Appeared in:
Proceedings of the IEEE First International Workshop on System Management, Los Angeles, 1993

Automatic Selection of an Update Strategy for Management Data

Ernö Kovács

University of Stuttgart, Institute of Parallel and Distributed High Performance Systems (IPVR),
Breitwiesenstr. 20-22, 70565 Stuttgart, Germany,
E-mail: ernoe.kovacs@informatik.uni-stuttgart.de

ABSTRACT

Systems management operations are based on data that is distributed throughout the network. Monitoring this information requires network messages and can increase network traffic significantly. Selecting the right update strategy for such continuous management activities reduces the network load. Automatic selection of an update strategy is based on two factors: the access behavior of the managing applications and the change behavior of the data. In addition, management applications may have special requirements on the actuality of the data. We show how an optimal update strategy can be selected regarding access rate, change rate and the requirements of the application. In this way the task of selecting the right strategy and updating the management data can be separated from the logic of the management task itself. At last we describe an implementation and the use in a network service trading system.

***Key Words**: Systems Management, Distributed Management System, System Monitoring, Update Strategies, Trading*

1. Introduction

Systems management has to monitor many different objects, which are distributed throughout the system, and which belong to different application fields or managed subsystems. During management operations information from these objects must be collected. This requires network messages and can consume a good deal of network bandwidth. Selecting the right strategy to update locally available information could reduce the network load significantly. An important factor that influences the update strategy is the characteristics of the management data itself. While fast changing data should only be accessed when needed, data with a very slow change rate could be stored in the local cache and updated whenever a change occurs. The decision where to collect the information, and when to update the local information is further influenced by the requirements of the management application. A lot of systems management functions can operate on slightly out-of-date data if certain, management function depended constraints are not violated. The selection of the update strategy should be made automatically by the management system. In this way the information collecting task can be adapted to new objects and new situations by just focusing on the requirements and not by changing internal working parameters, customizing polling intervals, setting event filters, or else.

As an example, consider the characteristics of the counter of free blocks on a hard disk that is under heavy usage. The management function that monitors the counter signals the system administrator when the disk fills up. The value of the counter changes very fast. In this case accesses to the counter should be made in very small intervals. Later in the evening disk accesses and counter changes slow down. Now the polling interval for this counter would be much to small, resulting in unneeded communication overhead. The management system should adapt to this situation automatically and extend the polling interval. The problem here is that a fixed update rate was selected by a management function. Another aspect of the same problem is that similar data items (e.g. free block counters for different disks) behave differently. Usually, one update strategy is selected for all counters, whether this is appropriate or not. Again, it would be wise to adapt the update strategy for all items individually.

For a lot of applications the value of the management data must not be up-to-date to the last second, thus enabling the management system to store and access older copies of the management data. By relating the requirements of the management application to the update strategies we can reach a communication behavior of the

This work was supported in parts by IBM Germany under study contract S142 “Management of Distributed Systems”

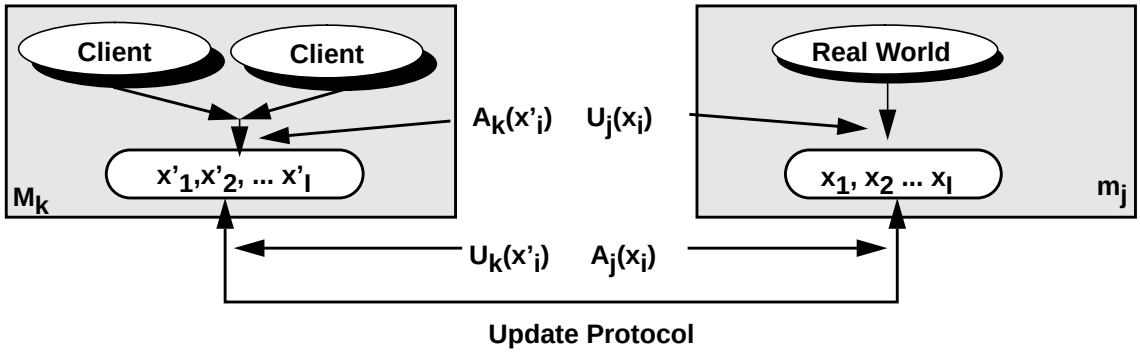

Figure 1: System model

management system that is less demanding for the whole system. To solve this problem we must take the following steps: First, we need a notation to express the requirements of the management application. Second, we must take this requirements and select an update strategy, that still guarantees the expressed requirements, but minimizes network traffic. Third, by observing the behavior of manager and managed application we can adapted the strategy automatically to individual changes of the system behavior.

At the end, the system reduces the task of optimally accessing management data from the real management function to just defining the appropriate requirements. A management application should only declare what data is needed and how up-to-date the date should be. After that the management application accesses the data when needed. The management system is responsible to access data on a per-request-base or to use local copies and to keep them up-to-date. Communication activities like periodic polling of the data or sending out change reports are performed by the underlying communication system and can be tailored to the characteristics of the management system and the management data. This results in a uniform way to access management information and to an easy programming model for management application.

In the following, we present our model of the characteristics of management data. Then we describe several possible update strategies and the parameters that influence this strategies. After that we conclude how to select a strategy with minimum cost. We then examine how a manager can state its requirements on the actuality of the data using delay predicates. We examine the delay predicates and discover that they are based on two different kinds of measure for actuality. We than show how this two measures can be used to implement an abstract actuality manager that can be tailored to special requirements for special data. At the end we give an overview how this is implemented in the *MELODY* management system and used for trading of services that are accessible over the network.

2. Characteristics of management data

Our system model consists of managed systems and management systems. Management application on the management systems access management data on managed system that represent real world objects. Some of this data may be stored in local copies on the management system and updated by using a certain update strategy. In this section we define our system model and introduce a notation to talk about certain aspects of the system. The introduced terms are later used to discuss different update strategies and network traffic rates, to identify values of a management data at different time, and to distinguish between the original information and local copies on management systems.

Figure 1 presents our model of the system and the characteristics of the management data. We study the case where pieces of management information (called x_i, $1 \le i \le I$) originate from managed systems m_j ($1 \le j \le J$) and are used on management systems M_k ($1 \le k \le K$). When used on the management system, we call the piece of information x'_i to distinguish between the real information and the information on the management system. x'_i denotes a piece of information in the local cache of the management system. We also use only x or x' if we refer to any x_i or x'_i. We call x(t) the value of x at the time t. The dependency between x and x' can be stated as follows:

$x'(t_0) = x(t_0-\alpha)$

α consists of the communication delay d and the delay between the last update of x' and the moment of the current access (t_0). The communication delay d is also called the *uncertainty interval* because one will always need this transfer time. Note that α is not a constant and depends on the time t_0.

Both, x and x', have access rates and update rates on m_j and M_k. This is written as $A_j(x)$ [$A_k(x')$] for the access rate and $U_j(x)$ [$U_k(x')$] for the update rate. $U_k(x')$ only counts the explicit change reports from x. This means an access to x may implicitly update the value of x', but is not counted as an update for x'. It should be clear that the value of the different rates depend on the selected update strategy. For example, if the update strategy is report every change, then $U_k(x')$ is the same as $U_j(x)$. Also in this case $A_j(x)$ is zero, because a manager only accesses the local copy x'.

3. Update strategies

There are three methods to access x and to update x' on the management system:

1. Direct Access:

Each access to the value results in a network access to x. The time to access x is twice the communication delay d. The value of x' is the best we can get regarding the uncertainty interval: $x'(t_0) = x(t_0 - d)$ (t_0 is the time the value is delivered to the management application).

The number of packets exchanged for one manager is twice the access rate for x', one packet for the request and one for the reply: $2*A_k(x')$. The packets exchanged for all managers is $2 * \sum A_k(x')$. Assuming that the access rates of all managers are comparable, this gives: 2K*A(x'). The access rate at the managed system m is $A_m(x) = \sum A_k(x')$.

2. Report Changes:

Each update of x is sent to all managers. In this case U(x) = U(x') on each manager. The network packet rate however is the number of managers interested in x (written as K) multiplied by U(x): K * U(x). Note that in this case we store x' on the manager.

The time to access x' is just the time to access the local cache. The value of x' is the best we can get: $x'(t_0) = x(t_0 - d)$ (t_0 is the time when x' is updated on the management system). Actually, $x'(t_0) = x(t_0 - d_i)$ with $0 \le d_i \le d$, reflecting the fact that x can change somewhere in the uncertainty interval, but the new change report has not arrived yet.

3. Periodic Polling / Periodic Reporting:

A third method to update x' is to periodically poll the value of x or to receive a periodic report from x. If we call the rate for this period P(Δt) and Δt the length of the period, then A(x) = P(Δt) = U(x'). There is no relation between U(x), P(Δt), and A(x'). The number of network packets are 2K * P(Δt) (Polling) and K * P(Δt) (Report).

The time to access x' is just the time to access the local copy. The value of x' is $x'(t_0) = x(t_0 - d_i)$ with $0 \le d_i \le (t_0 - t)$, t is the time of the last update, reflecting the fact that x changed somewhere between the last update and the present.

Looking at the packet rates it is easy to see that one has to find the minimum of 2A(x'), U(x) or P(Δt), to have a rule for selecting the best strategy. A(x') and U(x) can be observed at the manager or at the managed system. The value for P(Δt) must be determined by other means, for example by the requirements of the application.

Some more discussion about where to collect the network information and how to decide what update strategy should be used in the static case, can be found in [1]. In the following we examine how a manager can express its requirements on the actuality of the data and how this influences the decisions to select a certain strategy.

Strategy	Network Rate	Access Time
Direct Access	$2K * A_k(x')$	2 d
Change Report	$K * U(x)$	0
Periodic Access	$2K * P(\Delta t)$	2 d
Periodic Reporting	$K * P(\Delta t)$	0

Table 1: Comparison of different strategies

4. Delay predicates

Requirements can be expressed by predicates that are true when the requirement is met. In our case we use delay predicates to state requirements about the delay between x and x' (This notation of predicates is based on the work done in [2] and [3] about stashing and quasi-copies). We give predicates for different delay measures. We also state a minimum strategy that could be used to fulfill the predicate.

1. Null Predicate: $N(x') \equiv \{ \exists t: x' = x(t) \}$

The Null Predicate expresses that there are no requirements on the maximum delay of x'. It is always true after one access to x. As a result the attribute may be stored as a quasi-copy on the client node and used from now on for every access. This predicate is useful for constant x.

2. Time Predicate: $D(x') \equiv \{ d(x') \le \alpha \}$

The function d(x') is defined as follows: If x'(t) = x(t-a) then d(x') = a. In other words, x'(t) should not be older than $x(t-\alpha)$. The easiest way to fulfill this predicate is to use a periodic update policy with parameter

$\Delta t \leq a$. This guarantees the correct delay for this predicate.

3. **Change Predicate: $C(x') \equiv c(x') \leq \beta$**

   The function c(x') is defined as follows: Let C_r be the sequence of discrete values of x. If $x(t) = C_j$ and $x'(t) = C_i$, then $c(x') = j-i$. This predicate requests that x' must be one of the last β values of x. One strategy to fulfill this predicate, a change report must be send at least after every β changes. Note, that the direct access is also suitable to fullfil this predicate.

4. **Version Predicate: $V(x') \equiv v(x', f()) \leq \varepsilon$**

   The idea of this predicate is to define main versions of x and count only new versions instead of all changes of x. The function f() is the version criterion that decides which change in x is a new version. Let V_r be a sequence of main versions of x with $V_r = C_j$. If $x(t) = C_j = V_s$ and $x'(t) = C_i = V_t$ then $v(x') = s-t$. In this case, V_s and V_t represent major changes in x and the predicate expresses that x' should not be more than ε versions behind x. Version predicates are used in conjunction with special knowledge about the possible versions of the attribute. The decision function f() must be known in advanced to use this predicate.

   The update policy can be the same as in 3, with the exception that this time not every change in x is reported.

5. **Delta Predicate: $L(x') \equiv l(x, x') \leq \delta$**

   The function l(x') is defined as $|x'(t) - x(t)|$. This predicate requires that the absolute difference between x and x' is smaller than δ. This predicate is useful for an attribute with an total ordering and a compare operator (e.g. integers or real numbers). It is used when small changes within a certain range are allowed, but a discrepancy bigger than δ should not go unrecorded.

6. **Threshold Predicate: $T(x') \equiv \{t(x, x', \tau) = \text{TRUE}\}$**

   The function $t(x', \tau)$ is TRUE if both x(t) and x'(t) are bigger or smaller than τ. In the other case $t(x', \tau)$ is FALSE. So $t'(x, x', \tau)$ is defined as TRUE if both $x(t-u)$ and x'(t) are bigger or smaller than τ. The growth of the value of x above (below) a certain threshold τ is the condition that triggers a change report.

These delay predicates capture different aspects of the delay between x and x'. Many of them require additional parameters or some form of definition, e.g. what is a new version or how is a delta computed. So the above list defines some classes of delay predicates containing many different real predicates. For real implementations we need certain predefined data type dependent version functions and for special cases the exact definition of the version function.We deal with after the next section, where we examine some general principals of the predicates.

When looking at the delay predicates we notice that the direct access mechanism could fulfill every predicate. With respect to the uncertainty interval, the direct access mechanism guarantees the most current value of x, but may result in unnecessary network traffic. For the time predicate a periodic strategy or a change report strategy may be selected. For this case the time predicate gives us a value for Δt. This could be used to select a best update strategy according to our discussion in the last section.

Looking at the change predicate we notice that β reduces the update rate of x. Instead of U(x) one can select $U(x) / \beta$ for the update rate when computing a new strategy. All other predicates also reduce the update rate, but the effect depends on the new value of x and on some additional parameters like the threshold or the delta. No periodic update strategy can guarantee the change, version, delta and threshold predicate. This must be either done by the direct access or by the change report strategy.

Delay predicates as stated above are implicitly used in many management systems. An example of a system that uses some form of a time predicate is the Meta system ([4]) that states the maximum polling interval for a sensor. The sensor must be polled at this (or shorter) period to avoid missing significant events. [5] describes strategies for a decentralized resource management that uses a form of the delta predicate to update informations about the load factor of one machine. Several different management platforms have implemented some forms of delay predicates to define event generation. [6] uses forms of delay predicates in an ai systems to control the amount of input date during real-time diagnostics.

In the following section we show how the different delay predicates can be reduced to two basic forms. This basic forms can be used to build an abstract actuality manager which can be tailored to many different delay predicates.

5. What do delay predicates measure?

Delay predicates capture a certain aspects of the discrepancy between the original source of the information and the value that is used on another node. They restrict the allowed delay according to a certain measure. We now examine the characteristics of the delay predicates to find out what is really measured. The result will be that delay predicates measure two different aspects of the delay between x and x'.

Figure 2 shows the changes of x over the time t. The value of x changes in discrete steps. Let t_i be the time when the value of x changes the i-th time. Let C_i be the i-

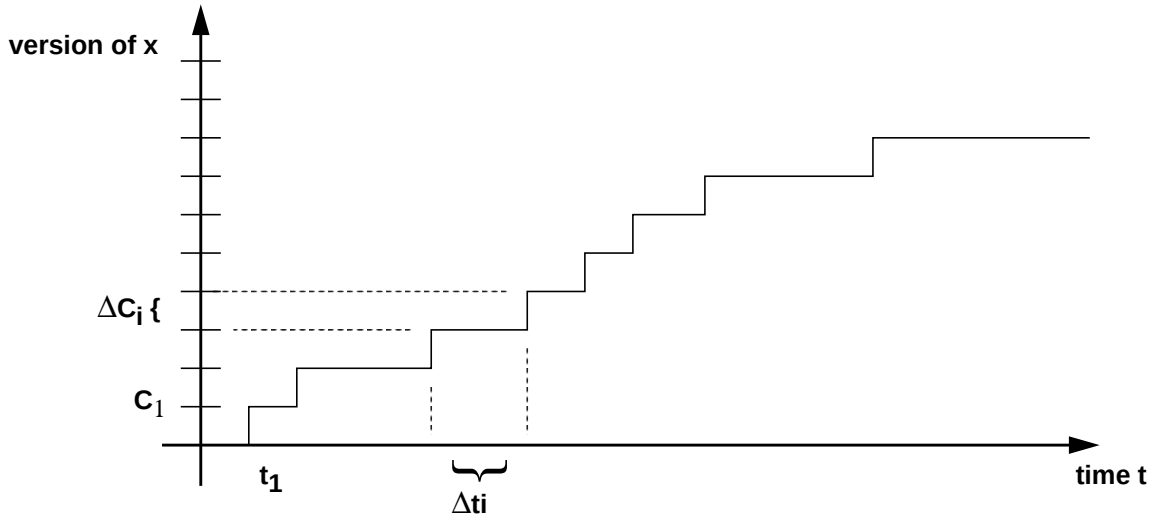

Figure 2: Behaviour of management data x

th new value of x ($C_i = x(t_i)$). Δt_i is the time period in which x keeps its i-th value, that is the time period from t_i to t_{i+1}. $\Delta t_{i,j}$ is the time period from t_i to t_j. In the same way is ΔC_i defined as the interval between the C_i and C_{i+1} (with the size 1), and $\Delta C_{i,j}$ the interval between C_i and C_j. We now show that the delay predicates define intervals on one of the two axes. We can measure the delay from x' to x with respect to time intervals. If $x'(t) = x(t_i)$, then we can use $t - t_{i+1}$ as a measure for the distance between x' and x. We select t_{i+1} because x keeps its value until t_{i+1}. In other words x' is $t - t_{i+1}$ time units behind x. This is called the time measure. A time predicate restricts the length of the interval on the time axis.

We can state the delay between x' and x with respect to the changes in the value of x. If $x(t) = C_i$ and $x'(t) = C_j$, then the number of missed changes ($\Delta C_{i,j}$)is the measure for the distance between x' and x. In other words, the value of x' is i-j changes behind x. This is the change measure. A change predicate restricts the length a interval on change axis.

Some other measures are:

- changes in the version of a datum, where a version is a major change in the value of x (version measure).

  In this case we just use a bigger raster to measure the changes. Instead of counting every change some definition for a major change is used. As defined above, the definition of a major change may vary. The change measure is a special kind of the version measure.
- changes of the value of x (delta measure).

  As above, not every change in x is counted. We have here another definition of a new version. Note that for this measure we must store the value of the latest update to compare the new value with. This is called the base value of a predicate.
- reaching a threshold defined for x (threshold measure).

  The new version is reached, when a certain threshold is crossed. As above, we need to store an additional base value, in this case the threshold value.

Looking at the figure and regarding the different measures, we see that there are only two relevant measures, the time measure and the version measure. The time measure gives values for the distance on the time axis. The version measure measures distances in between values of x. There are several different ways to express distances for the value of x. They are all based on counting some form of changes in x. Therefore, the different forms of the version measure may exist and could be adapted to the needs of the application or the semantics of the data. Instead of implementing every possible delay predicate, one can implement an abstract mechanism for the time and the version measure and then a way to adapt the version measure to the other measures. All that is needed for the other measures is some form of functions that decides wether a change in the value of x should be regarded as a new version or not.

6. Combined delay predicates

Delay predicates may be combined to fulfill the needs of different management applications running on the same node or to express different requirements on the delay. As an example, it should be possible to state that the informa-

tion should not be older than 10 minutes and should be within a 10% range of the value of x. This could be expressed with a time predicate and a delta predicate. We now have to examine the dependencies between combined delay predicates.

It should be clear that combined delay predicates are combined using a logical AND. This means that each delay predicate should be met. Using a logical OR would allow the system to choose the less demanding predicate. For this reason we don't consider this as an option. Combining predicates of the same kind could be reduced to meeting the strongest predicate and forgetting about the rest. A problem arises when predicates of different kinds and measures are combined.

We can define four classes of delay predicates. The first contains all time predicates, the second all change predicate, the third all version predicates with the same version function, and the fourth all delta predicates. Predicates of different classes are independent of each others, so it is not feasible to determine a strongest predicate. For this case the strongest predicate of each category must be supervised. When some actions like updating x' are performed to meet a single predicate, the base values of all other predicates (e.g. the compare value for a delta predicate) should also be set to the new value. In this case U(x) must be computed as the sum of all changes that would trigger an update.

Here the difference between an event that is triggered if a certain value has changed for more than 10% and a delta predicate can be observed. The delta event is always triggered, when the value is out of range. The delta predicate just ensures that the value of x' is within a range of x. If the value of x' increases in small steps due to update strategies needed for other delay predicates, the delta predicate would never trigger an update. The value of x' may always be in the correct distance to x. This may be confusing for people who think more in the event based way. A delay predicate however is only a statement about the discrepancy between a copy x' and the original value x. On the other hand, it is easy to implement the range checking function on the manager side and to verify this after each update of x'.

7. Automatic adaptation of the strategy

The decision which update strategy to select is strictly based on A(x'), U(x) and P(Δt). P(Δt) is given by the delay predicates that are declared for x. The problem is to collect the other factors, to select the right update strategy and to adapt the system to the new strategy.

Initially, either sound estimates for access and update rate can be provided or we have to start with an arbitrary update strategy. From that on the access rate can be verified by observing the real behavior of the management application and adapting the value of A(x'). This is particular needed if more than one managing application accesses x'. A(x') can be collected on the management system and is available there at nearly no cost. The update rate U(x) can be observed on the application side in the same manner.

To adapt the strategy, these factors must be collected occasionally at the instance that makes the strategy decision (usual the management system). This could be done in regular intervals, when a new delay predicate is expressed by a management application, or when a rate has changed significantly. The different rates are collected and in the case of an strategy change the new strategy must be propagated securely to both sides of the system. The cost for this can be minimized by piggybacking the information to management messages, but from time to time additional messages are needed. Therefore this only adds slightly to the overall load.

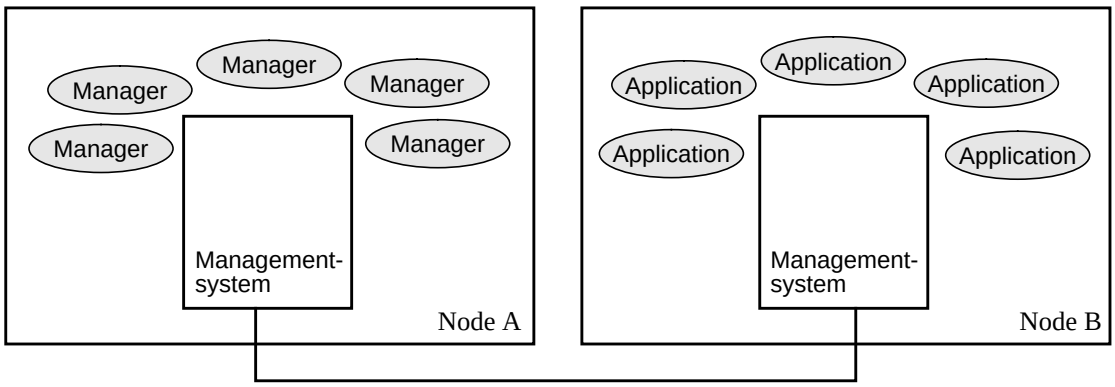

Figure 3: Management system

8. Implementation

The following section describes the implementation of an actuality manager (AM) that supervises delay predicates [7]. The implementation was part of the on-going project *MELODY* (Management Environment for Large Open Distributed sYstems). The *MELODY* management system has architecture as presented in figure 3. Management processes (managers) and managed components (applications) communicate with the management systems. Applications are instrumented using either shared memory or inter-process communication. They provide management objects (MOs) which are accessible for the management system. Managers state the name of the managed objects they want to access. The management system searches for the management object and provides the managers with a quasi-copy of the searched data. This has the advantage that managers accessing the same kind of data only need one copy of the data, but the disadvantage that managers accessing the same data must be synchronized. Although suggested by the picture, managers and applications may reside on the same node. In this case they use the same management system. Communication between management system and managers or agents is strictly asynchronous and message based. The management system is implemented in C++ on RS/6000 workstations.

A software component that adapts automatically to delay predicates and different characteristics of management data must be naturally divided in two parts, the one on the manager side and the one on the managed side. We call the one on the manager side AM_M (AM manager) and the other AM_A (AM agent). Both, AM_M and AM_A, have to agree upon the selected strategy and perform certain kinds of actions. We first describe the tasks that are perform by each side, and after this the messages that are exchanged and the method how our actuality managers are added to our management system.

The AM_M receives and stores the delay predicates for attributes of MOs. It communicates with the AM_A to set delay predicates that must be supervised on the agent side. Both sides of the AM collect the access and the update rates and exchange them in periodic intervals. The AM_M decides which strategy should be used and instructs the AM_A to keep track of the needed actions.

The AM provides the basic mechanism to supervise the time and the version measure. Other delay predicates are supported for basic data types like integers (delta and threshold predicates). Special delay predicates could be implemented separately from the AM. Consider the case of a water sensor that has the significant states ice, cold, normal, hot and boiling. This could be implemented by a management attribute of type integer giving the exact temperature. The significant states could be modelled by a range for each state. In addition this management attribute provides a version function that decides wether a change of the value is a new version. A new version of the management attribute equals the transition from one significant state to the other. Note that the AM implements the generic part of the system (calling the decision function, transferring the data), while the special parts (the decision function) is implemented by the application. This gives room for many different version predicates and associated decision functions. Some data type dependent delay predicates (e.g. threshold or delta predicates for integers) were implemented in our prototype, while other delay predicates that are based on the semantics of the date (e.g. version predicate) were implemented in the application.

The AM_M sends messages over the network to start and stop a delay predicate, to change the update strategy, to get the current update rate and to request a new value for an

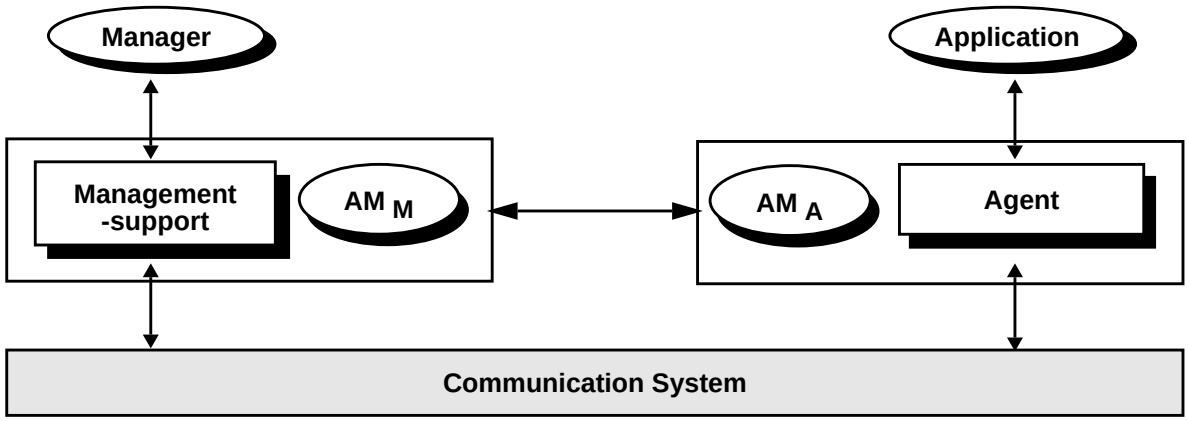

Figure 4: Management system with integrated actuality manager

attribute. The AM_A receives the requests and performs the necessary actions. Every time a management attribute needs an update, the AM_A triggers a change report to be sent to the management system. Management systems on different nodes are treated separately.

Due to the message based, asynchronous structure of the management system, it was very simple to add the two AM components. There were only a few changes to the internal message dispatching routine and to the functions that performs updates and accesses. If a manager accesses a value of a management attribute, the AM_M is asked wether this value is up-to-date. If the value must be fetched from the remote application, the request is triggered and the value of the attribute will be delivered to the manager later. On the other side the AM_A is informed of every change in an attribute. It then computes whether a change report has to be triggered or not. All other routines like periodic scheduling of updates or the exchange of update rates, are handled in the AM separate from rest of the management system.

Another observation was that due to their independence it should be possible to instantiate the AM and the management system on different node. The AM is then responsible to supervise management data on different nodes. This would increase the network traffic, but enable proxy-AM for dump devices or move the AM bookkeeping functionality to specialized nodes.

9. Use of delay predicates for service trading

A service trader ([8], [9]) stores information about services that are offered somewhere in the network. On demand of a service user the trader has to select an appropriate service, based on requirements, qualities and cost of the service. The trader has to verify that the offered service is still available and that the certain service qualities (like load factors or length of queues) are still valid. If a service provider vanishes without deleting its offer, a stale offer may remain. We used our AM to check the existence and the qualities of a service offer. A special attribute service_available could be tested to check the availability of the service. This attribute and the attributes that express qualities of the service are management attributes of our management system. For the service_available attribute we set a time and a change predicate. The time predicate ensures that there is an upper bound after that the availability is checked. The change predicate updates the availability attribute when the service is shut down. Other delay predicates are set for the quality attributes to base the service selection on up-to-date information. The values of these delay predicates are determined by the characteristics of the attributes itself. During the search for an appropriate service, the trader accesses the management system and asks for certain service attributes. He need not care about applying the right update strategy. Therefor the task of selecting a services is separated from the task of accessing the most recent version of the attributes.

10. Open problems and future work

Selecting the right update strategy currently depends on the access rate, the update rate and the requirements of the managing application. We want to extend this model to take different communication costs into account, depending on the topology and the used communication technology. Data that originates in the same network as the accessing manager, could be more often accessed than information that resides somewhere on the internet. A distance measure function may help to incorporate this aspect of distributed systems into our update strategies. For two managers residing on the same network and accessing the same remote value, it should be more effective to let one manager access the remote value and order the other to access the copy on the first site. At the end, this could lead to a system that automatically selects the nodes where copies of the data should be stored for reasons like load balancing, reduced network traffic on long distance lines and faster access.

While the system currently tries to guarantee the delay predicate, we want to examine the cases where the system does not try to fulfill every delay predicate every minute, but may decide to delay updates for a certain amount of time. This can be guided by probabilistic decisions based on the know values for the access and the update rate. Another reason to do this may be the case of a failures or network overload, where the management system should minimize the messages.

We currently used delay predicates to specify the actuality of monitored date. Another step would be to apply this kind of declarative programming to control operations that are executed on many different hosts. In this case the delay predicates may give thresholds after that the operation must be carried out.

11. Conclusion

We analyzed some characteristics of management data and developed a model for the delay between the data and the copy that is seen by the management application. We showed how this can be used to access the local copy, to reduce network traffic and to select an appropriate update strategy. We then showed that applications can express their requirements using delay predicates and showed how this can be used to separate the task of selecting the right update strategy from the management task itself. Delay predicates were incorporated into the *MELODY* management

system ([10], [11]) and used in the field of service trading. This showed the feasibility and the value of separating management tasks that access dynamic changing data, from the burden to select and implement the right updating strategy.

In the future we want to apply this techniques to automate some more management tasks, like deciding when to replicate management data for load-balancing reasons, selecting the right location to evaluate complex conditions, and to plan for migration of distributed system components to minimize network effects. Another area where a similar approach may be successful are distributed control operations that may be delayed until a given threshold.

Bibliography:

[1] C. E. Willis. Locating distributed information. In *Proceedings of the IEEE INFOCOM'89*, pages 303–311, IEEE Computer Society Press, April 1989. IEEE.

[2] R. Alonso, D. Barbara, and L. L. Cova. Augmenting availability on distributed file systems. Technical Report CS-TR-234-89, Princton University, Department of Computer Science, Princton, NJ 08544, October 1989.

[3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, September 1990.

[4] K. Marzullo, R. Cooper, M. D. Wood, and K. P. Birman. Tools for distributed application management. *COMPUTER*, August 1991.

[5] M. Stumm. Strategies for decentralized resource management. In *Proceedings ACM SIGCOMM-87 Workshop on Frontiers in Computer Communications Technology, Stowe, Vermont*, pages 245–253, August 1987.

[6] R. Washington and B. Hayes-Roth. Input data management in real-time ai systems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 250–255, Detroit, Michigan, USA, August 1989.

[7] T. K. Helbig. Strategies to meet demands on actuality (in german). Diplomthesis 944, University of Stuttgart, Institute for Parallel and Distributed High-Performance Systems, Stuttgart, December 1992.

[8] ANSA. Ansaware 3.0 implementation manual. Manual RM.097.01, Architecture Projects Management Limited, February 1991.

[9] ISO. Working document on topic 9.1 - odp trader. *Working paper of the ISO/IEC JTC1/SC21/WG7: N7047*, May 1992.

[10] K. Rothermel. Melody - a environment for the management of distributed systems (in german). *Proceedings of the Workshop: Development Trends in Computer Networks, Gaußig*, November 1991.

[11] I. Barth, E. Kovacs, and F. Sembach. Trading and management functions in melody (in german). *Proceedings of the workshop: Development Trends in Computer Networks, Gaußig*, November 1991.