F.Stamatelopoulos, B.Maglaris, "Performance and Efficiency in Distributed Enterprise Management", Journal of Network and Systems Management, Special Issue on Enterprise Network and Systems Management, Plenum Publishing Corporation, Vol. 7, No. 1, March 1999.

# Performance and Efficiency in Distributed Enterprise Management

F. Stamatelopoulos, B. Maglaris

Network Management and Optimal Design (NETMODE) Lab

Department of Electrical and Computer Engineering

National Technical University of Athens

Iroon Politechniou 9, Zographou, 157 80 Athens, Greece

tel.: (+301) 772.1448 fax: (+301) 772.1452

e-mail: fotis@netmode.ntua.gr, maglaris@netmode.ntua.gr

**Abstract**

This paper is motivated by the increasing need for scaleable, distributed management architectures for integrated network, system and application management within the enterprise network environment. Such integration, extension and wide area deployment of management functionality impose heavy performance requirements, in direct conflict with the increased volume of management traffic produced. Aiming to minimize this traffic and the overall response time, we propose a distributed hierarchical caching scheme that attempts to take advantage of the diverse consistency requirements of management applications. We define coherency conditions and update policies, we identify the appropriate interaction semantics and discuss an SNMP-based implementation. In order to evaluate the proposed model and to quantify the expected performance gains we construct a simple queuing model that provides analytical results on the improvement of response time and the reduction of management traffic. Finally, the analysis of experimental results provides some insight on performance improvement for specific classes of managed objects.

**Keywords:** integrated enterprise management, distributed management architectures, information caching, and management protocols.

## 1. Introduction

During the last five years, following the exploding expansion of networking infrastructure, enterprise networks have grown both in size and complexity introducing management requirements of increased reliability, performance and scaleability. The trend towards practical distributed and hierarchical management architectures has become more pressing and many research groups [7], or international organization working groups, have produced results to this goal ([10],[16],[8],[11],[12] to reference a few). Support for such architectures is inherent to the

OSI management framework, but even the IETF management framework that was designed with a "simple" and light-weight philosophy has grown to support multiple level architectures through the introduction of the mid-level manager concept [17] and delegation [6] through scripting [4]. Significant work on distributed management issues is conducted by various IETF working groups and especially within the DISMAN working group (http://www.ietf.org/html.charters/disman-charter.html). In addition, a lot of research work is focused on system, application & service management, aiming to their integration into a unified management framework. The need for distributed management architectures in these areas grew due to the increase of network dependencies in modern computer systems and the wide adoption of highly distributed applications and services, like the Internet Information Services [14][3][9].
Distributed architectures for integrated enterprise management should span over a wide area scale and handle huge volumes of information that flows through both LANs and WAN links. Delays and faults, as well as the management traffic imposed on the underlying networks, are major concerns and factors that affect the efficiency and cost-effectiveness of such frameworks. In this paper, we present a caching model, based on the concept of allowing copies to diverge in a controlled fashion, that can be applied to distributed management architectures and aims to improve overall performance by reducing response time and management traffic. We define specific coherency conditions that can be used for caching managed objects with different consistency requirements, throughout a hierarchy of management components that act as a management information client, server, or dual role server.

The remainder of the paper is organized as follows: Section 2 presents the proposed caching model; the application of caching in distributed management is discussed, selection and coherency conditions are defined and the associated caching update policies are presented in detail. Section 3 discusses implementation specific issues: the proposed architecture is outlined, client and server roles are identified within a multiple level hierarchy, interaction semantics are defined in the form of a generic message-based application protocol and an SNMP-based implementation is briefly discussed. Analytical and experimental results are presented in section 4 in an attempt to quantify the expected gains of the proposed scheme. Finally, section 5 concludes and presents current and future work objectives.

## 2. A caching model for distributed enterprise management

### *2.1 Caching in management*

The manager-agent paradigm is in essence a client-server model: the agent acts as a data server providing management information to the client/manager through atomic queries realized via a specialized management protocol and application interface. Any management framework consists of management components that either produce or consume management information provided by other entities. Distributed and hierarchical management architectures rely on multiple dual-role manager layers that span over wide area scale and utilize mostly WAN connections for inter-manager communication. Such hierarchies produce management traffic that may influence significantly the overall performance and efficiency of the management system. Further, the system response time and the output data validity is tightly associated with the faultless and efficient propagation of a large volume of management information between management components.

The main drive of the work presented in this paper is the application of well established data management techniques, like information caching, to integrated (network, systems and application) management in order to improve the overall efficiency in terms of response time and

management traffic. An important observation that lead us towards caching techniques, is that different management functions and layers in a management hierarchy tolerate a wide range of data consistency levels. For example, a planning application or a high-level manager that collects overall performance statistics, may operate on more relaxed data consistency as far as information age is concerned or may not require the exact temporal behavior of the managed object: an outline, sparse sampling of the object may be enough, as long as no important changes are missed. On the other hand, a fault management application may impose stricter coherency conditions and require up-to-date and detailed information. Caching techniques can take advantage of different consistency requirements, in addition to eliminating multiple requests for the same data. Combined with on-demand retrieval techniques for replacing polling with asynchronous updates caching provides an attractive solution for minimizing management traffic and overall response time.

In the following paragraphs, after a brief description of the quasi-caching concept we present the proposed model, the coherency conditions and associated caching update policies in detail. These are implementation independent concepts that do not pertain to any specific manager structure or management protocol; the only assumption is a multiple manager architecture and the manager-agent paradigm.

### *2.2 The concept of quasi-caching*

The notion of *quasi-copies* and *quasi-caching* was introduced by Alonso et. al in [1] and it is based on the notion of allowing cached object images to diverge from the actual object in a controlled way. Quasi-caching permits the client to define weaker or stricter consistency conditions between the central data and its local images depending on specific application requirements, thus establishing the exact degree of coherence desired in a per case basis. This allows the client to minimize caching and copy maintenance overheads and benefit from the advantages of caching without sacrificing data consistency.

Quasi-copies are characterized by two types of definition conditions, introduced in [1]: *Selection conditions* define a unique set of objects that will be cached at the client, while *coherency conditions* allow the client to establish the desired level of consistency between the actual object and its cached image. The coherency conditions specify the allowable deviations for the images of the objects identified by a selection condition and they are enforced only when the image exists. A selected object may be associated with one or multiple coherency conditions, allowing to satisfy any client requirements.

### *2.3 Selection and coherency conditions*

The selection and coherence conditions adopted are derived by the ones presented in [1]; some are applied as is, while others are modified or adapted to meet specific requirements of a management application framework. A selection condition consists of two parts; the *identification clause* that specifies the objects to be cached, and the *modifiers* that determine how the selection will operate. The selection identification clause is quite implementation specific, since it depends on the syntax and naming scheme used by the agents and managers; for example a CMIP or SNMP management framework will use ASN.1 expressions for specifying objects in the identification clause of selection conditions. In general, the identification clause uses logical expressions within the limitations of a given syntax for specifying sets of objects, and it is complemented by a group of modifiers that describe the selection action. The proposed model uses the following two modifiers:

(a) *Action modifier [add/drop]*. Specifies whether the selected object images will be added or removed from the client's cache.

(b) *Enforcement modifier [compulsory/advisory]*. Defines compulsory or advisory caching. The management information server and clients will guarantee providing the necessary updates for enforcing the coherency conditions. An advisory selection, on the other hand, does not guarantee that the server will propagate the updates and consequently the client image may not exist at all times or may not be valid. Non critical or less frequently accessed objects may be selected with this modifier in order to allow the server to defer updating them due to performance or other internal reasons.

Each selection of objects is associated with a set of coherency conditions that specify the desired consistency. The following conditions may be used for any selected object, as a single requirement or in combination (with the AND, OR and NOT operators):

(a) *Default coherency condition*: the image *x'* of the actual object *x* should have, at all times, a value previously held by the actual object, i.e. $\forall t \geq 0 \;\; \exists t_0 : \; 0 \leq t_0 \leq t \;\wedge\; x'(t) = x(t_0)$. The default coherency condition must hold implicitly and be enforced for all cached objects.

(b) *Delay condition*: a valid image may lag behind the actual object up to a given delay *d*, i.e. $D(x) = d \;\; \equiv \;\; \forall t \geq 0 \;\; \exists k : \; 0 \leq k \leq d \;\wedge\; x'(t) = x(t-k)$.

(c) *Periodic condition*: image *x'* matches *x* at time $t_0$ and is refreshed every *d* time units, i.e. $P(x) = t_0, d \;\; \equiv \;\; \forall t \geq 0 \;\; \exists n : \; n \geq 0 \;\wedge\; t_0 + n \cdot d \leq t \leq t_0 + (n+1) \cdot d \;\wedge\; x'(t) = x(t_0 + n \cdot d)$

(d) *Version change condition*: the image may fall behind up to a max. number of versions, i.e. $V(x) = n \;\equiv\; \forall t \geq 0 \;\; \exists k, t_0 : \; 0 \leq k \leq n \;\wedge\; 0 \leq t_0 \leq t \;\wedge\; v[x(t)] = v[x(t_0)] + n \;\wedge\; x'(t) = x(t_0)$

where *v[x]* is the version of object *x*, i.e. the number of modifications since object creation.

(e) *Absolute value change condition*: the image is valid if its deviation is below a threshold, i.e. $AC(x) = e \;\; \equiv \;\; \forall t \geq 0 \;\; \left| x'(t) - x(t) \right| \leq e$

(f) *Percentage change condition*: the maximum allowable deviation is defined and evaluated as a percentage change, i.e. $PC(x) = e \quad \equiv \quad \forall t \geq 0 \;\; \left| \frac{x'(t) - x(t)}{x(t)} \right| \leq e$

### *2.4 Caching update policies*

The enforcement of the coherency conditions is handled through specific caching update policies that govern the interaction of management information clients and servers within the proposed management caching scheme. Two categories of policies are defined:

(a) *Lazy policies* defer the propagation of updates until the last minute, regardless of the possibility that the coherency conditions may be violated between accesses; cached copies are not updated until they are accessed. Since only the maintainer of the image is able to sense when an update is required, it is the client who is responsible for initiating the condition evaluation and possible generation of updates. Three lazy update policies are defined:

- *update on demand*: The image of the object is always updated before it is used. The client forces an update when the image is accessed. Applied for the condition *D(x) = 0*.
- *update on demand with latency*: A *d>0* latency is tolerated. Condition *D(x) = d > 0*.
- *lazy update on value deviation*: The client generates a conditional update on demand and the server responds with an update if the condition is violated. *AC(x) = e* or *PC(x) = e*.

(b) *Continuous policies* propagate the updates as soon as the coherency condition is violated. The server generates updates for invalidated objects even when the images are not accessed.

- *periodical update*: This is, in essence, a continuous polling with a predefined period. Updates are generated periodically in order not to violate the condition $P(x) = t_0, d > 0$.
- *update on value deviation*: Updates are generated by the server when the object deviates from the last update beyond the acceptable limit. Suitable for *AC(x) = e* or *PC (x) = e*.
- *update on version change*: Updates are generated when *V(x) = n* is violated.

# 3. Implementation issues

### *3.1 Architecture overview*

Figure 1 outlines the overall architecture as perceived from the data management point of view. The components are organized in multiple levels depending on their position in the client-server chain. A component - be it an agent, manager or mid-level manager - acts as a management information server if it produces data offered to higher levels and as a client if it consumes data from lower levels. At the lowest level (level 1), servers scan the managed systems (or network elements, applications, services, etc.) collecting information and implementing a set of managed objects represented in a specific management information model. Level i (i > 1) servers retrieve information from lower level servers through the update policies discussed defined earlier in the text. In general, a management information server belongs to $i^{th}$ level, *where i = 1+ max[all levels of other servers it accesses as a client]*.

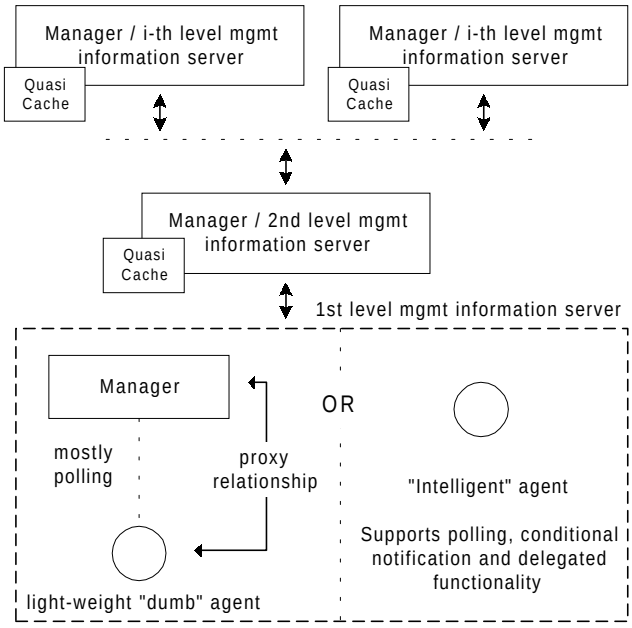


**Figure 1: Architecture overview and actors**

The first-level management information server is either a conventional light-weight agent with a proxy manager or an "intelligent" agent. In the first case, it is the proxy-manager that implements the proposed caching scheme and update policies, while in the second case, the agent supports built-in native support for the proposed caching policies (polling, asynchronous notification and delegated functionality). Higher level servers are implemented as conventional managers with the addition of a persistent data cache system.

### *3.2 Client-server interaction*

We defined an abstract, implementation-independent interaction protocol that captures the communication semantics in the form of messages exchanged between the manager components. Note that these interaction semantics may be implemented with either SNMP or CMIP (or any other management protocol that supports the manager-agent or client-server model). These generic messages provide a guideline for implementing the proposed model within any management framework that relies on a client-server (manager-agent) scheme.

The following client messages are generated for definition and configuration purposes, activation-deactivation of object selections and for requesting updates on special cases:

- *DefineSelection (selectionID, selection_condition, action[add|drop], enforce[compulsory|advisory])*. Allows the definition or modification of a selection. An initial message with *selectionID = NULL* is issued for obtaining a valid ID from the server, and subsequent messages may be used to refine the selection (add/drop actions).
- *DefineCoherency (selectionID, coherencyCondition, parameter, logicalOperator)*. Used for "attaching" consistency conditions to a valid selection. At least one such message must be sent to the server before activating the selection. The default coherency condition is implied

and additional conditions are added with subsequent messages through the *coherencyCondition*, *parameter* and *logicalOperator* (*NULL*, *AND*, *OR*, *AND_NOT*, *OR_NOT*):

| ***coherencyCondition*** | ***parameter*** | ***condition specified*** |
|---|---|---|
| automatic | automatic | default coherency condition |
| 1 | delay | D(x) = delay |
| 2 | period | P(x) = current time, period |
| 3 | # versions | V(x) = # versions |
| 4 | value | AC(x) = value |
| 5 | percentage | PC(x) = percentage |

- *Activate (selectionID).* The client activates caching for a defined selection.
- *Deactivate (selectionID, purgeFlag).* Deactivation and deletion if *purgeFlag=TRUE.*
- *ForceUpdate (selectionID).* Force an update for a valid selection.
- *ConditionalUpdate (selectionID, $oid_1$, $v_1$, $oid_2$, $v_2$,..., $oid_n$, $v_n$).* Supports lazy updates on selections that are characterized by the *AC(x) = e* or the *PC(x) = e* conditions. Issued by the client and evaluated by the server.
- *KeepAlive (period).* Request the server to periodically sent "hello" messages. If no messages are received within the *period* the client assumes that connection to the server is lost.

Server messages provide status information and acknowledgment, or propagate updates:

- *Acknowledge (statusCode, data).* All messages received by the server are acknowledged. Acknowledgment is "piggy-backed" with status/error codes and return data.
- *Update (selectionID, $oid_1$, $v_1$, $oid_2$, $v_2$,..., $oid_n$, $v_n$).* Updates a cached selection of objects.
- *Invalidate(selectionID).* Invalidates a selection or its subset.
- *ServerUp() - ServerDown().* Sent to all clients after-before a server recovery-shutdown.
- *Hello().* A "keep alive" message sent to the clients that request it through *KeepAlive().*

Figure 2 sketches a client-server interaction example where the definition and activation of a cached selection is completed without errors. After attaching a coherency condition, the client activates the caching procedure and the server starts propagating updates for enforcing the coherency conditions. Updates will propagate continuously until the management server crashes or the client deactivates the selection.

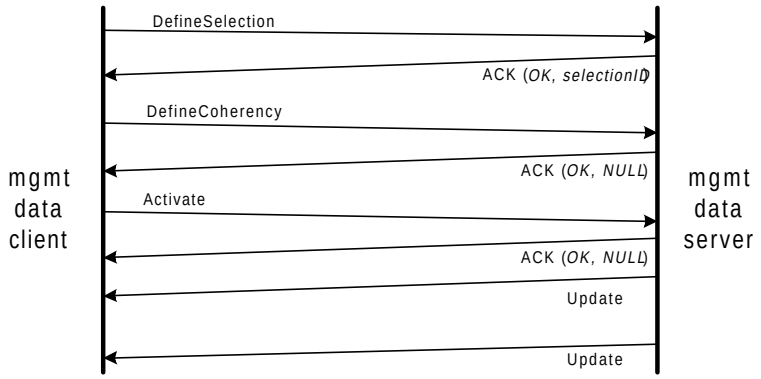


**Figure 2: Interaction example - defining and activating a selection**

### *3.3 Responsibilities and overheads*

The responsibility of evaluating the coherency conditions and generating updates depends on the adopted coherency condition and update policy. Lazy policies force this responsibility to the client since the condition violation is associated with the client object access patterns. When the *D(x)=d* condition is violated then it is the client that will send a *ForceUpdate* message to the server requesting a fresh object instance. Lazy update on value deviation is an exception, since both client and server participate in the evaluation of the condition: when the object is accessed, a *ConditionalUpdate* is issued to the server for evaluation and possible generation of an update.

Continuous policies, on the other hand, are handled by the server; the server evaluates the condition (*AC(x) = e, PC(x) = e, V(x) = n* or *P(x) = t0, d*) and generates the appropriate updates. If the object is associated with multiple coherency conditions, then the conditions are evaluated in a specific order of priority: conditions that lie in the responsibility of the client take precedence.

Responsibility comes with a price: storage and processing overheads are imposed to the entities responsible for evaluating coherency conditions. The periodical condition requirements in terms of storage and processing are minimal; a priority queue with client addresses, the object IDs and the polling period is enough for implementing the associated update policy. The delay condition is also quite simple, since only the last update time must be attached to cache image. All other conditions need to keep track of the current image (last update) in order to make evaluation feasible at the server side without consulting the client, a need that may impose significant storage requirements on heavily accessed servers. All these data, including the selection and coherency definitions, must be kept in persistent storage so that recovery is possible. Consequently, management information servers must be built around an object storage system that will implement an efficient and persistent cache (we present such an architecture in [13]).

### *3.4 Delays and faults in updates*

Client-server connectivity faults result to data inconsistencies at the client sites that may propagate upwards through the hierarchy with catastrophic results. Thus, the client must be able to early sense when the connection to the server is lost and notify higher level entities. The "keepalive" mechanism (*KeepAlive() - Hello()* primitives) allows the client to define a maximum time period after which loss of connection to the server is sensed: no messages received during this period and the client assumes that the server is unreachable and invalidates all cached images accordingly (objects with the *D(x) = d* condition may be considered valid for a brief time after loosing communication with the server). Since, unexpected network delays may bring the client to such a state, it is advisable to provide a "keep-alive" period larger than the maximum delay or periodical update parameter.

### *3.5 Propagation and invalidation of data*

An important implementation issue is the propagation of updates within the hierarchy. Single level caching of an object is quite straightforward, as presented in Figure 2. Problems arise when the client caches an object that is already an cached image. Consider for example, an object *x* that is produced at level *i* and its copy *x'* is cached at a server at level *i+1*. A level *i+2* server caches the *x'* object from server *i+1* (with a local image *x''*) or another object *y=f(x')* (with a local image *y'*) produced at server *i+1*, where *f(x')* any function of *x'*. If image *x'* is invalidated due to a server failure or an *Invalidate* message, then *x''* and *y'* must also be invalidated. Similarly, if *x'* is updated then the coherency condition of *x'' | y'* must be evaluated on the *Update* message arrival. If a lazy policy is defined for *x'' | y'* then an *Invalidate* message must be sent to the *i+2* server for this object. To avoid inconsistencies in such cases, the following compatibility rules must be enforced in the coherency conditions definition phase:

- *The image x' of object x, when cached with an selection advisory modifier at level i, cannot be defined with a compulsory modifier at any level j>i.*
- *If the coherency conditions of x' include $D(x)=d_1>0$ then x'' at level j>i can include $D(x')=d_2$ to its coherency conditions only if $d_2=0$, since at the worst case $x''(t) = x(t - d_1 - d_2)$.*

### *3.6 SNMP-based interaction implementation*

In order to implement the proposed scheme with the SNMP protocol, we defined a MIB module that implements the generic interaction semantics through SNMPv2 SMI [5] constructs. Each

distributed manager that supports the proposed caching model is equipped with the policyUpdate MIB so that higher level manager may define selection and coherency conditions through SNMPv2 queries. Update messages are conveyed through the SNMPv2 Inform-Request PDU (acknowledged asynchronous notification syntactically identical to the SNMPv2 Trap PDU).
The policyUpdate MIB module is quite simple and includes two tables and a NOTIFICATION-TYPE object (for update propagation). The remote manager defines selection conditions by adding and modifying rows to the selectTable. Part of the table definition is presented below (the full ASN.1 description is available at http://www.netmode.ntua.gr/mibs/policy_upd_MIB.txt):

```
selectTable OBJECT-TYPE
       SYNTAX SEQUENCE OF  selectEntry
       MAX-ACCESS          not-accessible
       STATUS              current
       DESCRIPTION
              "Selection conditions table. Each conceptual
              row defines a selected object instance."
       ::= { updatePolicies 1 }

SelectEntry ::= SEQUENCE {
       selectIndex         Integer32,
       selectObject        OBJECT IDENTIFIER,
       selectValue         Opaque,
       selectTarget        EntryIndex,
       selectEnforce       INTEGER,
       selectActive        TruthValue,
       selectStatus        RowStatus
       }
```

Selected MIB objects are defined through `selectObject` and caching enforcement through the `selectEnforce` attribute, while `selectTarget` identifies the manager that receives the updates for the selection (the `EntryIndex` type is defined in the Management Target MIB [15] defined by the DISMAN working group). The selection is activated by setting `selectActive` to TRUE and the `selectStatus` attribute is used for ading/deleting rows. Finally, the `selectValue` attribute holds the current value of the selected object and it is encapsulated in the notification PDUs (see below).
One or more coherency conditions are attached to the selection through the coherencyTable:

```
coherencyTable OBJECT-TYPE
       SYNTAX SEQUENCE OF  coherencyEntry
       MAX-ACCESS          not-accessible
       STATUS              current
       DESCRIPTION
              "This table is used for defining the coherency
              conditions enforced on specific selections. Each
              conceptual row defines a condition and associates
              it to a selection."
       ::= { updatePolicies 2 }

CoherencyEntry ::= SEQUENCE {
       coherencyIndex             INTEGER32,
       coherencySelectionID       INTEGER32,
       coherencyCondition         INTEGER,
       coherencyParameter         INTEGER32,
       coherencyBoolOp            INTEGER,
       coherencyStatus            RowStatus
       }
```

Coherency conditions are defined as rows in this table with `coherencyCondition` and `coherencyParameter` identifying the condition type and parameter respectively. The `coherencySelectionID` attribute assigns the condition to a selection by setting its value equal to the appropriate `selectIndex`; multiple conditions are attached with logical operators defined by `coherencyBoolOp`.

The `imageUpdate object` defines the syntax of the notification PDU. It includes the selection table index, the associated object identifier and the updated value:

```
imageUpdate NOTIFICATION-TYPE
       OBJECTS { selectIndex, selectObject, selectValue }
       STATUS  current
       DESCRIPTION
              "The generic notification used for conveying
              updates to other manager-role SNMP entities."
       ::= { updatePolicies 3 }
```

This MIB structure is quite simple and allows the inclusion of single objects in the selections. For our SNMP-based prototype we are currently expanding the MIB structure to add a mechanism (in concept similar to Views [JAN98]) that allows the definition of MIB sub-trees for selection conditions.

# 4. Performance evaluation

### *4.1 Analytical performance model*

Consider a two-level hierarchy of managers: at the lowest level a manager *M* is responsible for a specific domain of managed nodes, while the second level is occupied by *N* other managers ($M_i$*: i = 1..N*) each of which relies on a subset of the managed objects maintained in *M*. This may be easily scaled up to more levels; third level managers may use objects from levels 1 and 2, etc. However, for simplicity, the following analysis is applied on the first two levels. Figure 3 provides an overview of this configuration and identifies the flow of requests and updates between and within the manager modules (average arrival rates are indicated). We model all requests and updates as Poisson processes and the managers as M/G/1 servers.

Since each $M_i$ uses a set of objects maintained by *M*, its human operator and management functions generate requests for such objects with an average rate $\lambda_r$. In order to satisfy these requests, $M_i$ collects the requested objects from *M* in two ways: either every query is directly propagated to *M* (polling) or the requests are served through stored images that are asynchronously updated by M (messages generated by monitoring procedures defined at *M*). Our model adopts quasi-copies in both cases for a fraction *h* of the objects, while the percentage of objects updated through direct requests to *M* is assumed to be equal to *l* (*1-l* is updated by monitoring procedures at *M*). So, each $M_i$ maintains a local object database that stores all the asynchronous updates and caches the quasi images for the appropriate objects. Let us also define the ***coherency index*** *q* as the fraction of the requests for quasi-cached objects that are actually sent out to *M*. Given the configuration and definitions, the request arrival rate $\lambda_r$ in $M_i$ is split in the following way:

- Only part of the rate, equal to $h \cdot \lambda_r$, refers to quasi-cached objects. Further, the fraction *l* of these objects are updated through direct polling and will generate messages to M only if their coherency conditions are violated, i.e. requests are propagated at a rate of $l \cdot h \cdot q \cdot \lambda_r$ (also modelled as a Poisson process). The remaining fraction refers to objects updated through monitoring procedures and is directly served through the local database.
- The request rate for objects that are asynchronously updated by *M* and do not use quasi caching is equal to $(1-l) \cdot (1-h) \cdot \lambda_r$. This is, also, served through the local database (updates asynchronously received and stored) and does not generate request messages to *M*.
- The remaining requests refer to objects directly retrieved from M are propagated to M at an average rate of $l \cdot (1-h) \cdot \lambda_r$.

In effect, each $M_i$ serves two Poisson processes. Requests arrive at a rate of $\lambda_1^c = \lambda_r$ and they are served at an average service time $t_r^c$. In order to satisfy them the manager searches the local cache to determine whether the requested objects are quasi-cached, and if so it evaluates the coherency conditions (we assume that the delay for evaluating the conditions is negligible compared to accessing the database). The second process is formed by the update messages received. These are responses to requests or asynchronous messages generated by monitoring procedures. They arrive at an average rate $\lambda_2^c$ and they are served by installing them in the local cache/database at an average service time $t_u^c$.

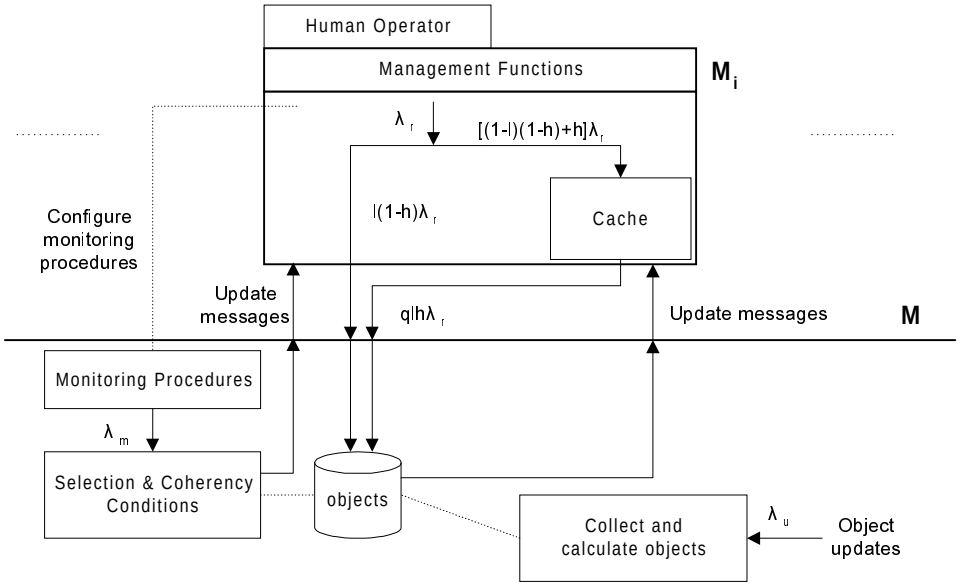


**Figure 3: Configuration assumed in the performance model**

Similarly, the first level manager *M* is responsible for servicing the following Poisson streams:

- Objects changes are perceived at a rate $\lambda_1^s = \lambda_u$ leading to updating the object database and recalculating/updating any dependent objects. This calculation/update is completed with an average service time $t_u^s$.
- Object requests are received from each $M_i$ at an aggregated rate $N \cdot l \cdot (1 - h + h \cdot q) \cdot \lambda_r$, where *N* is the total number of managers using objects maintained at *M*. These requests are served directly from the object database and the appropriate update messages are sent back to $M_i$. Database access and message generation is performed with an average service time $t_r^s$. We assume that only if the requested object has changed, an update message is sent; otherwise a "no change" message is issued (as it will be explained later on, only the former will be processed at $M_i$). In addition to the external requests, the local monitoring procedures also trigger accesses to the object database and produce update messages when appropriate. If $\lambda_o$ is the average rate at which each monitoring procedure polls an object, then such requests emanate at a rate $\lambda_m = \#monitor\_procs \cdot \lambda_o = N \cdot O \cdot (1 - l) \cdot \lambda_o$, where *O* is the average number of objects used by a single $M_i$. Consequently, the aggregated rate served at the average service time $t_r^s$ is $\lambda_2^s = N \cdot \left[ l \cdot (1 - h + h \cdot q) \cdot \lambda_r + \lambda_m \right]$.

Now that we have a global view of the flow of messages between the managers, let us return to the second level managers and define $\lambda_2^c$. Even though all requests that are transmitted to the first

level manager generate responses, in the calculation of $\lambda_2^c$ we consider only to the ones that include an update message; only these will be installed in the local object cache/database. Since object changes occur at rate $\lambda_u$ and we observe them at rate $\lambda_r$, the probability that a response message contains an update is equal to the probability of having at least one arrival of a Poisson process with rate $\lambda_u$ within an exponentially distributed interval with a mean of $\frac{1}{\lambda_r}$. This can be easily proved to be $\frac{\lambda_r \cdot \lambda_u}{\lambda_r + \lambda_u}$, yielding a rate of "non-null" update messages equal to $l \cdot (1 - h + h \cdot q) \cdot \frac{\lambda_r \cdot \lambda_u}{\lambda_r + \lambda_u}$. In like manner, the asynchronous updates are received at a rate of $(1 - h + h \cdot q) \cdot \frac{\lambda_m \cdot \lambda_u}{\lambda_m + \lambda_u}$, thus $\lambda_2^c = (1 - h + h \cdot q) \cdot \left( l \cdot \frac{\lambda_r \cdot \lambda_u}{\lambda_r + \lambda_u} \cdot \frac{\lambda_m \cdot \lambda_u}{\lambda_m + \lambda_u} \right)$.

Given the above analysis, the average object request response time at $M_i$ can be computed by the following formula:

$$R = \left[ l \cdot h \cdot (1 - q) + (1 - l) \right] \cdot \left( w_c + t_r^c \right) + l \cdot (1 - h + h \cdot q) \cdot \left( w_s + t_r^s + 2 \cdot t_n \right) \quad (1)$$

where $t_n$ is the average network transmission time and the $w_c$, $w_s$ are the average queue wait times at $M_i$, and $M$ respectively. The first term corresponds to the request processed locally, while the second term covers the case when the request is propagated to the first level manager. Likewise, the network traffic in messages per second is given by the following formula:

$$M = 2 \cdot N \cdot l \cdot (1 - h + h \cdot q) \cdot \lambda_r + N \cdot (1 - h + h \cdot q) \cdot \frac{\lambda_m \cdot \lambda_u}{\lambda_m + \lambda_u} \quad (2)$$

where the first term covers the requests sent by $M_i$ to $M$ and the corresponding responses, while the second term refers to the asynchronous updates generated by $M$.

In order to calculate the average queue wait times ($w_s$ and $w_c$) we model all managers ($M$ and $M_i$) as M/G/1 servers. The manager in both levels receives more than one Poisson streams with different arrival rates ($\lambda_1$, $\lambda_2$, ..., $\lambda_n$) and service times ($t_1$, $t_2$, ..., $t_n$). The combined flow forms a Poisson process with arrival rate $\lambda = \sum_{i=1}^{n} \lambda_i$, but the combined service time $t$ is no longer exponentially distributed, but its mean and second moment are:

$$E[t] = \sum_{i=1}^{n} t_i \cdot \frac{\lambda_i}{\lambda} \quad \text{and} \quad E\left[t^2\right] = \sum_{i=1}^{n} 2 \cdot t_i^2 \cdot \frac{\lambda_i}{\lambda} \quad (3)$$

The queue wait times can be computed by applying the Pollaczek-Khinchin formula and substituting from equation (3):

$$w = \frac{\lambda \cdot E\left[t^2\right]}{2 \cdot (1 - \lambda \cdot E[t])} = \frac{\sum_{i=1}^{n} t_i^2 \cdot \lambda_i}{1 - \sum_{j=1}^{n} t_j \cdot \lambda_j} \quad (4)$$

thus:

$$(4) \Rightarrow w_s = \frac{\lambda_1^s \cdot t_u^{s^2} + N^3 \cdot \lambda_2^s \cdot t_r^{s^2}}{1 - \left( \lambda_1^s \cdot t_u^s + N^2 \cdot \lambda_2^s \cdot t_r^s \right)} \quad (5)$$

$$(4) \Rightarrow w_c = \frac{\lambda_1^c \cdot {t_r^s}^2 + \lambda_2^c \cdot {t_u^s}^2}{1 - \left(\lambda_1^c \cdot t_r^s + \lambda_2^c \cdot t_u^s\right)} \quad (6)$$

Table 1 summarises the model parameters and presents the values used in our analysis.

**Table 1: Model parameters**

| Parameter | Description | Value |
|---|---|---|
| $h$ | fraction of objects that are quasi-cached | 0..1 |
| $q$ | coherency index | 0..1 |
| $N$ | number of manager at level two ($M_i$: i=1..N) | 5 managers |
| $O$ | total number of M's objects at each $M_i$ | 100 objects/$M_i$ |
| $\lambda_r$ | request arrival rate at each $M_i$ | 5 requests/sec |
| $\lambda_u$ | update arrival rate at each $M_i$ | 34 requests/sec |
| $\lambda_o$ | rate of single object monitoring procedure activation | 0.1 requests/sec |
| $l$ | fraction of objects polled (*1-l* asynchronously updated) | 0..1 |
| $t_r^c$ | request service time at a single $M_i$ | 1 msec |
| $t_u^c$ | update installation time at a single $M_i$ | 2 msec |
| $t_r^s$ | request service time at M | 1 msec |
| $t_u^s$ | update installation time at M | 2 msec |
| $t_n$ | network transmission delay | 25 msec |

Figure 4 and Figure 5 plot the expected response time and network traffic as functions of *h* for several values of *q* and for *l*=0.8. For *q*=1 all requests are propagated to the first level manager, thus quasi caching (as any method of caching in general) does not yield any improvement in performance. Both the network traffic and response time remain practically unaffected by changes in the percentage of objects cached. Actually, performance is expected to slightly deteriorate as *h* increases, due to the additional overhead of cache maintenance; but since the network delay is much higher than processing delays (Table 1) this decrease is not perceptible in the figures. On the other hand, when quasi caching is adopted (*q*<1) both the response time and the network traffic are reduced, while lower coherency levels (smaller values for *q*) yield better performance.

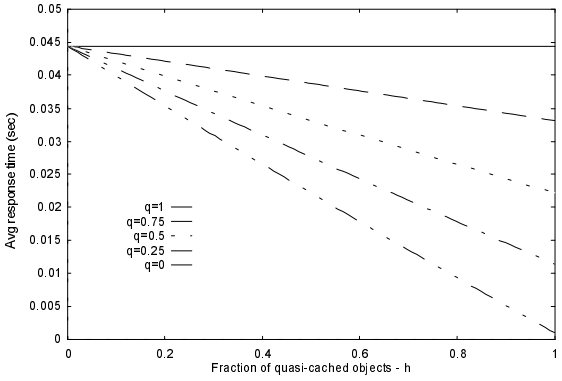


**Figure 4: Response time in sec (*l*=0.8)**

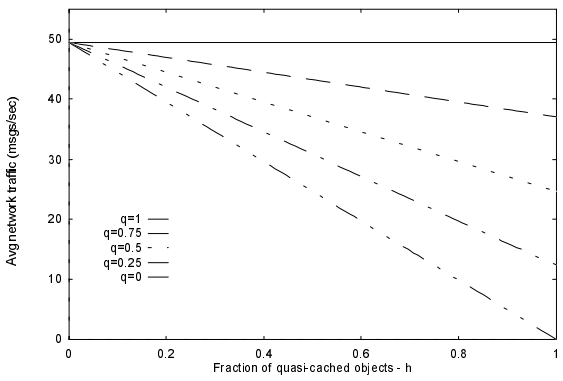


**Figure 5: Network traffic in msgs/sec (*l*=0.8)**

Note that in both curves, *l* is equal to 0.8, i.e. only 20% of the objects are handled by monitoring procedures at *M*. This value was chosen as the most accurate in real life scenarios, a choice driven by our network management practical experience. Different values of *l* in [0..1] yield slightly different maximum values both in response and messages, but the same pattern is observed as *h* and *q* increase. The following table and figures summarise the impact of *l* to the response time and network traffic with 50% of the objects quasi-cached. Even though both quantities increase as *l* increases, the response time is much more sensitive to it.

**Table 2: Response time and network traffic for different values of *l***

| *l* | *R*(*h*=0.5) | *M*(*h*=0.5) |
|---|---|---|
| 0 | 1 msec | 38 messages/sec |
| 0.5 | 28.7 msec | 46 messages/sec |
| 1 | 54.6 msec | 50 messages/sec |

For our analysis we considered average round-trip network delays of 50 ms, a typical value for 1024 bytes messages over 2Mbps WAN communication lines with average utilisation (measurements over the Greek Research Network). It is interesting to observe how the response time curves are affected if we consider lower or higher delays, i.e. the two extreme cases when the proposed management architecture operates within a high speed LAN/Intranet or a very congested WAN.

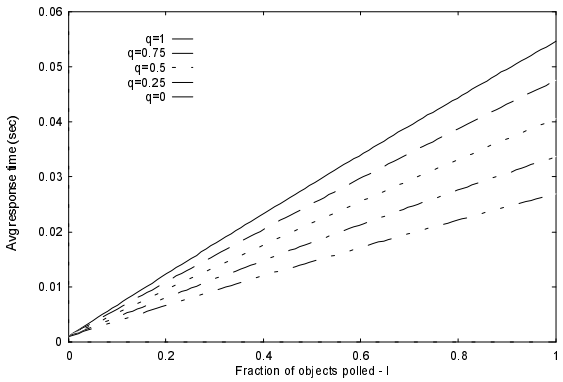


**Figure 6: Impact of *l* on response time (*h*=0.5)**

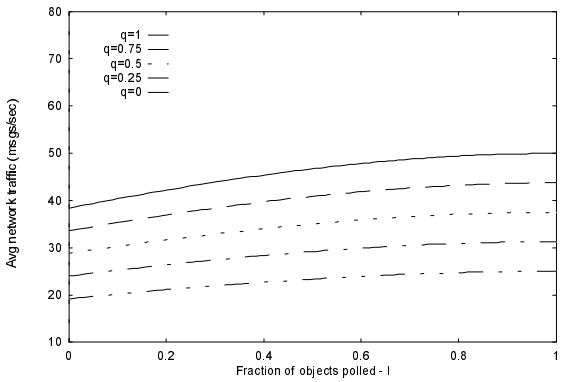


**Figure 7: Impact of *l* on network traffic (*h*=0.5)**

Figure 8 plots the response time as a function of h for l=0.8 and an average round-trip network delay of 0 ms and Figure 9 plots the curve for 1 sec delay. In the first case the response time is defined by the processing at the two managers ($M_i$ and $M$) and messages are exchanged between them without any significant delay compared to the processing delays, while in the second case the network is the dominant delay factor. Let us compare these curves to the ones presented in Figure 4. As expected, the response time in the high speed network is significantly reduced: 4.5 msec versus 45 msec for $q$=1. Notice also that the performance gain is not linear anymore and since the saturation of the processors can be observed in the curve for larger values of $h$.

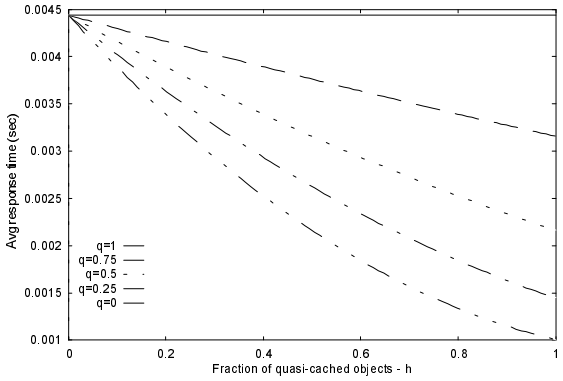


**Figure 8: Response time for *l*=0.5, $t_n$=0**

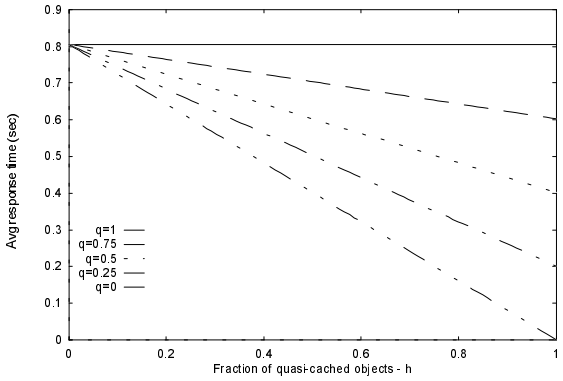


**Figure 9: Response time for *l*=0.5, $t_n$=0.5**

In the congested network case, the network is by far the governing delay and as expected the response time is much higher than in the 0 and 25 msec cases: nearly 800 msec compared to 4.5 msec and 45 msec.

### *4.2 Experimental results*

Encouraged by the analytical results we developed a prototype of the caching scheme and interaction semantics. We conducted experiments with slimmed-down versions of distributed managers using embedded lightweight RDBMSs for cache implementation and a simple UDP-based version of the generic message protocol defined in section 3. The low-level managed objects used in the experiments were not retrieved directly from agents, but their were fed to the lower level managers in the form of pre-recorded real-time traces. The main goal of the experiments was to identify the performance gains for various types of managed objects.

Figure 10 outlines the configuration used during the experiments. Managers are deployed in two levels: the i-level manager prototype retrieves the (i-1)-level objects, provided as real-time traces, and calculates i-level objects offered to the higher levels. The (i+1)-level managers (clients) cache the i-level objects (quasi-caching is used) and receives updates according to the defined coherency conditions.

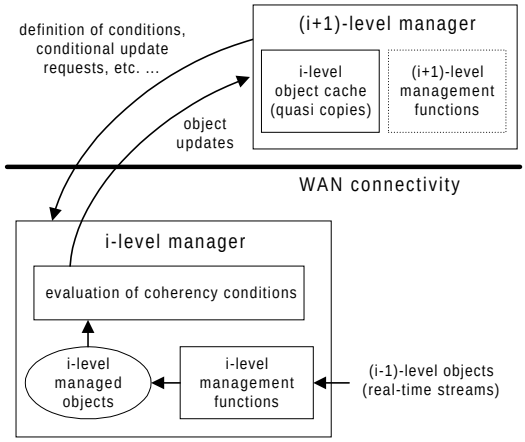


**Figure 10: Experiment configuration**

The (i-1)-level objects are polled by the i-level manager and appropriate calculations are performed (e.g. throughput) producing the i-level objects that are in turn used by the (i+1)-level manager (quasi-cached). We examined three managed objects types: input throughput of a WAN link (Kbps), the http requests served by a WWW server (requests/sec) and the free memory of a server (Kbytes). A set of real-time traces were fed to the experimental configuration for each of the three of objects; each experiment lasting 8 hours (real-time). Examples of traces (the first 3000 sec) from each category are presented in Figures 11, 12 and 13.

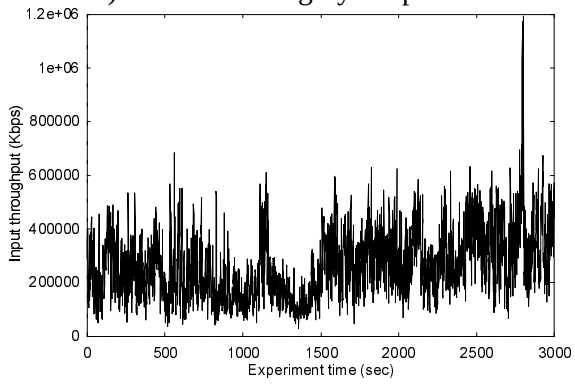


**Figure 11: WAN object trace (t ≤ 3000 sec)**

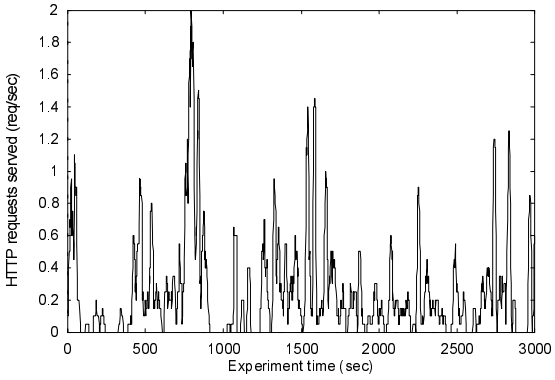


**Figure 12: WWW object trace (t ≤ 3000 sec)**

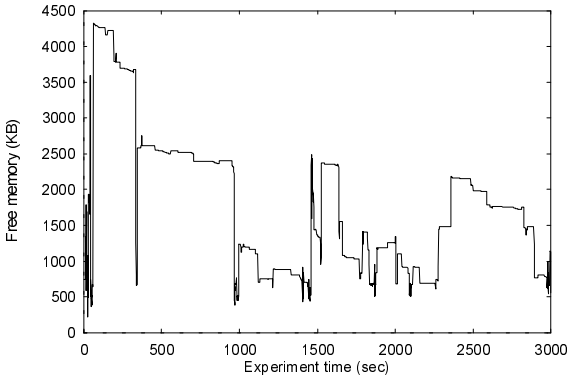


**Figure 13: Memory object trace (t ≤ 3000 sec)**

Although statistically all three traces are characterised by high autocorrelation and their Hurst parameters are higher than 0.85 (variance plot method), observe that the memory trace is the 'smoothest' of the three. Changes are far less frequent than in the WWW trace, while the WAN trace fluctuates even faster.

For every object type we measure the number of messages actually sent and the cached image deviation (error %) at level i+1. We have chosen the *PC(x)=e* coherency condition for our experiments since the version and absolute change conditions are similar in concept and the delay condition depends greatly on object access patterns (most suitable for lazy updates). Then we compare these two factors (management traffic and error %) to the ones achieved when the conventional periodic polling is used. For comparison reasons we identify two 'equivalent' polling periods: the polling period that yields the same average error % (for comparing management traffic curves) and the period that yields an equal number of messages (for comparing errors curves). Figure 14 plots the first 350 sec of an experiment with PC[X(t)] = 10% and the equivalent polling that yields the same number of messages. Notice that the proposed method provides a curve that follows the actual object fluctuation more closely than the traditional polling that produces the same management traffic.

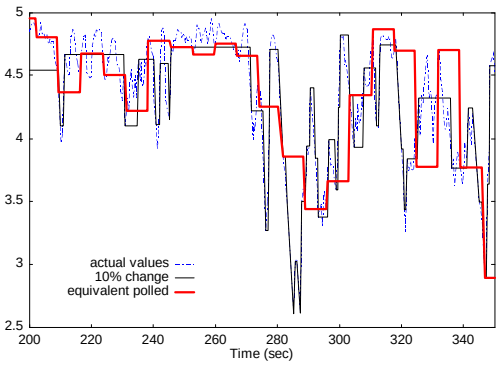


**Figure 14: PC[X(t)] = 0.1 and equivalent polling**

Figure 15 and Figure 16 present the results of the experiments with the WAN link throughput object (average for a set of various traces for the same link). Management traffic and error % are plotted for various deviation parameter values (*e* in [0,0.6]) and they are compared to the equivalent polling curves. As expected, the management traffic is reduced as the maximum tolerated deviation (*e*) increases, since fewer updates need to be propagated. The equivalent polling method yields a similar behaviour, however, quasi-caching provides at least a 25% improvement. The image deviation (presented as average error %) increases near-linearly to *e*, yielding again better results than polling. The manager that uses quasi-caching operates on images that deviate from the actual object at least 25% less than in the equivalent (equal number of messages) polling case.

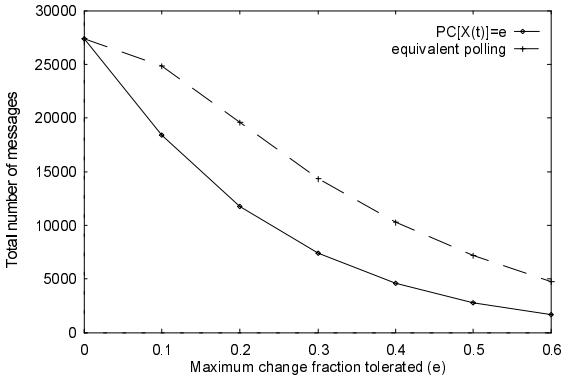


**Figure 15: Management traffic - WAN object**

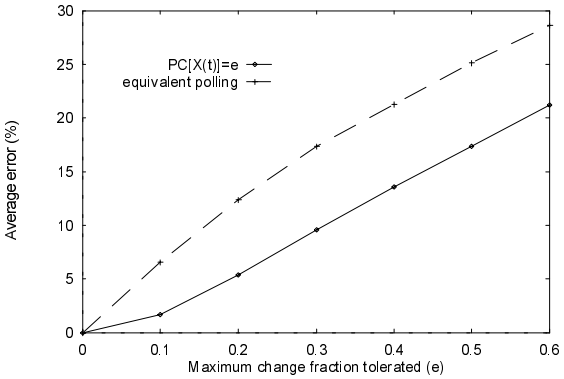


**Figure 16: Error % - WAN object**

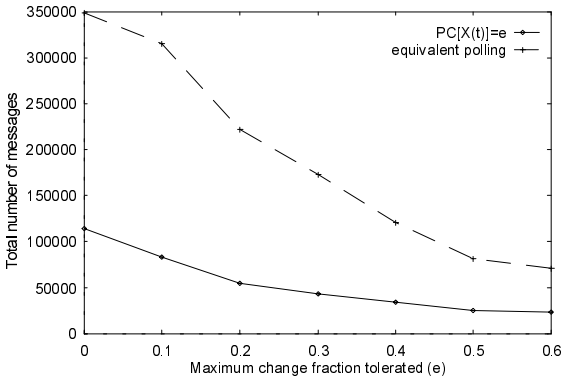


**Figure 17: Management traffic - WWW object**

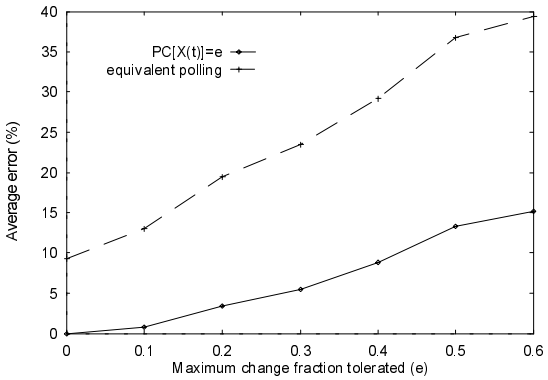


**Figure 18: Error % - WWW object**

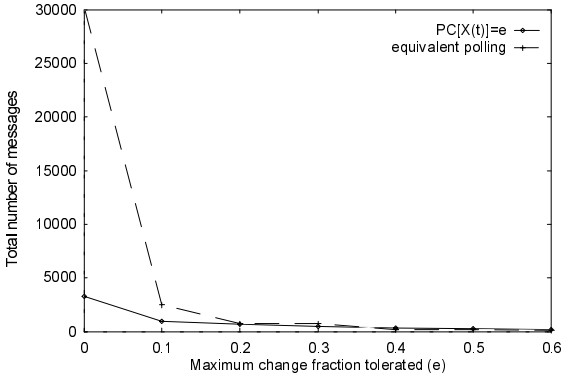


**Figure 19: Mgmt traffic - Memory object**

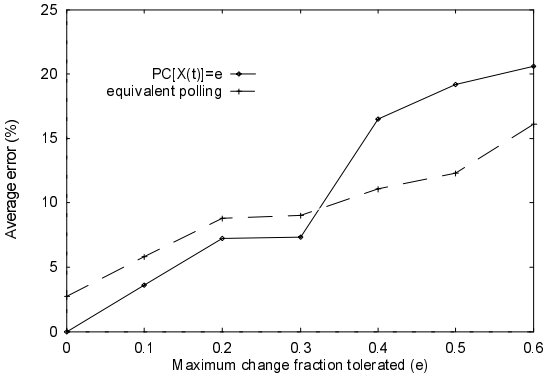


**Figure 20: Error % - Memory object**

Management traffic and average error curves for the WWW requests throughput object are presented in Figure 17 and Figure 18 respectively. The gain is a little higher than the WAN object, but the real difference is the very significant improvement over polling achieved for $e = 0$, i.e. in the case where updates are not propagated if the object remains unchanged. As it was pointed out earlier in the text, the WWW trace is 'smoother' than the WAN trace. Thus eliminating messages when the WWW object remains constant yields a higher improvement. On the other hand, the WAN object changes so fast that no improvement is achieved for $e = 0$.

Management traffic and object deviation for the memory object experiments are presented in Figure 19 and Figure 20 respectively. These curves increase/ decrease similarly to the previous cases, however, when comparing to the equivalent polling method the gains are not significant. In fact, polling seems to perform better in terms of average error for tolerated deviation higher than 30%. Recall that the memory object changes more slowly than the other two, thus the gain for $e$ = 0 is much higher. Yet, the rate of change is so small that controlled updating performs similarly to periodical polling, and after a point may even prove worse. This can be alleviated if the percentage change method is combined with the delay method so that an update will be generated if $PC[X(t)] = e$ is not invalidated for some pre-established interval (3 sec in Figure 21). As our further experimentation has shown, the combination of the "delay method" with "value or version change" conditions is in most cases able provide near-optimal performance gain, even when the condition parameters are not fine-tuned (instead roughly approximated or based on guessing).

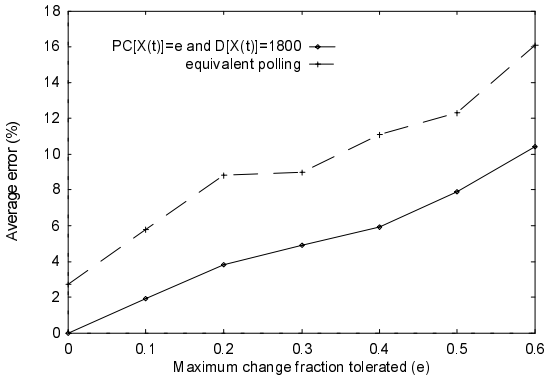


**Figure 21: Error % - Memory object (percentage change and delay coherency conditions)**

## 5. Conclusions - Future work

We presented a caching and updating scheme for improving performance in distributed management architectures that takes advantage of the various coherency requirements, inherent in management applications. We define a range of coherency conditions and the related caching update policies that govern their implementation within a hierarchical, scaleable architecture. The main advantage of applying these policies is the achievement of limited management traffic through caching and the introduction of timely notifications in place of periodical polling. Management functions with relaxed object consistency requirements may take advantage of the proposed model by demanding updates only when their coherency conditions are violated. The model supports various coherency levels, depending on the object type and the application requirements. As can be observed by our experimental results, the improvement in performance depends greatly on the temporal behaviour of the managed object in question. Inappropriate use of coherency conditions will not improve and may even slightly decrease performance. This is probably the main disadvantage of the proposed method; selecting the most suitable set of coherency conditions and update policies may prove complex and must be decided by the client. Aiming to simplify this difficult task, we are currently working on defining a mapping of object classes to best suited coherency conditions and we intend to investigate an automated way for simplifying the selection by the client. Also, are we plan implement a full-fledged prototype of an SNMP manager that implements the proposed architecture, replacing the lightweight RDBMS of the experimental prototype with the Wisconsin EXODUS [2] object Storage Manager and the generic protocol with SNMPv2 (some hints were presented in section 3). Investigation of a CORBA-based implementation in also in our future goals. Intensive experimentation and testing with this manager prototypes in real-life scenarios will provide a clearer view of the performance gains and implementation issues.

# 6. References


[1] R. Alonso, D. Barbara, and H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System", ACM Transactions on Database Systems, 15(3), September 1990.

[2] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg, "The EXODUS Extensible DBMS Project: An Overview", Readings in Object-Oriented Databases, S. Zdonik and D. Maier, eds., Morgan-Kaufman, 1990.

[3] J.A. Carillo, E.R.M. Madeira, “A Scheme for FTP Management”, INET’94 / JENC5, 1994.

[4] J. Case, B. Levi, “SNMP mid-level-manager MIB” and “SNMP script language”, Internet Drafts, 1993.

[5] J.Case, K.McCloghrie, M.Rose, S.Waldbusser, "Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)", RFC 1902, January 1996.

[6] G. Goldszimdt, Y. Yemini, “Distributed Management by Delegation”, 15th International Conference on Distributed Computing Systems, June 1995.

[7] M. Kahani, H.W.P. Beadle, "Decentralized Approaches for Network Management", in Computer Communications Review, ACM SIGCOMM, vol. 27, no. 3, July 1997.

[8] R. Konopka, M. Trommer , “A Multilayer-Architecture for SNMP-Based, Distributed and Hierarchical Management of Local Area Networks”, In Proc. 4th International Conference on Computer Communications and Networks, ICCCN’95, 1995.

[9] C. Picoto, P. Veiga, “Management of a WWW Server using SNMP”, In Proc. JENC6, 1995.

[10] S. Rabie. "Integrated Network Management: Technologies and Implementation Experience", IEEE Infocom '92, 1992.

[11] M.R. Siegl, G. Trausmuth, “Hierarchical Network Management”, In Proc. JENC6, 1995.

[12] F. Stamatelopoulos, T. Chiotis, B. Maglaris, “A Scaleable, Platform-Based Architecture for Multiple Domain Network Management”, In Proc. IEEE International Conference on Communications ‘95, June 1995.

[13] F. Stamatelopoulos, N. Roussopoulos, "Using a DBMS for Hierarchical Network Management", NETWORLD+INTEROP ’95, In Proc. 2nd Annual Interop Engineers’ Conference, Las Vegas, March 1995.

[14] F. Stamatelopoulos, B. Maglaris, "A Management Architecture for Internet Information Services", In Proc. 4th Plenary Workshop of the HP OpenView University Association, Madrid, April 1997.

[15] B. Stewart, "Management Target MIB", Internet Draft, draft-ietf-disman-mgt-target-mib-01.txt, March 1997.

[16] S.L. Waldbusser, “The Trend Towards Hierarchical Network Management”, The Simple Times, The Bi-Monthly Newsletter of SNMP Technology, Comment, and Events, Volume 2, Number 6, November/December 1993.

[17] S. Waldbusser, B. Stewart, A. Bierman, M. Greene, "Distributed Management Framework", Internet Draft, <draft-ietf-disman-framework-01.txt>, December 1997.