# 33

# Service Management using up-to-date quality properties

*A. Kuepper, C. Popien, B. Meyer*
*Aachen University of Technology, Dept. of Computer Science IV*
*Ahornstr. 55, D-52056 Aachen, Germany, Tel.: 0241/8021415,*
*Fax: 0241/8888220, axel@i4.informatik.rwth-aachen.de*

## Abstract

The ever increasing growth of global computer networks is leading to an open service market which will rely on strategies for service trading. These services have to meet the requirements of their customers. The distributed platform ANSAware offers suitable facilities, but is too short of concepts that cope with the dynamic character of service properties. Therefore, the process of trading was extended by strategies providing an update of dynamic attributes during run time. These strategies are presented and compared, taking measurements into account. Furthermore, an architecture is described which supports the update of property values through recording, evaluation and processing.

## Keywords

Trading, dynamic properties, polling, caching, service management.

## 1 INTRODUCTION

The importance of distributed systems is determined by the growing number of interconnected computers on the one hand and by the availability of cost effectiv services on the other. The latter implies the creation of an open service market where a server offers several services which are used by a large number of clients. The complexity of distributed systems leads to the client's problem of locating suitable services. Therefore, the client/server model is extended to a three-party model. A so-called trader supports the binding between clients and servers at run time. To provide for the mediation of services, the trader primarily consists of a database to store service offers and their associated parameters.

Since different services of the same type may differ in some important aspects, each service is described by additional attributes containing the service property values. Upon selecting a service, the trader considers the client's demands concerning these properties. Thus, it is crucial that the trader holds up-to-date property values. This demand can only be met by using efficient methods providing the most recent values to avoid high network and trader utilization.

The Reference Model of Open Distributed Processing emphasizes this fact by distinguishing static and dynamic service properties (ISO/IEC, 1994; Popien, Schürmann, Weiß, 1995).

A. Schill et al. (eds.), *Distributed Platforms*

However, no solutions regarding update strategies are presented. Such strategies are supported by only a few distributed systems currently being offered. For example, commercial products like the Distributed Computing Environment (DCE) of the Open Software Foundation (OSF) (Schill, 1993) or the Common Object Request Broker (CORBA) of the Object Management Group do not include the trader concept. Admittedly, trading is well established with ANSAware of the Architecture Project Ltd. (APM), but updating of dynamic attributes is not supported.
This paper describes first experiences which have been made with an extended ANSAware-trader. The second chapter gives an overview of requirements of service trading described in terms of ANSAware. The notions of service, trader, factory and capsule are introduced. The third chapter discusses the implemented update strategies. The results obtained from measurements are compared and discussed. The fourth section presents a management application which is responsible for monitoring and evaluating the property values as well as supplying the trader with the obtained results. Finally, the last section derives the conclusions and lists some open questions.

## 2 THE DISTRIBUTED PLATFORM ANSAWARE

The basic components of ANSAware are *services*. A service is a function provided by an object at a *computational interface* (Spaniol, Popien, Meyer, 1994). That is, it is a set of capabilities available at an interface of this object. Very general, these capabilities consist of storing, processing and transfering of information. Every service is an instance of a *service type*. Associated with each service type is an *interface type*, which determines its computational behaviour. However, instances of the same service type differ in some noncomputational aspects. These additional aspects are called *service properties*. They will be discussed in more detail in the next chapters.

Objects that use a service are called *clients*, objects that provide a service are called *servers*. Every object can take over both roles. Services are subdivided into *application services*, which are specific to the client's requirements, and *architectural services*, which support the functionality of the distributed system. The most important architectural service of ANSAware is called trader, and is described below. Furthermore, the working of the so-called factory is presented.

A *trader* is an object that performs service trading, primarily satisfying indentification requirements (Popien, 1995). On the one hand, it is used by servers to advertise their services called *service export*. On the other hand, it is used by clients to locate a required service within the distributed system. This process is called *service import*. Within the trader, each service is represented by a service offer that is stored in the local trader database. Considering the trader in more detail, it performs two major functions: the type management function and the domain management function.

The *type management function* realizes the management of subtype relationships between types, that is, the set of all service types known by the trader is organized into a service type hierarchy, which is represented as a directed acyclic graph in which each node is a service type and each directed edge represents supertype to subtype relationships.

The *domain management function* manages the service offer space, which may be structured into so-called context-sets within a trader. A context structure is defined by a containment relationship between contexts. A trader context structure can be represented as a directed

acyclic graph with nodes representing the trading contexts, and arcs representing the containment relationship. A service offer is a member of a trader context and of any super context of it. A trader service offer space can reflect administrative structures and organizations.

A service offer is defined by its providing server declaring the service type, a context, the service properties and the interface reference.

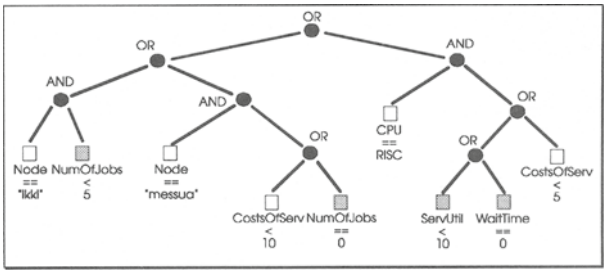


**Figure 1** Evaluation tree of property constraints.

A client wishing to import a service has to specify the requested service type and the context in which the offer is arranged. Moreover, it might specify requirements on the features of the service. These requirements are called property constraints and refer to property values. Upon receiving a client's request, the trader maps the property constraints into an evaluation tree, which is compared with the property values of suitable service offers. For example, the following term of property constraints is maped into the evaluation tree (cf. Fig.1.)

```
((((Node=='ikki') AND (NumOfJobs<5)) OR
((Node=='messua') AND ((CostOfServ<10) OR (NumOfJobs==0)))) OR
((CPU==RISC) AND (((ServUtil<10) OR (WaitTime==0)) OR (CostOfServ<5))))
->min[JobTime]
```

By evaluating these property constraints the trader selects a service which is optimal with respect to the property `JobTime` assuming that the previously defined requirements are fulfilled.

Each service is realized within a *capsule*, the unit of an autonomous operation within ANSAware. If ANSAware is supported by a multi-tasking operating system, each node may support several capsules. Each capsule is an autonomous process. Within a capsule, concurrency is supported through threads and tasks. The resources within a capsule are shared non-preemptively, that is, one thread will not let another run until it has voluntarily passed control to that new thread (Arch. Proj. Manag. Ltd, 1989).

Interactions between capsules are supported by the *remote execution protocol* (REX), a remote procedure call which supports two types of interactions. Initializing a *call*, the calling thread is blocked until a response is received in order to synchronize with the execution of the remote operation. To ensure reliable delivery, calls are retransmitted at regular intervals until an acknowledgement has been received. However, initializing a *cast*, the calling thread is blocked only until the message has been transmitted. Casts are not acknowledged or

retransmitted. The reliability of their delivery is determined by the underlying operating system and by the network.

Tasks and threads as well as REX are managed by an object called *nucleus* which is a part of the capsule. The nucleus provides the capsule with the node's resources, that is, a capsule consists of objects representing the application services, several transparency services and the nucleus. A *transparency service* manages the capsule's resources and communicates with its peers in other capsules to provide the required transparency. By using transparency services, the application programmer can delegate all responsibility for distribution to the underlying support environment (Arch. Proj. Man. Ltd., 1989).

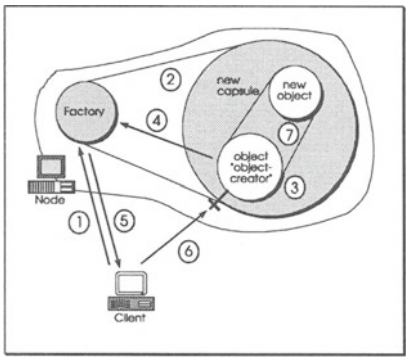


**Figure 2** Instantiating services using the factory.

ANSAware allows dynamic creation and termination of services at run time on behalf of potential clients of these services. The architectural service providing these features is called *factory*. Assuming that a factory is running on the node intended for the instantiation of a service, the factory works as can be seen in Fig. 2.

First, the client asks the factory to instantiate a new capsule (1). After this instantiation (2), the capsule comprises a single object providing the operation `Instantiate` (3). The interface reference of this object is returned to the factory (4) which forwards it to the client (5), which may then use this operation to create whatever objects it is interested in (6). Having received this invocation, the single object creates the desired objects within its capsule and returns the appropriate interface references to the client (Arch. Proj. Man. Ltd., 1989).

## 3 UPDATING DYNAMIC SERVICE ATTRIBUTES

Service properties are of overriding importance for the frequence a service is accessed, because clients specify their requirements on the features of a requested service through property constraints. Therefore, clients and server are very much interested in up-to-date attributes in the remote trader's database. This can only be achieved by special procedures which copy the values of the properties to the related attributes.

To update attributes, two new instances are introduced. The first one is a so-called *originator*, the features of which are described by service properties, and which is associated with one or more services. An originator is either a logical part of the distributed system like

the service itself or it is established in the environment as eg. the hardware or network components. Originators are not responsible for the management of service properties. Instances which are responsible for registration, investigation, updating and publication of service properties are called *publishers*. For example, a management application can be a publisher. In contrast to an originator, a publisher is always an instance of the distributed system. It observes, analyses and investigates the originator's properties and transmits them to the trader.

The service properties are subdivided into static and dynamic properties. *Static properties* never change during the life of the service offer. For instance, the capsule's process number or the name of the node are static properties. In contrast, *dynamic properties* may change frequently, base for instance on the utilization of a node or the number of jobs in a queue (Keller, 1993).

One of the major problems of supporting dynamic properties is to provide the trader with latest property values. For example, to update an attribute within the ANSAware trader the service offer must first be removed and then the service must be exported again with the new value. Since this method is not very efficient and time consuming, the usage of dynamic properties in ANSAware cannot be recommended. However, the consistency of static properties is of no importance, as their values do not change after service export.

To overcome this problem, ANSAware has been extended by mechanisms maintaining the updating of attributes. These mechanisms and their performance are presented in the following chapters. Similar methods were realized in the distributed platform MELODY (Wirag, 1994; Kovacs, 1994).

## 3.1 The polling of property values

The *polling* of property values is performed during the evaluation of property constraints. The trader requests the latest values from the publisher belonging to the concerning service offer. The process of polling works as follows: a client triggers a service import by specifying the property constraints and passing them to the trader. The trader puts together a list of all service offers which belong to the requested type and which are arranged in the specified context. From this list, the trader chooses suitable service offers by a procedure called *staggered evaluation*. First, it evaluates the constraints of the static properties. Only if the constraints are fulfilled regarding to these attributes, the values of the dynamic ones are requested from the publisher. By using the staggered evaluation, the trader obtains values as up-to-date as possible. Moreover, it avoids the unnecessary polling of values from service offers which are not suitable because of their static properties. Having received the latest values, the trader continues the evaluation of property constraints and transmits the suitable offers to the client.

If the property constraints must be evaluated for several service offers, the polling of attributes could be performed in two ways, called synchronous and asynchronous polling. By using the *synchronous polling* the trader initializes the request to the publisher and expects the answer before performing the evaluation tree for the next service offer. However, the *asynchronous polling* separates the process of initializing and waiting. It allows initialization of all requests before expecting the answers from the publishers. The trader is thus utilized in a more efficient way.

Fig. 3 shows the average trader service time for a synchronous, an asynchronous and a non-polling strategy. The results obtained depend on the number of service offers matching the

requested type, context and static properties. The results confirm the assumption that the asynchronous polling is more effective than the synchronous one. Moreover, the measurements emphasize that a polling-based trading is much more expensive than the evaluation of static attributes. The resulting costs must be paid by the client, which has to accept the increased service time. These costs must be considered in the following analyses. The advantages of polling must be weighed against the increased service time of the trader.

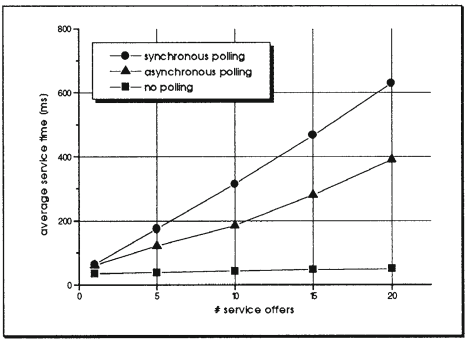


**Figure 3** Synchronous vs. asynchronous polling.

Another way to optimize the performance of the ANSAware trader is to evaluate the property constraints of several clients in parallel. For this purpose, the concurrency of the trader must be higher than one, that is, the trader has to run with several threads for the process of trading. Since each request from a client is allocated a separate thread, the property constraints of the next client can be evaluated, while waiting for the response of the previously initialized polling requests. This way, the non-preemptive multitasking within the capsule becomes apparent and the trader can be utilized much more efficiently.

Fig. 4 shows the average throughput of the trader running with different concurrencies against the number of service offers. The figure does not show large divergences between different concurrencies, except for a smaller number of service offers. These results can be explained as follows: short network delays in LANs cause short response times. Therefore, initialization of a large number of polling requests takes longer than the response of the publisher contacted first. After initializing all requests, the first publisher has already replied. No evaluation of other property constraints needs to take place. However, after initializing only a few requests, the replies often have not yet arrived. The trader can start the evaluation of the next property constraint. It is expected that running the trader with higher concurrencies is more effective in WANs.

To evaluate polling, a corresponding strategy was implemented to realize load sharing between participated servers in a distributed system based on ANSAware (Kuepper 1995). By using load sharing, the trader delegates the work from occupied servers to little loaded ones. Thus, it is possible to increase the system performance substantially. For the purposes of load sharing service offers of the same service type were combined with a dynamic attribute `UsageState` which indicates whether or not a service is working. Subsequently, a number of clients repeat the following procedure several times. First, the interface reference of the service is imported using the operation `lookup` combined with the constraint `'UsageState ==0'`. After getting the interface reference the service is called.

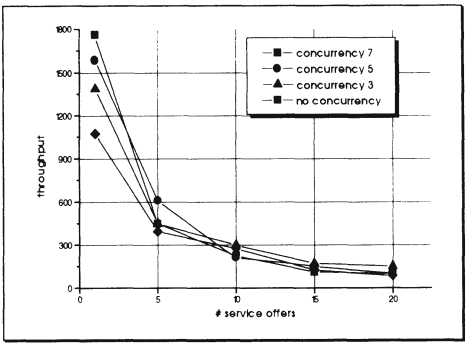


**Figure 4** Throughput of the trader running with different concurrencies.

Fig. 5 shows the average response times of a server achieved by a non-polling strategy, using the dynamic attribute `UsageState`. As the procedures were performed in a multi-user environment, two measurements have been carried out. The first measurement was performed at a small utilization of the participated nodes caused by users outside the distributed system, whereas the second one reflects the behaviour of the servers under high load. The figure shows that polling-based trading leads to reduced server response times and is therefore suited to increase the global system performance. This fact could be compensated or even overturned by the increasing trader service time mentioned above. Therefore, the following two methods represent a compromise between short trader service times and a selection of servers considering its latest property values.

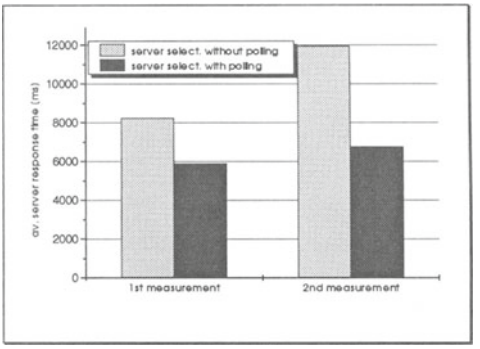


**Figure 5** Polling-based trading vs. non-polling-based trading.

The first method is called *random polling* and is based on a concept described in (Wolisz, Tschammer, 1993). Based on the staggered evaluation as described below, the client's property constraints are first matched against the static attributes of service offers. After this evaluation, N service offers remain in the list of suitable offers. From this list n<N offers are randomly

chosen to perform the polling. The other N-n service offers are not considered any more. After having received the current values, the most suitable of the n service offers is selected and returned to the client. By doing so, the trader service time can be reduced substantially. On the other hand, may yield offers which may on average be less optimal than those selected by the pure polling method.

In the following, this shall be clarified by further measurements. The load sharing experiment described below is repeated by using random polling. Fig. 6 shows the average server response time as well as this time plus the average trader response time for different values of n. n=1 corresponds to a selection without considering the attribute `UsageState`, whereas n=N corresponds to the pure polling method. The results obtained make clear that the n has a considerable influence on the server response time. By using a value of n=2, a significant decrease is achieved compared to not requesting `UsageState`. Considering the measurements, a value of n=6 is optimal. For values n>6 the gain achieved by decreasing server response times is compensated by the increasing trader response times, as it can be seen from the figure. The sum of both response times reaches a minimum at n=6. It should be pointed out that the results obtained from this experiment depend on the attribute `UsageState`. Using random polling for other attribute types may cause other results for an optimal n.

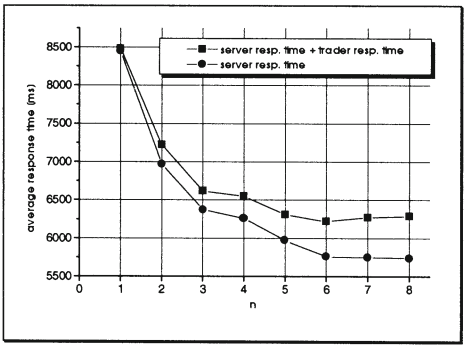


**Figure 6** Response times using random polling.

A further way to reduce the response time of the trader is the extension of each service offer with so-called *update predicates*. An update predicate gives a formal description of the required frequency of executing a polling request. After evaluating the static attributes of a service offer, the trader decides whether or not polling must be performed, taking into account the offer's update predicates. There are several kinds of update predicates, including for example a time predicate that allows polling only if the last request took place at least a certain number of time units ago. Moreover, it is possible to combine several update predicates by using conjunctions, disjunctions and negations.

Theoretically, there are no limits for defining such predicates. Even so, it is recommended that the choosen conditions be sensitive to the trader environment or the respective service offer. Furthermore, the trader must have knowledge of the information a predicate is based on.

For example, the trader has knowledge of an offer's access rate, i.e. how often it is compared with a client's property constraints, but it does not know the changing rate of a property.

## 3.2 The caching of property values

An alternative method to obtain current property values is called caching.Whereas polling is performed by the trader, caching is initialized by the publishers. Depending on certain conditions, the publisher transmits the latest values to the trader, which stores them in its local database. During the trading, the values stored are compared with the property constraints. Usually, the caching is initialized whenever a value of a property is changed. This may cause some problems like blocking the underlying network if the changing rates of properties are extremely high or if caching is applied by a large number of servers. Furthermore, the trader may become heavily loaded by updating service attributes and be no more able any more for serving further clients.

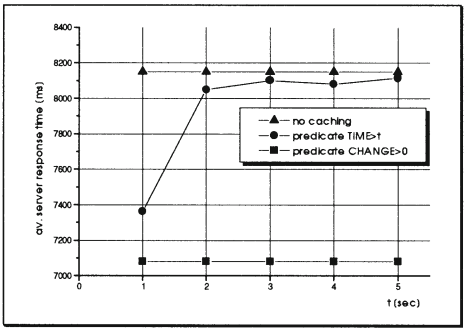


**Figure 7** Average server response time using caching.

To avoid problems like these, each property is provided with update predicates similar to those of service offers used for polling. The update predicates are hold by the publisher. Whenever a predicate allows updating the remote attribute, a transfer takes place. There are also several types of predicates like time predicates, change predicates or version predicates. Each predicate has to be based on information available for the publisher, so for example the frequencey of changing a property's value.

The load sharing experiment was executed by using a publisher-initialized transfer of `UsageState`. Fig. 7 shows the average response time of servers selected by a caching based trading with different update predicates. The predicate `TIME` causes the updating of the respective attribute every t time units. However, caching of a property controlled by `CHANGE` is performed whenever the property value changes. The measurements show that a non-delayed caching using `CHANGE` yields best results. Using a time predicate only makes sense, if the intervals choosen are sufficiently small. Otherwise, the achieved response time is not better than ignoring `UsageState`.

## 3.3 Polling vs Caching

The results make clear that each of the strategies polling and caching can be optimal under certain circumstances. If a service offer access rate is lower than the changing rate of its properties the polling of values should be preferred. However, if the change rate is lower, a caching strategy is required.

Because the mentioned rates are changing frequently, it is recommended to bind a strategy dynamically to a service offer taking the actual circumstances into account. This may cause problems since an instance is required which decides whether polling or caching must be performed. In the following, a method is presented where trader and publisher take this decision in cooperation.

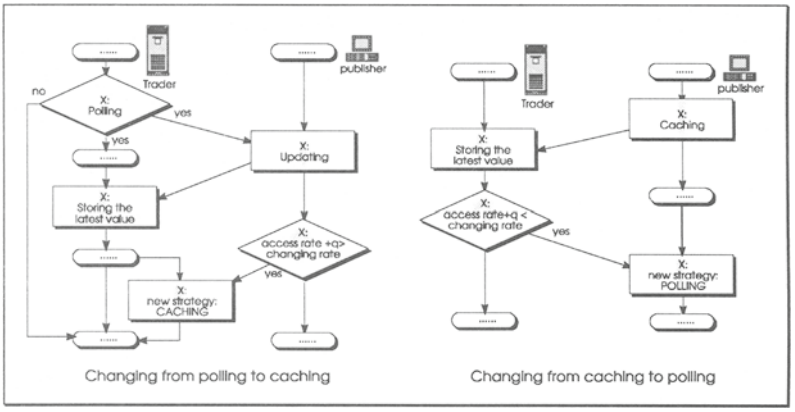


**Figure 8** Changing a strategy.

Changing a strategy assumes that the offer access rate and the change rate of its properties are available at the performing instance. By evaluating property constraints, the trader has knowledge of the access rate whereas the publisher is informed about the change rate. Thus, to induce a changing, both instances need information that is hold by the other. To solve this problem, the publisher derives the access rate from the number of polling requests performed in a time unit whereas the trader derives the changing rate from the number of updates performed in the course of using the caching strategy. Consequently, assuming that polling is performed, the publisher has to trigger the change from polling to caching, if the change rate is lower than the access rate. However, if the current strategy is caching, the trader has to induce the polling of requests since the access rate is lower than the change rate. This method is described in Fig. 8.

# 4 REALIZATION OF THE PUBLISHER

The previous chapters have pointed out the importance of the publisher for the polling and caching strategy. Moreover, distributed systems are of growing heterogenity and complexity,

which leads to a requirement for powerful and efficient management platforms. As publishers need access to administrative information which are related to service properties, the publishing functionality has been integrated into a management application under ANSAware. The realization of this management application with special regard to the publisher is explained in this chapter.

The management application acts as a server called manager, and thus it runs in an autonomous capsule. Each server of a distributed system is connected with exactly one manager. There are several possibilities for establishing a manager, depending on local environments. The favoured way is to instantiate exactly one manager on each node on which servers are running, in which case the information flow between manager and server will cause almost no delay if using the interprocess communication of UNIX. If this method fails, the manager must be established in the immediate vicinity of the server to guarantee a delay as small as possible. Managing a server, it must be bound to a manager first. This process is performed by the trader as a part of the export process.

After the export operation `Register` is called by a server (1), the trader establishes a service offer in its database. Subsequently, it checks the existence of a manager on the node in question or on another one nearby. If no manager is available, the trader tries to instantiate a new capsule by using a factory (2, 3). After getting this capsule's interface reference (4), the trader establishes the manager within the new capsule (5), and authorizes it for managing the server. As a result of `Register`, the trader returns the manager interface reference to the server. Now, manager and server are able to communicate. If a suitable manager already exists, steps 2 to 5 will be skipped.

At the manager, each server is represented as a *managed object* (MO) (Kuepper, Popien 1995; Popien, Kuepper, Meyer 1995). A MO contains a data structure which represents the service properties, and where the property values are stored. The set of all MOs forms the *management information base* (MIB). After authorizing the manager for the management of a particular server, it establishes a new MO and initializes a so-called *guard object* within its capsule. Each managed server has its own guard object. First of all, it is responsible for performing the caching by considering defined update predicates and by calling the operation `Update` of the trader. Furthermore, the guard object is in charge of controlling the server, which includes proof of its existence by sending test jobs for example.

The manager object constitutes another important object. It offers an operation `Event` which is called by the managed server if a predefined event occurs combined with a particular event code. For this purpose, `Event` has to be included at significant points within the server code. To guarantee a non-delayed working of the server, `Event` is realized by a cast. That means, the calling server is not blocked for receiving a confirmation. Because the manager in charge is located on the same node, the use of casts is sufficiently reliable. The manager object analyses the received event codes, schedules them and relates several events. The results are stored as property values in the appropriate MOs. For example, entering and leaving of a server are defined as results recorded by the manager. By counting and relating them, the queue length of a server can be determined, which is of importance for the mediation of this server. Moreover, the manager object is responsible for answering the trader's polling requests. The described architecture can be seen in Fig. 9.

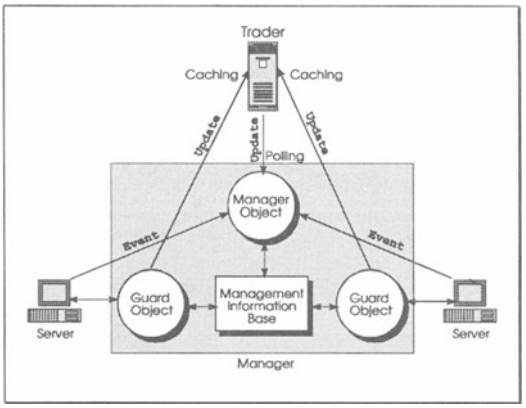


**Figure 9** Architecture and way of working of the manager.

## 5 CONCLUSIONS

Considering dynamic attributes during the process of trading requires powerful mechanisms within the trader as well as in the environment of the concerning server.

Within the context of trading the two main concepts polling and caching have been presented. The results of performed measurements have shown that the use of such concepts must be paid by increasing trader service times. The random polling as a modification of polling is capable to minimize the costs assuming that the suffering of resulting attribute quality is acceptable. Performing polling and caching under consideration of update predicates can be another way to decrease long trader service times. Because of the changing of environment circumstances like user frequency, attribute change rates and access rates, the mentioned strategies must be bound to attributes dynamically. To optimize the updating, it is necessary to perform further tests, especially taking a wide range of different attribute types into account.

A manager was established for the purpose of updating attributes. Furthermore, it is a first approach to deal with the requirement of managing large distributed systems caused by growing complexity and heterogenity. For example, the recording of special events used for getting property values is suitable for extending to an adequate monitoring system. Together with the factory and the node manager - which was not considered in this paper - the presented manager proposes a platform for further developments.

## 6 REFERENCES

Architecture Projects Management Ltd. (1989) The ANSA Reference Manual. Poseidon House, Castle Park, Cambridge, CB3 0RD, United Kingdom.

Architecture Projects Management Ltd. (1989) ANSA; An Engineer's Introduction to the Architecture. Poseidon House, Castle Park, Cambridge, CB3 0RD, United Kingdom.

Keller, L. (1993) From Name-Server to Trader - An Overview of Trading in Distributed Systems (in german). In : Journal PIK 16, pp. 122 - 133

Kovacs, E. (1994) Trading and Management of Distributed Applications: central tasks for Distributed Systems in future (in german). In: New Concepts of Open Distributed Processing, Aachener Beiträge zur Informatik, Bd. 7, pp. 57-66, Aachen

Kuepper, A. (1995) Studying dynamic attributes within service trading of ANSAware (in german), Diploma Thesis at Department of Computer Science, RWTH Aachen.

Kuepper, A.; Popien, C. (1995) A management scenario of Trading in Distributed Systems (in german). In: Communication in Distributed Systems, Springer, pp. 460-474

ISO/IEC JTC1/SC21 N8409 und ISO/IEC JTC1/SC 21 N 9122 (1994) Working Document - ODP Trading Function, Jan. 1994 bzw. Information Technology - Open Distributed Processing Trader.

Popien, C.; Kuepper, A; Meyer, B. (1995) A Formal Description of ODP Trading based on GDMO. In: Journal of Network and Systems Management, Plenum Press, New York and London.

Popien, C. (1995) Trading in Distributed Systems - Service Algebra, Service Management and Service Request Analysis (in german). TEUBNER-TEXTE zur Informatik, Bd. 12, Teubner-Verlag.

Popien, C.; Schürmann, G.; Weiß, K.-H. (1995) Distributed Processing in Open Systems: The ODP Reference Model (in german). Teubner-Verlag Stuttgart.

Schill, A. (1993) DCE - The OSF Distributed Computing Environment - Introduction and Foundations. Springer.

Spaniol, O.; Popien, C.; Meyer, B. (1994) Services and Service Trading in Client/Server Systems (in german). TAT 1, International Thomson Publishing.

Wirag, S. (1994) Dynamic Parameters within Service Selection (in german). Diploma Thesis at IPVR, University of Stuttgart.

Wolisz, A.; Tschammer, V. (1993) Performance aspects of trading in open distributed systems. Computer Communications, Vol. 16, No. 5, pp. 277-287.

## 7 BIOGRAPHY

**Axel Kuepper** studies computer science at Aachen University of Technology in Germany. He received his pre-diploma in 1993. Since 1994 he works at the Department of Computer Science where he is involved in distributed systems and network management research. He has submitted his diploma thesis entitled „Studying dynamic attributes within service trading of ANSAware".

**Claudia Popien** has studied mathematics and theoretical computer science in Leipzig Germany, Diploma 1989. After a research work at Technical University of Magdeburg she became an assistant at Aachen University of Technology in the Department of Computer Science in 1991. She finished her Ph. D. thesis entitled „Service trading in distributed systems - service algebra, service management and service request analysis" in 1994.

**Bernd Meyer** has studied computer science at University of Karlsruhe and Aachen University of Technology in Germany. In 1994 he got his diploma. Then he became a research assistant at the Department of Computer Science at Aachen University of Technology.