Division, Civil Action No. 1:05-CV-3449, January, 2006.

*b-50.com, LLC v. Xformity, Inc.* United States District Court, Northern District of Texas, Dallas Division, Civil Action No. 3:04-CV-0542-B, March, 2006.

*Hyperion Solutions Corporation v. OutlookSoft Corporation* United States District Court, Eastern District of Texas, Marshall Division, Case No. 2:04-CV-436 (TJW), June, 2006.

*Spreadsheet Automation Corporation v Microsoft Corporation*, United States District Court, Eastern District of Texas, Marshall Division, Civil Action No. 2-05-CV-000127-TJW, February, 2007

Trial Testimony:

*BCE Emergis Corporation v Community Health Solutions of America, et al.* United States District Court, Western District of Texas, Austin Division, Case No. A-02-CA-741-JN, February, 2004

*Kermit Aguayo and Khan N. Tran v. Universal Instruments Corporation* United States District Court, Southern District of Texas, Houston Division C.A. NO H-02-1747, August, 2004.

# EXHIBIT B

# Exhibit B – Documents Reviewed[1]

DISCOVERY MATERIALS

December 15, 2006 Expert Report of Dr. Gary S. Tjaden Regarding Invalidity of United States Patent No. 5,988,078

February 9, 2007 Rebuttal Expert Report of Dr. Gary S. Tjaden Regarding Non-Infringement of U.S. Patent No. 5,988,078

OTHER DOCUMENTS

Bell, Gordon C., "Toward a History of (Personal) Workstations (Draft), ACM-0-89791-176-8-1/86-0001, 1986.

Goldszmidt, G., Yemini, Y., and Yemini, S. 1991. Network management by delegation: the MAD approach. In Proceedings of the 1991 Conference of the Centre For Advanced Studies on Collaborative Research (Toronto, Ontario, Canada, October 28 - 30, 1991). A. Gawam, J. K. Pachl, J. Slonim, and A. Stilman, Eds. IBM Centre for Advanced Studies Conference. IBM Press, 347-361.

Grant, A., D. Hutchinson and W.D. Sheperd. "A Gateway for Linking Local Area Networks and X.25 Networks." ACM 0-89791-089-3/83/0300-0234. 1983.

Korb, J. T. and Wills, C. E. 1986. Command execution in a heterogeneous environment. In Proceedings of the ACM SIGCOMM Conference on Communications Architectures & Protocols (Stowe, Vermont, United States, August 05 - 07, 1986). W. Kosinsky, J. J. Garcia-Luna, and F. F. Kuo, Eds. SIGCOMM '86. ACM Press, New York, NY, 68-74.

Stallings, William. Data and Computer Communications. Third Edition. Macmillan Publishing Company 1991.

Wyle, M.F. and H.P. Frei. "Retrieving Highly Dynamic, Widely Distributed Information." ACM 0-89791-321-3/89/0006/0108. 1989.

[1] Additional materials I have considered in this case are identified in Exhibit E to my December 15, 2006 Expert Report and Exhibit A to my February 9, 2007 Rebuttal Expert Report

# EXHIBIT C

# TMS' Revised Proposed Constructions of Disputed Claim Terms
## January 18, 2007

| Claim Term | TMS' Proposed Construction |
|---|---|
| Television viewing location/particular viewing location | The specific room within a building where a television set is positioned and watched by a user |
| Operator input | A keyboard or other device for manually entering data into a computer. |
| Modem | A data signal conversion device that converts digital signals to analog signals (and vice versa) and allows a computer to exchange data over a standard dial-up telephone line |
| [Receiving/receive] information specific to the type of programming available to the particular [viewing location/television receiver][1] | Having data transferred to a computer such that a user can perform database-type manipulations directly on the transferred data without being connected to the original source of the data. |
| Controller | An electronic device for manipulating information including at least memory for storing and executing computer programs and I/O ports for connecting to peripheral devices, such peripheral devices including modems and display devices. |
| Controller being programmed to perform | Computer application programs are installed on the "controller," as defined above, prior to its use by a user to perform. |
| Available to a particular one of the viewing locations | Deliverable from a programming source to a specific "television viewing location," as defined above. |
| Wide-area network | A data communication network that is not a local area network. |
| Information regarding the geographical location of a particular viewing location | Any information, irrespective of how the information is encoded, that expressly indicates in geographic language (e.g., name of a continent, country, province, state, city, town region, street address, zip code, area code, etc.) the approximate portion of the planet in which a particular "television viewing location," as defined above, is located. |
| Schedule of the television programming | A list, not necessarily ordered in any way, containing at least one item of television programming. |

[1] This claim term was previously identified as "receive/receiving," both in TMS' Proposed Constructions, exchanged October 16, 2006, and in the Expert Report of Dr. Gary S. Tjaden Regarding Invalidity of United States Patent No. 5, 988,078. TMS' proposed construction remains the same, and this revision to the disputed claim term does not change Dr. Tjaden's opinions regarding invalidity.



Case 1:05-cv-00725-SLR-LPS Document 179-2 Filed 02/28/2007 Page 8 of 25

# TMS' Revised Proposed Constructions of Disputed Claim Terms
## January 18, 2007

| Claim Term | TMS' Proposed Construction |
|---|---|
| Memory for storing the information | A device that can retain computerized information for retrieval later. |
| Receive information through the operator input | Signals produced by the operator input in response to user actions and transferred to the controller in the electronic terminal unit. |



Case 1:05-cv-00725-SLR-LPS Document 179-2 Filed 02/28/2007 Page 9 of 25

# EXHIBIT D

# Network Management by Delegation – The MAD Approach

German Goldszmidt,* Yechiam Yemini
Computer Science Dept., Columbia University

Shaula Yemini
IBM T. J. Watson Research Center

## Abstract

Network management systems built on a client/server model centralise responsibilities in client manager processes, with server agents playing restrictive support roles. As a result, managers must micro-manage agents through primitive steps, resulting in ineffective distribution of management responsibilities, failure-prone management bottlenecks, and limitations for real time responsiveness. We present a more flexible paradigm, the Manager-Agent Delegation (MAD) framework. It supports the ability to extend the functionality of servers (agents) at execution time, allowing flexible distribution of management responsibilities in a distributed environment. MAD can store and instantiate delegated scripts, and provides a concurrent runtime environment, where they can execute asynchronously without requiring the manager's intervention. A delegation protocol allows a manager to transfer programs, create process instances, and control their execution. We describe the delegation model, its application to network management, and the design of a prototype implementation.

## 1 Introduction

Industrial enterprises are increasingly dependent upon large scale complex networked systems serving as their information backbones. The likelihood and costs associated with faults or performance inefficiencies of these systems increase with their scale and complexity. Effective manageability of such systems, to ensure fault-free, efficient and secure operations, is thus a vital need of industry. Unfortunately, typical networked systems have not been designed to be manageable. The management problem presents a broad range of non-trivial technical challenges (see [DSYBC89] for sample problems). There are growing efforts by vendors, standards committees [ISO89] and the research community [IFIP89, KMW90], to establish manageable systems. The technical problems involved in accomplishing manageability are not yet fully understood and so their research is still in an embryonic stage.

A typical network management system, [Rose91] involves agents, which are responsible for monitoring and controlling managed-objects of the network, and managers, which have the responsibility for collecting dynamic status data from the agents, interpreting that data, and directing the agents (e.g., how to handle fault scenarios). Both managers and agents use a management protocol to coordinate their activities. The fundamental characteristic of such protocols, is that they establish an interaction paradigm in which program logic resides in managers while managed objects data resides within agents. For example, an agent mantains error counters, e.g., the number of packets that couldn't be transmitted because of errors. To compute an error rate or determine its variation or its relationship with other counters, managers need to frequently query the agent to obtain the value.

Standardisation efforts have focused primarily on management protocols and the structure of management information [Rose91, CMIS89]. The management model that they are based on, however, is only partially elaborated: the management needs and styles to be used are tacitly assumed, and the software architecture required to support the model is left unspecified.

In what follows, we will use the term man-

*Part of the work of the first author was done while at IBM T.J. Watson Research Center

ageability to describe the ability to monitor and control the behavior of a networked system. In this paper, we aim to address the following question: how should responsibilities be distributed among managers and agents to accomplish effective manageability?

Paradigms for distributed computing fix the functionality of processes at compilation time. In the typical client/server paradigm, a server exports a fixed interface, and a client can invoke its functions remotely in a synchronous fashion. While this scheme is sufficiently powerful for many distributed applications, it is too inflexible for many others. In particular, network management systems built on the client/server paradigm centralise responsibilities in client-manager processes, with server agents playing restrictive support roles.

Managers can delegate to agents only primitive monitoring and control tasks, so even non-trivial management tasks they must micromanage agents through primitive steps. This results in ineffective and costly distribution of management responsibilities; produces failure-prone management bottlenecks, whose loss can result in paralysis; and limits real-time responsiveness to problems. These problems are compounded as networks become faster and more complex, and require distribution of management function.

We here present a more flexible paradigm, which supports the ability to extend the functionality of agents at execution time, and thus enables a more flexible distribution of functionality between processes. This scheme builds on a multiapplication process model that supports instantiation, interconnection, and communication of processes. By treating programs as data that can be communicated between processes and later instantiated, we obtain a powerful delegation mechanism. Managers *delegate* to agents the execution of management programs prescribed in a *management scripting language*. Management programs include primitives that permit agents to monitor and control local managed objects effectively without unnecessarily involving managers.

The proposed Manager-Agent Delegation (MAD) framework allows flexible distribution of management responsibilities in a distributed environment. Delegated management processes, defined in the manager's environment, are dynamically delegated to the MAD agent. This agent implements a flexible and responsive management strategy by receiving and executing delegated programs that monitor and control objects in its local environment.

This paper aims to explores fundamental problems that arise in the design of manageable systems. In the next section we focus on the micromanagement problems arising in current management protocols. In section 3 we outline the remote delegation model. Section 4 describes the software architecture of Mad; then in Section 5 we review management scripting languages. Section 6 describes other paradigms for distributed applications, and why they do not completely meet the requirements for delegation, and Section 7 briefly describes the current state of the implementation. Conclusions are presented in Section 8.

## 2 The Problem of Micro-Management

The lack of appropriate expressive power of a management protocol can seriously limit the manageability of a networked system. This will occur when the protocol induces *micro-management*, that is, when the manager must control the execution of a program or script by stepping the agent through it. Micro-management occurs when management protocols are unable to compose primitive management actions in a flexible and efficient fashion, and can lead to highly ineffective and costly systems. Communication bandwidth and manager cycles may be required to accomplish even simple tasks, which places severe restrictions on manageability, barring all but trivial scripts from being executable. Since all management functions must be centralised into the manager, it is rendered most vulnerable to network failures as even simple failures could load its bandwidth and cycles, bringing it down. Once the manager is down, local agents cannot accomplish recovery, as they must wait for instructions. Thus, even a minor failure may potentially lead to an avalanche failure of the management system. The following example illustrates the problem of micro-management.

Consider a network management agent (server) that is responsible for monitoring a link (or a virtual circuit) object. A typical fault management

scenario may consist of detection of certain unusual link conditions, execution of diagnostic testing to identify the cause of the conditions, procedures to handle the problem, and notification to appropriate manager. This is illustrated by the following management script:

```
Example:
Management script for link failure

On (link.control.stat>normal.stat)
    AND (link.q.length>normal.q);
/*trigger on composite event*/
Begin       /* handling actions*/
   link.handle.congestion; /*fix */
   link.test;  /* diagnose*/

   If link.failure then
/* if problem */
    Begin
      recover(link.failure.type);
      ....
      notify(Manager,link.failure.params)
      ....
    End
    ::
End
```

Typical management scripts would similarly use the occurrence of an exceptional event, typically signaled via threshold conditions, to identify potential troubles and trigger diagnostic and corrective management actions. Both detection of the event and execution of the actions involve data and procedures entirely local to the agent and the managed link object. The manger must thus use the management protocol to transfer execution responsibilities to the agent to process the script. What does it take to accomplish execution of this script using a network management protocol such as CMIP?

The OSI CMIS management model [CMIS89], for example, distributes management responsibilities through the following manager-agent interaction primitives: creation (M-CREATE) and deletion (M-DELETE) of managed object instances; retrieval (M-GET) and setting (M-SET) of managed objects attributes; reporting of events (M-EVENT-REPORT); and invocation of actions (M-ACTION). Thus, for example, a manager could use M-GET to retrieve certain managed-object attributes values; then use these monitored values to decide on appropriate control/diagnostic measures, which may be invoked by setting the values of certain managed-object attributes with (M-SET) or activating certain agent actions via M-ACTION.

The manager must first establish an appropriate association with the agent, in order to establish mechanisms to detect the occurrence of the triggering events. Suppose that it polls the respective variables *link.control.stat* and *link.q.length* using M-GET, and evaluates the event condition. (This may be the case with agents whose associations do not include M-EVENT-REPORT capabilities.) Polling, however, is a poor event detection-mechanism, since events may occur very infrequently and hold for a relatively short random time (or require a response within a short time of occurrence). In order not to miss such an occurrence, the manager would have to poll the agent at a frequency determined by the expected holding time (or permissible tolerance to detection latency). A high polling frequency can potentially exhaust the manager communication bandwidth and cycles. Intermittent events (e.g., failures followed by device rebooting [DSYBC89]) might be unobservable through polling. In summary, polling can result in highly ineffective division of event-detection responsibilities among manager and agents.

Because of these intrinsic limitations of polling, management protocols provide primitives for asynchronous event reporting (Traps in SNMP, M-EVENT-REPORT in CMIP). For example, using M-EVENT-REPORT, a manager could be notified of the occurrence of threshold events as in the example above. However, in the absence of mechanisms to specify composite events, it would have to seek independent notifications of the two events:

```
(link.control.stat > normal.stat)
and
(link.q.length > normal.q)
```

and decide on their conjunction. This division of event-detection responsibilities leads, again, to inefficiencies and hazards. Event notifications may be randomly delayed, and thus the conjunction event may go unnoticed even if its components have been adequately reported. Agents and manager must spend bandwidth and cycles to divide detection responsibilities.

Once an event is detected, the management script calls upon certain diagnostic handling to be executed. This requires the execution of certain diagnostic actions (e.g., invoking *link.test*), checking certain managed variables (e.g., *..If link.failure..*) or invoking a corrective action (e.g., *recover(link.failure.type)*). Each of these computations may be accomplished via an appropriate CMIP primitive. Thus, diagnostic-testing may be invoked using M-ACTION, and the value of *link.failure* may be retrieved by the manager via M-GET. There may be, however, a problem in invoking a procedure that requires parameters as in *recover(link.failure.type)*.

However, both SNMP and CMIP provide limited or no capabilities to combine primitives to handle composite scripts. Thus, the manager must step the agent through the execution of the script. In this example, the manager must step the agent through the execution of:

```
link.handle.congestion;
link.test;
:::
```

via two (or more) invocations of M-ACTION. This is followed by a retrieval of *link.failure* (using M-GET) to evaluate the conditional statement, and with additional calls upon the agent to complete the rest of the script. In other words, the manager must micro-manage the execution of the script.

Centralisation of management is inconsistent with the emerging needs of complex, large-scale or high-speed networks. In such networks the complexity of management needs and the speed at which a response may be required may render centralisation ineffective or even impossible. Nor is management centralisation consistent with the trade-offs suggested by current and future technology trends. Agent processing cycles are abundantly available, increasingly so as one climbs in the protocol hierarchy towards managing high-layers entities. The tacit assumption in the design of current management protocols that agents lack management "smarts" and should leave all that to manager cycles, may reflect the tradeoffs represented by traditional voice and teleprocessing networks of the past rather than networks of the future.

The micro-management problem can be resolved by delegating entire composite management scripts from managers to agents. Thus, in the example above, a manager would pass the entire script to an agent that is responsible for the link object. The agent would then execute the composite script without manager intervention (except when the script itself requires it). Indeed, the agent is ultimately responsible for executing *all* the primitives of the script, from event detection to invocation of local actions. Management involvement in execution is neither necessary nor useful, and is only mandated by the limitations of current management protocols.

## 3 MAD: A Software Model for Management

We introduce an enhanced software model and architecture for distributed management applications. The central component of this architecture is the MAD agent, a hybrid combination of a hierarchical manager with a proxy agent which provides a collection of services required for effective monitoring and control of managed objects. A MAD agent is analogous to a middle level manager in a large organisation, which has control over some scope of the global environment. As a traditional server process, it encapsulates a level of local autonomy, and exports a fixed set of services that can be invoked by client processes. In addition, it is a *Elastic Server* capable of dynamically adapting its set of responsibilities via *delegation*.

### 3.1 Remote Delegation

One of the most significant properties of MAD is that it provides a new, general purpose scheme for interaction between distributed components, remote delegation. Its main feature is to allow management programs to be defined in and by managers and then delegated to MAD agents for their execution. Delegation provides a flexible way for managers to dynamically associate functionality and responsability with other managers. It enables a manager to transfer the responsibility for performing certain functions to a remote agent. It allows the agent to perform these functions independently of the manager's execution, except were coordination is required. This framework enables a manager to augment, during execution time, the functionality of a subordinate

agent, in effect allowing it to perform an open-ended set of management programs.

Thus, management programs can be designed and coded as part of a specific management scheme rather than as part of the individual agent design. This can be done much later than agent design and deployment, and can be tailored to the specific requirements of any given installation. By taking advantage of the runtime services provided by MAD, (which are described in following sections), management programs can be dynamically delegated by managers on-the-fly.

Delegation empowers manager processes to flexibly adapt management responsibilities assigned to agents, to be executed in close proximity to the managed devices. This enables managers to ensure a much faster response in the event of faults. Furthermore, it provides an effective tool to balance the computational requirements of management applications as temporary conditions change over time, e.g. network load, and as the configuration of the network evolves, e.g. adding new devices.

From a software configuration perspective, delegation provides a simple yet powerful scheme to dynamically compose management systems, by connecting and integrating independent delegated scripts. It is not required, though allowed, that manager process actually write programs to be delegated during their execution. Installations can provide libraries of such programs, which encapsulate network management expertise, to be used as management conditions require. Designers and vendors of devices should provide libraries of predefined routines that can be used to compose delegated management programs. Using the facilities provided by MAD, manager processes can build, configure and control distributed multi-process applications.

A process that delegates responsabilities to another process is referred to as a *MAD manager*. The process that receives the delegated responsabilities is referred to as a *MAD agent*. The roles of a MAD manager and a MAD agent refer specifically to a given set of functional responsabilities. A process which is a manager with respect to a given agent may itself be an agent with respect to another manager.

### 3.2 Delegated Management Programs

A Delegated Management Program (DMP) is transferred via a delegation protocol, from a delegating manager to an accepting agent, which stores it in its Repository. Several important issues are raised (and will be addressed) regarding the use of remote delegation: What type of programs are the DMPs? In which language(s) should they be written? What are the semantic implications of writting DMPs in a general purpose programming language or a specific management language? How is the execution of a management program controlled (e.g., when is it invoked, terminated)? How can a DMP instance obtain access to managed objects? How can a DMP coordinate its execution with other management programs and with managed objects? We examine some of these questions in the following.

### 3.3 Types of Delegated Management Programs

From a programming perspective, there are three basic types of DMPs: *complete processes, objects and pure functions*. Complete process execute an independent program with its own thread of control and its own private data. It contains logic to control its communications with managers, managed objects and other DMPs. Object DMPs are similar to complete processes in that they keep state in the MAD agent environment. However, they are passive server which go into a sleep state waiting for requests. When a request message is received, the proper actions are taken. A pure function DMP is a program whose instantiation may return some value but will not have any side effects. Thus, the code of a pure function DMP can easily be shared by many other DMPs. In the sequel, unless stated specifically, all DMPs are complete processes, since this is the most general case, e.g., all others are degenerate cases.

A DMP is written in a Management Scripting Language (MSL). There is a whole spectrum of languages that can serve as MSLs, from general-purpose programming languages such as C, to more restricted specialised languages for management. (We discuss some of the issues in selecting and supporting a particular MSL in Section 5.)

### 3.4 Instantiation of Delegated Management Programs

A Delegated Management Process Instance (DMPI) is an executing instance of a DMP. The lifetime of a DMPI can be best conceptualised as a lightweight process with its own thread of execution in the MAD environment. DMPIs start being executed when instantiated by a MAD agent upon request by an authorised manager. Typically the delegating manager is authorised to request instantiation of the DMPs that it delegated. When a DMP is delegated, the agent returns to the delegating manager a *capability right* to apply mad protocol operations to the delegated DMP object. A manager may make copies and further distribute the capability right to instantiate DMPs to other processes, e.g., other manager processes and even delegated DMPIs can obtain the right to instantiate another DMP.

### 3.5 Controlling the Execution of a DMPI

The execution of DMPIs procedes independently of the execution of the delegating manager. However, a manager may need to reta[Bin control over the execution of a DMPI.

For example, a manager may instantiate a DMPI that monitors and reports periodically some statistical value which is a function of certain managed object attributes over some period of time. At certain times, the manager may wish to suspend this reporting to release resources (eg cycles) used by the DMPIs. This will enable the agent to allocate the freed resources to other critical problems, and the manager to reduce the amount of information that it receives from the DMPI, thus releasing some of its own resources. Once these problems are handled, and resources become available reporting should be resumed.

MAD provides a protocol and agent services for DMPI execution control that enable such functionality. A DMPI is an integrated unit of execution over which the agent's environment can exercise control: it can be instantiated, suspended, resumed and terminated. Also, a DMPI may request to be awakened after a specified period of time.

### 3.6 Delegation and Access Rights

A DMPI must have some means to interact with managed objects in order to obtain information and to control the managed objects. DMPIs obtain controlled access to managed objects attributes thru explicit interfaces provided by *Observation and Control Point* processes. Indiscriminate access to these objects would introduce concurrency conflicts. A DMPI only has access to the attributes of the managed objects which have been explicitly declared as exported, thru an OCP interface. OCPs for the managed objects accessed by a DMP must be allocated and maintained in the environment of the agent to ascertain that accesses by the DMPI to these objects are properly supported.

A DMPI is able to invoke locally the same actions that its creator could invoke remotely (or a subset of them), if it inherits the required access capabilities from its creator. Additionally, DMPIs can send and receive messages between themselves, with their creator and with other processes, by using an asynchronous send/recieve interface (see Section 4.2). For example, a DMPI can acquire the role of receiving event reports from a device which were previously been routed to its creator. In effect, they have all the rights and responsibilities associated with a managerial role, including the right to use the delegation protocol.

## 4 The Architecture of MAD Agents

In this section we describe the software architecture structure of MAD agents. A MAD agent is a combination of a proxy agent and hierarchical manager which implements the runtime support to enable dynamic program delegation and control over the DMPIs. Figure 1 depicts the relationships between a manager, a MAD agent, and managed objects of the network. Figure 2 shows the internal structure of the MAD agent.

The Mad agent includes componenents which are responsible for implementing the delegation protocol storing DMPs, instantiating and controlling DMPIs, providing communications between managers and DMPIs, and providing a concurrent runtime environment where all the DMPIs execute.

The independent components of the Mad agent are: Controller, Delegation Protocol, DMP Repository, Translator, DMPI Server, Name Service, IPC, and Scheduler. Delegated Management Process Instances (DMPIs), and Observation and Control Points (OCPs), execute as independent processes inside the agent. A prototype MAD Agent supporting all the above functionality has been implemented [GT91]. Most of the above components are implemented as one or more independent threads, and full address space processes.

### 4.1 Controller

The *controller* contains the main logic of the Mad agent, that is, a specific algorithm that the particular agent should execute. When the agent is loaded into memory, the controller is the 'main' entry point. It is responsible to initialise the agent's environment, by instantiating all the component processes. Then it loads and instantiates all the load-time DMPs, taken instructions from a given initialisation file. When the agent terminates, the controller is responsible to ensure proper 'clean up', e.g., disposal of resources.

During normal execution the controller plays the role of local manager for the agent. For example, it may decide how to handle requests from managers, which protocols to support, etc. Installation specific policies which are invariant for the lifetime of the Agent are programmed in the controller of an Agent. For example, authentication schemes for managers, handling of exceptional events, etc. The controller may itself use Mad primitives to delegate this responsibility to other processes. In the event of contradicting or conflicting management actions, the controller may establish precedence. In many cases, (and in the default case), there may be no specific logic associated with the controller. In such cases, the controller will only perform the role of initiator and terminator of the agent.

### 4.2 Delegation Protocol

The delegation protocol is an application-layer protocol used by Mad managers and agents to exchange requests. It is implemented by one (or more) dedicated (lightweight) process which provide associations between a manager and the Mad agent and between the manager and its DMPIs. Our current prototype uses TCP or UDP to communicate between managers and agents. An asynchronous IPC implemented over shared memory by the MAD kernel is used to communicate between lightweight processes that execute as part of the Mad agent, and between DMPIs.

The protocol enables a manager to:

1. transfer a DMP to a Mad agent
2. instantiate DMPIs (typically executed as light-weight processes) each with its own instantiation (load time) parameters and its own private state;
3. exercise control over the DMPI's execution, e.g., their execution can be suspended, resumed and terminated by the manager.
4. communicate with the DMPIs to which it has handles to.
5. receive information from the DMPI server about the execution state of the DMPIs.
6. implicitly authenticate itself to the Mad agent, i.e., the protocol will transparently carry authenthication information.

### 4.3 The Repository

The Repository allows storing and retrieving of DMPs from which the DMPIs can be instantiated. DMPs can be stored at agent startup (boot time), or downloaded from a remote manager into the Repository during execution, via the delegation protocol. After receiving a Delegate request, the agent will store the DMP in a repository, and return to the delegating manager (through the protocol) a handle to it (DMPid). The manager can further distrbute this handle, thus enabling other managers to have access rights over the DMP. The DMP handle can be used to request that an instance of the DMP be instantiated as a DMPI, and to request deletion of the DMP from the repository. The repository uses the name service to mantain a mapping between the manager's DMP identifier, and the internal names of the objects used by the actual storage facility (file system).

### 4.4 The Translator

The Translator component of the Mad Agent is responsible to ensure that the DMPs are legal

programs, and to compile them into executable code. Each DMP is represented in the repository as a data structure which contains the actual code of the program, and additional information describing its characteristics and requirements. For example, a DMP will describe in which language it is written, and what is the minimum amount of memory required for instantiating it. DMPs can be in source code format, which may require compilation and linking or interpretation, or as object code that requires only dynamic linking. The Mad Agent uses the Translator for preparing the DMP for instantiation, via compilation and linking. The Translator uses the Repository to store the executable images which are produced by the compilation. If the DMP violates any of a set of defined rules for the given MSL of the program the translator will reject the DMP. For example, the current prototype, restricts DMPs on their ability to bind to external functions. If a DMP invokes some arbitrary external function, it is rejected by the Translator.

### 4.5 Observation and Control Points (OCPs)

An OCP is a process that executes in the Mad Agent environment and defines a set of specific services to enable DMPIs to interact with managed objects. OCPs can be defined in the Mad Agent, and can also be delegated as DMPs, thus providing a programmable management interface to managed objects.

An OCP's service interface might be generic and hide many details of the actual management protocol, and it can also reproduce a given protocol interface. They can serve as protocol gateways between different management protocols. OCPs provide a process representation of managed objects, trapping events and signals, and making them accessible to multiple DMPIs. They provide concurrency control in the access of DMPIs to managed objects, by serialising their requests.

An OCP can report the current status of a managed object and can forward DMPI requests to the object, by invoking an available protocol interface to the managed object. Such protocol can be device specific (i.e., it becomes a proxy agent) or a management protocol (e.g., SNMP, CMIP). OCPs can locally store the values of attributes, caching frequently used information. Finally, OCPs provide granularity of access control, e.g. a given DMP can obtain bindings to a specific OCP but not to others.

A sample OCP implemented with the Mad prototype [GT91] uses the SNMP protocol to access network objects. It also provides a translation facility between SNMP and manager-defined name spaces, and it provides concurrent access to the DMPIs.

### 4.6 Delegated Management Processes Instances

A process DMPI is instantiated from a DMP template, via the Mad_Instantiate call. Each DMPI has its own private state and can have different initialisation parameters. For example, two DMPIs may perform some accounting task using the same algorithm (DMP) but different parameters (resources being measured).

A typical DMPI will perform certain initialisation steps, request services from OCPs and from the Mad kernel, and then 'sleep' until 'awakened' by some event, usually the arrival of a message. It will not poll for messages, but will (implicitly) yield the CPU when waiting for a message. Typically, most of their time is spent in a suspended state, resting between bursts of activity. They will periodically collect information, analise it and possibly take some action as a result. If they are not equipped to fully handle the given situation they can send a status report to the manager processes or other DMPIS to request assistance.

DMPIs have access to the attributes of the managed objects that have been explicitly made available via protected programming interfaces (the Mad API) provided by the Kernel and those provided by OCPs. Since the Kernel's scheduler provides pre-emptive scheduling, DMPIs can not monopolise the CPU.

### 4.7 Agent Kernel

The Kernel implements several services, which together provide an execution environment for DMPIs. It is divided between the following components: Scheduler, DMPI process manager, IPC and Naming. The Scheduler provides pre-emptive multitasking for the DMPIs, and a wakeup function that activates a suspended thread after a specified interval. The DMPI process manager

allows to instantiate, kill, suspend, resume, retrieve, and modify the status of DMPIs. The IPC facility, supports asynchronous local communications between DMPIs and OCPs, over shared memory. The name service provides a private, protocol neutral way to associate names with managed objects attributes, OCPs and DMPIs.

## 5 Management Scripting Languages

A Management Scripting Language (MSL) is used to encode management programs to be delegated from managers to agents. In this section we describe some of the requirements for MSLs.

Delegated management programs must be compatible with the computational capabilities that are supported by the respective agent's software environment and the processors in which they execute. The computational resources that are available for management purposes vary greatly among networked systems and devices. Small devices, such as modems or multiplexors, will typically offer very limited computational capabilities for management purposes. In contrast, agents controlling switches or large computing systems may be able to afford substantial resources in order to support extensive manageability.

In addition to the intrinsic limitations of the available hardware, users of the management system may want to allocate these resources according to installation-specific parameters. Administrative policies may impose additional restrictions in the allocation of management resources. For example, any user may be able to query the liveness of a given device, but only a few may be allowed to reset it. Thus, different manager processes should be allowed to perform different subsets of the management operations supported by the device.

The combination of a wide variety of computational capabilities in managed devices over a distributed system, and the administrative considerations regarding access to those resources, yields a highly heterogeneous management environment. To overcome this inherent complexity, writers of management applications require tools that can help take advantage of this heterogeneity. A comprehensive management system environment must therefore provide facilities for a spectrum of possible manager requirements and agent capabilities.

The need to provide such restrictions has been recognised in working implementation agreements for management protocols [IEEE90] by defining protocol subsets for CMIP. Association-types are used to define a restricted subset of the CMIP protocol over a particular application layer association. Thus, a pair of processes, – one in a manager role, the other in an agent role – can establish an association of a given type in order to restrict the type of messages that can be exchanged. For example, an association of type *event* restricts agents to only report events over that association. Thus, association-types allow a manager-agent relationship to specify a computational restriction indirectly, by specifying a protocol restriction.

In contrast to the above protocol-oriented restrictions, the Mad paradigm enforces computational restrictions via sub-languages or types of MSLs. For example, a management program can be specified in a sub-language that restricts the program to read-only access to managed objects attributes. Thus, a management script can be written in an MSL that is a subset of a general-purpose programming language, (e.g., C, Pascal).

A management script written in such an MSL will be processed as follows. The management script is transferred from the manager to the agent via the delegation protocol. In the header of the message, there is a field that specifies the MSL identifier for this script. If the manager process is authorised to delegate scripts in the given MSL, then the program is analysed by a Mad pre-processor for the given MSL. For example, the pre-processor will check that there are no assignment to non-local variables in the script. The output of the pre-processor, if found to keep the constraints specified by the sub-language, is compiled with the source's programming language compiler. The object code generated by the compilation is then stored in the agent's repository. Thus, a Mad agent can use a combination of compile time pre-processing and runtime authentication to enforce specific sub-language constraints.

The MSL used in the current prototype [GT91] is a subset of C, which restricts the capability to invoke arbitrary external functions. Thus, a DMP can only invoke external functions made

available in a Mad Application Programming Interface (MADAPI) library. This library enables delegated management process instances to request services from the Mad kernel; control the scheduling execution of the processes; exchange messages with other DMPIs, OCPs, and manager processes; and receive notifications of events as messages from the Mad kernel.

Although general-purpose languages like C are sufficiently expressive, they can expose the Mad agent instance to hasards that may be unacceptable in a reliable management application (for example, overwriting unintended memory locations within the shared address space, memory leaks, non-reentrant code, etc.). The current prototype MAD agent employs a variety of runtime protection schemes to reduce the likelihood of these events and to recover from them. However, a sound solution to these problems must use compile-time analysis techniques to ensure safety.

### 5.1 Special Purpose MSL

One of the fundamental questions that we must answer in order to provide effective management capabilities is: what is an appropriate paradigm in which to express management specifications? Management applications need specialization in order to overcome the huge complexity involved in distributed systems. Thus, we must raise the level of abstraction of the programming interface in which to encode management scripts. We will do that by providing a very high-level special-purpose management scripting language with special primitives to simplify the encoding of such an application.

From an implementation perspective, these new primitives can be considered macro-extensions that can be compiled into any general-purpose programming language (e.g. C), or they can be interpreted by the agent. For example, they could provide the ability to specify management operations as temporal operators over traces [GKY90].

## 6 Models for Manager/Agent Interaction

In this section we review existing models of process interactions, and evaluate their suitability for resolving the micro-management problem.

The remote procedure call (RPC) interaction paradigm [BN84] is widely used for structuring distributed systems. In this model, a server exports a number of fixed procedures that can be invoked by remote clients. Upon a remote call, the caller is suspended, the remote procedure is executed, and its results are returned to the caller, which then resumes its execution.

It is natural to ask whether the RPC mechanism is sufficient to solve the micro-management problem. Let us examine this question via the example 1. Suppose that the entire management program shown there is encoded, as part of the agent code, in terms of a particular action – say, *link.handler*. A manager will need to execute a remote procedure call invoking execution of the script by the agent. Does this solution offer an adequate model for management?

There are two fundamental problems associated with the solution above. The first is rooted in the synchronous invocation semantics of RPC mechanisms. A management script will typically involve actions to be triggered by independent event occurrences in the agent, asynchronously with the manager's invocation of the RPC. Clearly, the manager's own execution should not be tied with the execution of the management script. An RPC mechanism would block the manager until the completion of the manager script.

A second limitation of this model for supporting manager/agent interactions is the static nature of the exported procedures. That is, there is an underlying assumption that the programming of management scripts is done by the agent/managed-object designer at agent-design time (just as server calls are part of server design), rather than at management design time as part of incorporating the agent/managed-objects in a network. Designers of the agent (server) must thus predict and code as agent procedures the entire range of composite management scripts in which this agent may be usefully involved. It is typically not possible to predict at agent-design time all the possible management scenarios in which it may be involved, nor is this desirable, as it would limit the manageability afforded by the product. Furthermore, coding such scripts into the agent would significantly increase its complexity and costs.

Remote evaluation (REV) [SG90] is another in-

teraction paradigm for processes in distributed systems. It allows a procedure (written in a LISP[A dialect) to be trasfered from a client to a server where an interpreter will execute it. It thus overcomes the problem of fixing all the server procedures that can be invoked remotely at the design time of the server. However, the execution of the remotely evaluated procedure is still synchronous with the execution of the caller (manager): i.e., the REV procedure is executed upon its receipt by the server (agent).

In summary, the RPC model fails to provide an adequate solution of the micro-management problem, as agent designers must predict and code all management scripts required as part of the agent. The remote evaluation model goes a step further in permitting the dynamic transfer of programs to remote execution. However, both models involve synchronous procedure-call interactions whereby the agent executes the management program upon invocation, while the caller is blocked until completion. Thus neither technique proves to be adequate for management in a distributed environment.

## 7 Status

The MAD prototype [GT91], has successfully demonstrated the delegation model of network management. It can receive, compile, link, and execute complex management scripts from remote manager processes. The scripts can execute asynchronously from their delegating manager and take independent action, eliminating the micro-management problem. Several sample DMPs have been written and tested over the prototype.

The prototype includes the following components:

1. The MAD Kernel: Includes a preemptive scheduler, a message-passing IPC service over shared memory, a name service, and a DMPI process manager.
2. A simple Mad Agent Controller
3. An implementation of the delegation protocol, using ASN.1 presentation services [Rose90] and TCP/IP transport services.
4. A DMP repository: A storage and retrieval facility for delegated programs.
5. A Translator supporting DMPs written in C and C++.
6. An SNMP based OCP.

## 8 Conclusions

Current management models, as pursued by standardisation efforts, centralise all management functions within managers with agents playing minor roles. This centralisation induces managers to micro-manage agents, resulting in inefficient and costly management systems. It also results in intrinsically unreliable network management, since managers are turned into sensitive communications and processing bottlenecks whose loss can result in paralysis. Centralised management is neither commensurable with the management needs of emerging networks (e.g., managing high-speed increasingly complex networks), nor is it reflective of emerging cost/performance trade-offs. Network devices and their agents may require and support as powerful processing capabilities as manager applications.

Effective management requires support for flexible, efficient distribution of management functions and allocation of resources. The Manager-Agent Delegation (MAD) model proposed in this paper, supports such a distribution. It provides a software framework that enhances the expressive power of management interactions. A manager can delegate an entire management procedure to be executed in the MAD agent's environment, while keeping control over its execution. These management programs can then execute locally at the agent without involving the manager unnecessarily. The capabilities and limitations of agents may be reflected in the management scripting language that they use.

From a performance perspective, delegation presents an attractive trade-off. It allows lower communications costs (of management messages) by dynamically allocating management function closer to the real devices. Of course, flexibility and expressiveness do not come for free: they require more computing cycles and memory in the MAD agent environment. These costs are divided into two parts: a fixed component (the built-in Mad support) and a varying component (the OCPs and DMPs). There are two reasons

to justify this cost. First, consider the increasing costs associated with mismanagement of networks, due to our increasing reliance on them. On the other hand, there is a continous decrease in the cost of hardware (cycles), and an increase in their abundance in distributed environments is expected. Thus, we advocate increasing the degree of manageability at the cost of allocating – intelligently – more resources over the network.

## ACKNOWLEDGMENTS

The current prototype of Mad was implemented by German Goldszmidt, Antonis Maragkos, James Tuller and Chris Wood, in the DCC Laboratory of the Computer Science Department of Columbia University

## REFERENCES

[Ada82] Reference Manual for the Ada Programming Language, D.O.D, Ada, Joint Program Office, July 1982.

[BN84] Birrell A. and Nelson P., "Implementing Remote Procedure Calls", ACM TOCS, 2-1, Feb. 84.

[CMIS89] ISO, Information Processing systems, Open System Interconnection, Management Information Service Definition Part 2: Common Management Information Service Element, December 1989.

[GKY90] Goldszmidt G., Katz S. and Yemini S., "High-Level Language Debugging for Concurrent Programs", ACM Trans. on Computer Systems, November 1990.

[GT91] Goldszmidt G., and Tuller J., The Design and Implementation of the MAD System: Project Report", Computer Science Dept., Columbia University, Working Draft, 1991

[GY91] Goldszmidt G. and Yemini Y., "The Design of a Management Delegation Engine", Proceedings of the IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Santa Barbara, California, October 1991.

[IEEE90] "Working Implementation Agreements for Open Systems Interconnection Protocols", Boland, F.E., Editor, IEEE Computer Society Press,Vol. 2, Number 2., 1990.

[IFIP89] "Proceedings of the IFIP TC6/WG 6.6 Symposium on Integrated Network Management", North-Holland, B. Meandzija and J. Westcott (eds.), May 1989.

[ISO89] Information Processing Systems, Open Systems Interconnection, Systems Management Overview, December 1989

[KMW90] "Network Management and Control", 1990 Plenum Press, Kershenbaum A., Malek M. and Wall M., editors, proceedings of the Network Management and Control Workshop, September 1989.

[Rose90] Rose M.T., "The Open Book, A Practical Perspective on OSI", Prentice Hall, 1990.

[Rose91] Rose M.T., "The Simple Book, An introduction to Management of TCP/IP-based Internets", Prentice Hall, 1991.

[SG90] Stamos J.W. and Gifford D.K., "Implementing Remote Evaluation", IEEE Transactions on Software Engineering, Vol 16, No 7, pp 710-722, July 1990.

[Sun88] Sun MicroSystems, "Systems Services Overview", Part Number: 800-1753-10, May 1988

[TR88] Tanembaum, A.S. and van Renesse, R., "A critique of the Remote Procedure Call Paradigm", Research into Networks and Distributed Applications (Edt. by R. Speth), Elsevier Science Publishers B.V. (North Holland), Apr 1988.

[YGY91] Yemini Y., Goldszmidt G. and Yemini S., "Network Management by Delegation", The Second IFIP International Symposium on Integrated Network Management, Washington, D.C., April 1991.

## About the authors

German Goldszmidt received his B.A. (1985) and M.S. (1988) degrees in Computer Science, both from the Technion, Israel Institute of Technology. Currently, he is a Ph.D. candidate in Computer Science at Columbia University. Since 1988 he has been working at IBM's Watson Research Center. His research interests include distributed computing systems, network management, and debugging of concurrent programs. He can be reached at the address Computer Science Dept, Columbia University, New York, NY, 10027. His Internet addresses are german@cs.columbia.edu and gsg@watson.ibm.com.

Yechiam Yemini is an associate professor at the Columbia University Computer Science Department. His current research interests include network management, high-speed networking, protocol design, performance analysis, and neural networks. He can be reached at the address Computer Science Dept, Columbia University, New York, NY, 10027. His Internet address is yemini@cs.columbia.edu.

Shaula Yemini received her BSc Degree in Mathematics and Physics in 1972 and her MSc degree in Applied Mathematics in 1974 both from the Hebrew University in Jerusalem. She received her PhD in Computer Science from UCLA in 1980, majoring in programming languages and systems. In 1982 she joined the IBM T.J. Watson Research Center. Dr. Yemini's research interests include programming languages, semantics, exception handling, language constructs for concurrency and distribution, debugging for distributed systems and fault tolerance. She can be reached at the address IBM Research, P.O.Box 704, Yorktown Heights, NY 10598. Her Internet address is yemini@watson.ibm.com.

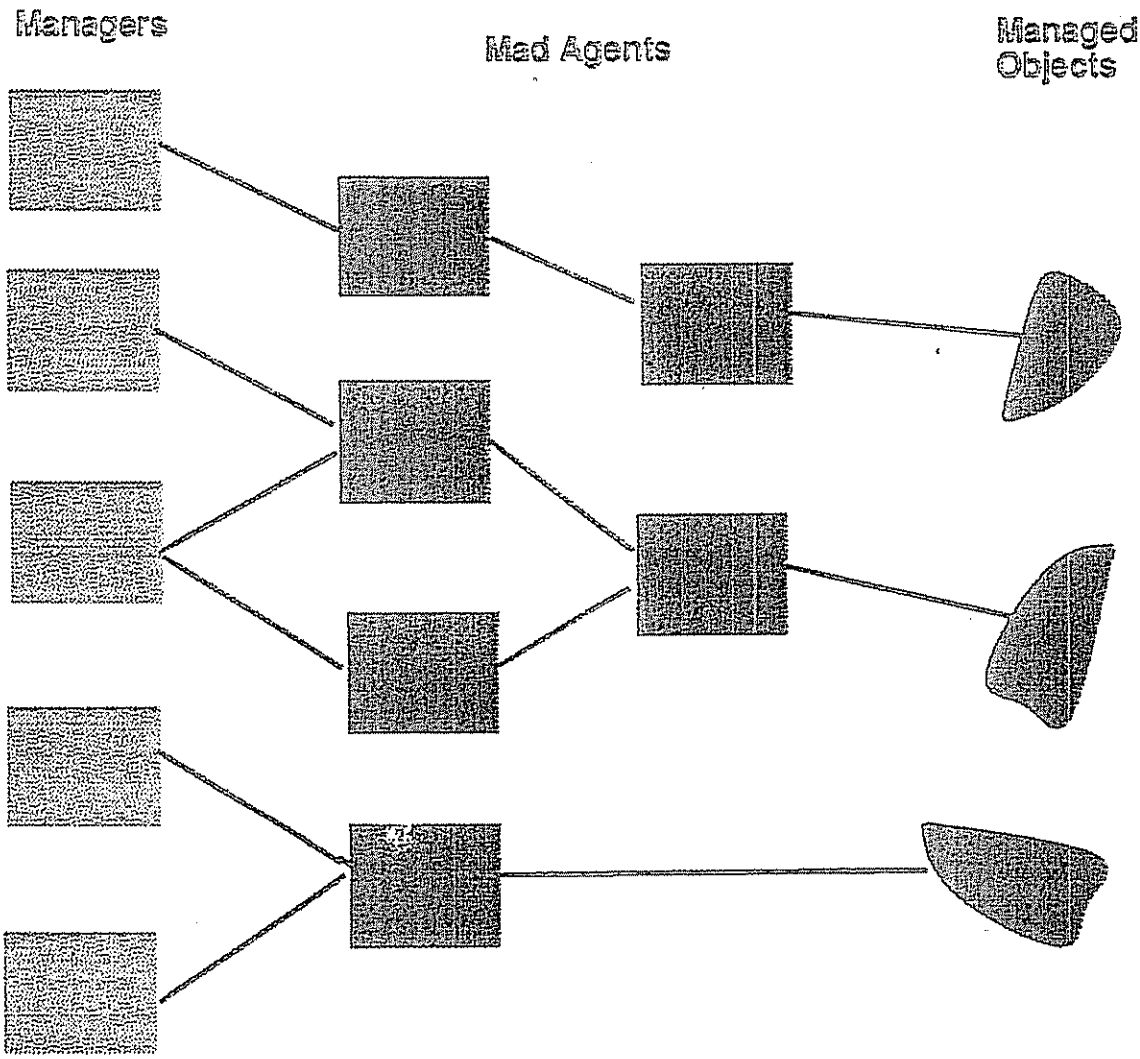



Figure 1: MAD Managers and Agents

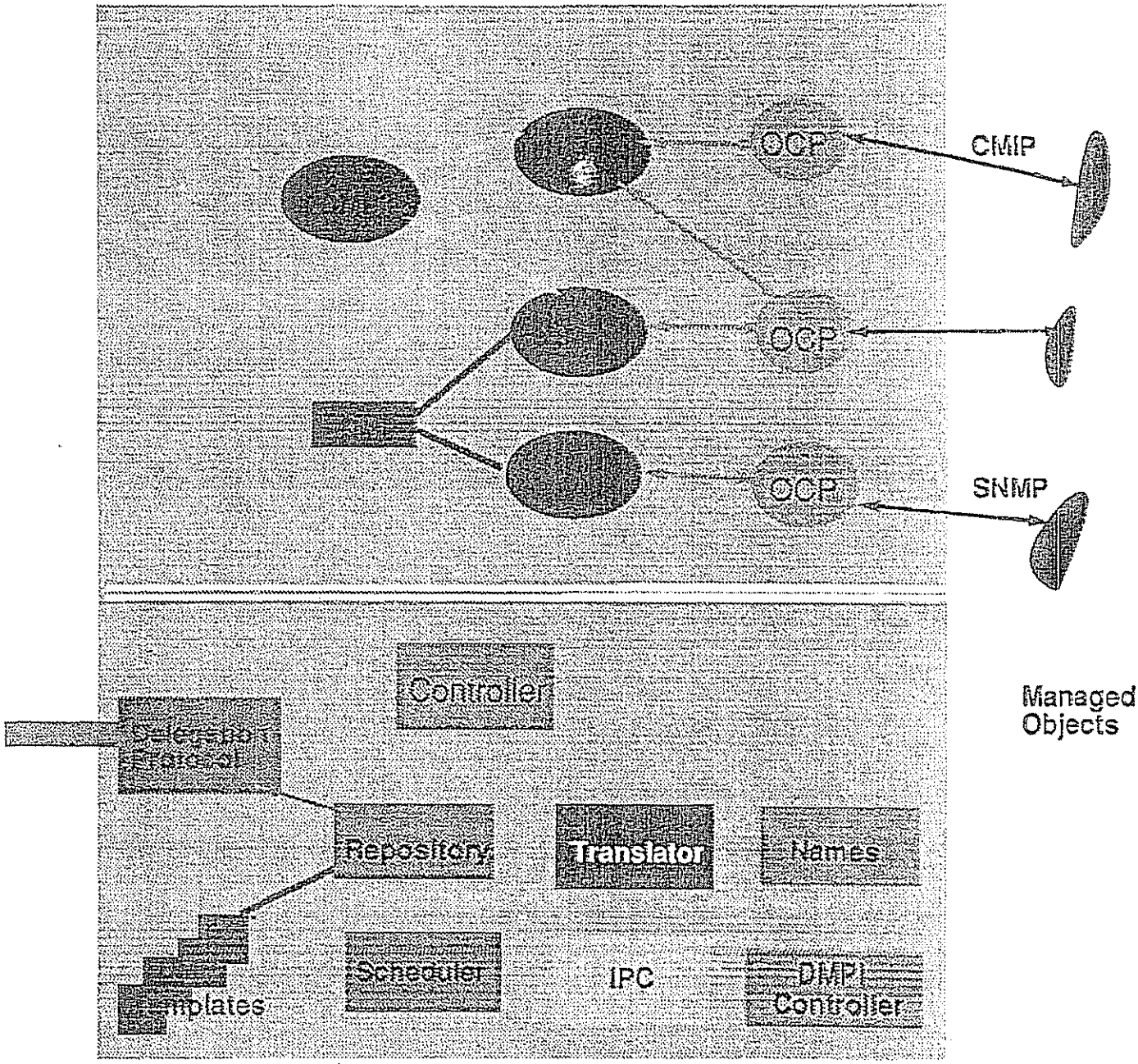



Figure 2: Mad Agent