A Middleware Architecture for Scalable, QoS-Aware, and Self-Organizing Global Services[1]

Franz J. Hauck, Erich Meier, Ulrich Becker,
Martin Geier, Uwe Rastofer, and Martin Steckermeier

IMMD 4, Friedrich-Alexander University Erlangen-N rnberg, Germany

AspectIX@cs.fau.de

Abstract: Globally distributed services need more than location transparency. An implementation of such a service has to scale to millions of users from all over the world. Those users may have different and varying quality-of-service requirements that have to be considered for an appropriate distribution and installation of service components. The service also has to scale to thousands of administrative domains hosting those components. *AspectIX* is a novel middleware architecture which extends CORBA by a partitioned object model. A globally distributed service can be completely encapsulated into a single distributed object which contains not only all necessary components for scalability (e.g., caches and replicas) but also the knowledge for self-organization and distribution of the service. For distribution and installation of components, the service considers object-external policies to achieve administrative scalability.

1. Introduction

The Internet forms a large distributed system and one of its services, the World Wide Web, is probably the largest distributed service that has ever been built. The Web has some anarchic structure with limited flexibility and it is desirable to do better than the Web when it comes to globally distributed services. These services could span the various intranets of large companies or the whole Internet for serving users all over the world. With standard off-the-shelf middleware like CORBA implementations [20] those services can be modelled as distributed objects and be globally accessed using a worldwide unique object identifier. For a client, the service is completely location-transparent, i.e. the client does not need to know where the server object resides. Unfortunately, this does not scale to millions of users from all over the world, because a server object in CORBA can reside only at one place at a time. For geographical and numerical scalability the service has to be built out of multiple components using replication, caching and partitioning of code and data. With using CORBA the globally unique identifier of the service would be lost, because every component had to be implemented by an individual CORBA object with its own identity.

[1] This work was partially funded as project OVEST by the Bavarian Research Foundation, Sun Microsystems Munich, Siemens ATD Erlangen, and 3Soft Erlangen. The IBM Zurich Research Lab granted an IBM Research Award. The project has been funded by the German Research Community, DFG.

Partitioned object models have been adopted, e.g., in the *Globe* [25] and *SOS* [21] research projects. Both systems allow to combine the service components to a single distributed object with a single identity. Regardless where the client resides, it can bind to the distributed object and will get a local component which will become part of the whole service object. Some parts of the object may replicate or cache the object's data whereas others may just serve as stubs connecting to a replica or a caching component. Thus, such systems can encapsulate the components of a scalable distributed service in a single distributed object.

The users of a globally distributed system usually have different quality-of-service requirements when using the service, e.g., one user will heavily use the service and expect a certain throughput whereas another user will only invoke a few query methods and expect up-to-date answers. Neither CORBA nor Globe nor SOS sufficiently support quality of service. In case of a partitioned object model, the QoS requirements have to be considered for selecting an appropriate implementation for the object's local part at the client side, and even for building the complete internal structure of the object.

AspectIX is a novel middleware architecture extending CORBA by a partitioned object model combining the benefits of both worlds. Additionally, it supports a special per-object interface that allows a client to specify QoS requirements on the object's service. A policy-based mechanism encapsulates the decision process, e.g., where to place which part of the distributed object, in the object itself. Even dynamically varying requirements can be handled during run-time, e.g., by transparently replacing the implementation of the local part. The distributed object becomes self-contained and self-organizing. Different objects may have a completely different internal organization, which remains transparent to clients.

As globally distributed services will span over thousands of administrative domains it is necessary to give domain and application administrators some influence on the distribution and instantiation of object parts. Therefore for every *AspectIX* object, administrators can express policies that influence the selection of implementations, the choice of protocols and internal communication channels, etc. Our novel approach thus helps not only to achieve numerical and geographical but also administrative scalability [18].

This paper is organized as follows: Section 2 will identify the demands of globally distributed services. We show how currently available systems can be used to implement such services and uncover the deficiencies of these systems. In Section 3 our own architecture is introduced. Section 4 will compare our approach to other related work as far as it was not already mentioned in Section 2. In Section 5 we will give our conclusions and present our plans for future work.

2. Globally Distributed Services

2.1 Location Transparency

Location transparency means that regardless where the client and the service components reside, the client will be able to easily access the service. The easiest and most transparent way for a client is that the client just gets a location-independent object reference to the service (e.g., from a name service), binds to the service object, and uses the service by invoking methods. The client does not need to care about locations.

CORBA provides location-independent references in form of IORs [20]. The IOR contains at least one contact address of the object for clients, e.g., it contains a so-called IIOP[2]-address. As IIOP is based on TCP/IP, an IIOP address is just an Internet address and a port number, which together are unique on the Internet. The client will get a CORBA stub initialized with the IOR, and this stub will always contact the same server object, the one serving the IIOP address. Alternatively, a so-called implementation repository may be used to serve the IIOP address. It maintains a mapping to the current address of the server object and sends an IIOP location-forward message to the client which will use the returned actual address for subsequent calls. In case of a broken connection, the client will repeat the binding process. Thus, the implementation repository helps to hide the migration of server objects [7].

In both cases there is always one single instance which has a fixed location and cannot be moved without invalidating the IOR, the server object itself or the implementation repository. This single instance is not only a single point of failure but also a bottleneck in case of millions of worldwide users. Thus, CORBA objects cannot scale.

2.2 Scalability

For scalability, we need to structure the service by using replication, caching and partitioning of code and data. In a CORBA environment, our service will consist of multiple CORBA objects implementing replicas, caches and partial services. We would need to install all these objects around the world so that they can cooperate optimally. Now the client has to deal with many object references in order to invoke a method at the service. For hiding that complexity, we could introduce a single mediator which maps a unique service address to the right object references. Unfortunately, such a mediator (e.g., an enhanced implementation repository) will again be a bottleneck and a single point of failure, or has otherwise to be replicated which recursively applies the problem.

Partitioned object models as used by Globe [25] and SOS [21] solve that problem. They allow to combine multiple distributed parts into a single distributed object, which has a single identity. For example in Globe, a client can bind to a so-called *distributed shared object* and will get a *local object*. This local object becomes a part of the distributed shared object. It may replicate or cache the object's data whereas other local objects may just serve as stubs connecting to a replica or cache.

Both, Globe and SOS provide very similar frameworks for building the implementations of replicating and caching *local objects* [26, 12][3]. In the following, we will refer to Globe's framework: Application developers can program a pure nondistributed server object and combine it with layered consistency models [8]. The framework provides all other necessary sub-components of a local object. However, in case of caching it is restricted to the state of the entire object. So, a local cache cannot store semantics-dependent data, e.g., the results of query methods. A

[2]IIOP = Internet Inter-ORB Protocol.

[3]SOS's framework was named BOAR.

sophisticated location service delivers the contact information for a newly created local object so that stubs can find replicas and replicas can find each other [24].

Thus, a Globe object can encapsulate the components needed to build a scalable distributed service in a single object. However, it is unclear how the system determines which client gets which available implementation of a local object. If there are multiple available implementations the right choice is crucial for scalability. We believe that it is not feasible to allow the client to decide on that. Instead the local implementation has to conform to the needs of the object and its client.

2.3 Quality of Service

Clients often have different requirements with respect to the quality provided by the service. One client may want to heavily use the service and expects a certain throughput whereas another client may only invoke a few methods and expects up-to-date answers. In a locally distributed environment often a best effort service is enough for the clients, except they have very strict quality-of-service (QoS) requirements, e.g., for transmitting multimedia data. However, if a globally distributed service has no information about the client's expectations, it can only guess what best effort means for that client. Is it more important to achieve good throughput or is it more appropriate to get up-to-date results? An optimum for all of those aspects is not generally possible.

Thinking in terms of a partitioned object model it is even more important to know the clients' requirements because the choice for an implementation of the local part has severe influence on the quality of service that a client perceives. For example, if we choose a local replica we may have up-to-date results but perhaps only poor throughput due to the overhead imposed by the necessary synchronization with the other replicas. Neither Globe nor SOS provide any mechanisms for the client to express quality-of-service requirements.

With the *CORBA Messaging* document [19], the OMG adds some QoS support to CORBA. As CORBA only offers remote method invocation, the requirements are restricted to this communication scheme (e.g., priorities on requests) and cannot deal with general QoS requirements. CORBA extensions like *QuO* [28, 30] and *MAQS* [1, 2] provide interfaces to express quality-of-service requirements, but their implementations focus on QoS characteristics that can be implemented independently of the object's semantics. We believe that this does not help for scalability, e.g, caches cannot be implemented independently of the object semantics.

3. *AspectIX* Middleware Architecture

AspectIX is our novel middleware architecture which extends CORBA by a partitioned object model. First, we explain our CORBA extensions. Then, we will introduce our QoS interface and present how *AspectIX* encapsulates decisions concerning the object's internal structure. Finally, we introduce administrative policies that influence the object's decisions and achieve administrative scalability.

3.1 Partitioned Object Model

In *AspectIX*, a distributed object is partitioned into *fragments* [5]. Clients need a local fragment to invoke methods at a distributed object. Access to a fragment, and to the distributed object respectively, is provided by fragment interfaces connected to a fragment (see Fig. 3.1). When a client binds to a distributed object, the CORBA IOR is evaluated, an implementation for a local fragment is chosen and loaded. Finally, the fragment is connected to its fragment interfaces. Fragment interfaces are automatically derived from CORBA IDL descriptions. As long as the client just binds to an object and invokes methods, the client will not see any differences to CORBA. Standard CORBA objects can be accessed by *AspectIX* objects. Conceptually, for ordinary CORBA objects there is also a local fragment, but it is nothing else but the standard CORBA stub, which is automatically generated. *AspectIX* can also host ordinary CORBA servants.[4]

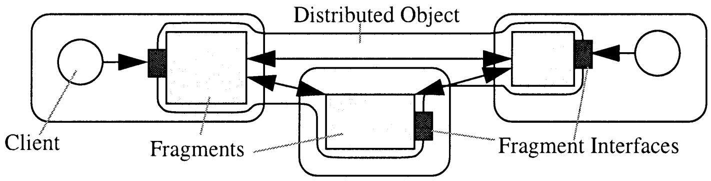

Fig. 3.1 A distributed object with three fragments each placed on a different **host.**

The fragments of a distributed object usually need to communicate. Therefore, *AspectIX* provides so-called communication end points (CEPs), which are similar to sockets but part of *AspectIX*. A fragment can open such a CEP and attach a stack of predefined protocols to it. There are three different kinds of CEPs: connectionless, connection-oriented and RPC-based CEPs. For example, there is a protocol stack GIOP over TCP/IP, which implements IIOP. This stack is used together with an RPC-based CEP to implement standard CORBA stubs and skeletons. However, fragments may also use datagram, stream or multicast communication implemented by various protocols to meet the object's requirements, e.g., to update the state of all replicating fragments. With the *AspectIX* CEPs, fragment developers do not need to use other and nonportable communication mechanisms, e.g., from the operating system. We imagine that an *AspectIX* ORB may download necessary protocol modules on demand from an external repository, but this is beyond the scope of this paper.

Let us consider a simple, global information service that is supposed to implement a parts list for one of a company's products. Using the interface we can enumerate the parts, and for each part number we can query a description string and a price. Some update methods are provided for filling in and correcting the data. First, a developer will describe the interface in CORBA IDL. Then, he will design two different fragment implementations: a server and a stub fragment. The latter is automatically provided by a tool, e.g., an IDL compiler. So far, the design is the same as on a CORBA system. As the developer knows that the service may have millions of users, he has to take care of scalability. Thus, he provides two additional fragment implementations: a replicating and a caching fragment[5]. The caching fragment has the same functionality as the stub fragment, but can store query results in a local cache.

[4]This is especially interesting if only some of the application objects need to be truly partitioned, e.g., if a legacy application is ported to *AspectIX*.

[5]Instead of hand-coding the replicating fragment, the replication framework of Globe [26] or BOAR [12] could be used to create it.

Stub and cache fragments can contact either a server or a replica. Replicas are connected by some internal update protocol depending on the chosen consistency model. Object-internal contact addresses are provided by a location service, e.g., the Globe Location Service [24].

3.2 Quality of Service

AspectIX supports a special per-fragment interface that allows the client to specify QoS requirements on the object's service [6]. We adopt the term *aspect* to describe nonfunctional properties of a distributed object, e.g., QoS requirements. But we also consider hints to be aspect configurations of an object , e.g., about the usage pattern of the client. A client can provide aspect configurations on a per-fragment basis, but only for aspects supported by the distributed object. Each aspect configuration has a globally unique name (perhaps maintained by some standards authority). A client can retrieve the names of supported aspects by using the special interface.

In our example, the developer of the information service object chose to support three different aspects: actuality of the returned data, read access characteristics, and lifetime of the object binding. Aspect configurations are represented as objects described in CORBA IDL. The developer of our service may reuse existing specifications, e.g., those described in Fig. 3.2. The *data actuality* aspect can configure how long the returned data can be out of date defined by a maximal age of the data since the last validation. The *access characteristic* can distinguish rare or continuous read access to the object. The *binding lifetime* aspect can be configured for an expected *long* or *short* time that the client wants to keep the local fragment, or the binding to the object respectively. For the sake of brevity, we rather simplified the aspects. However, it is possible to make them as precise as needed by introducing additional attributes.

```
interface DataActuality {
attribute unsigned long maximalAge;// in milliseconds
};

enum AccessPattern { continous, rare };
interface ReadAccessCharacteristic {
                    attribute AccessPattern pattern;
};

enum Lifetime { _short, _long };
interface BindingLifetime {
                    attribute Lifetime lifetime;
};
```

Fig. 3.2 IDL description of three different aspects.

For setting a configuration, the client can create his own aspect configuration objects and initialize them accordingly. Finally, he will use the distributed object's aspect interface to pass those objects into the local fragment. The fragment then has to fulfill the requirements. Hints can be used to optimize fragment-internal processes.

If the local fragment cannot fulfill the requirement, the aspect configurations become invalid. As soon as the fragment detects that it can fulfill the configurations, they become valid again. These transitions can be signalled to the client via a callback interface. So, the client could try to use another, perhaps less strict, configuration. Additionally, the client can configure how method invocations are handled in case of invalid aspect configurations. The invocations can be blocked as long as the configuration remains invalid (useful when requirements are needed in any case, e.g., communication has to be encrypted). They can raise a run-time exception, signalling the invalid configuration to the client, or they can ignore the invalidity and proceed as usual.

If a fragment detects that it cannot fulfill the requirements it might know another fragment implementation that will do. In this case, the local implementation can be transparently replaced by the other one. In any case, the fragment interfaces and the local aspect configurations remain the same.

3.3 Self-Organization

AspectIX objects should be able to self-organize their internal structure. The internal structure of a service and its development over time should be encapsulated in the object. This urges the object developer to make certain structural decisions inside of fragment implementations. *AspectIX* supports the programmer by asking him first to strictly and carefully separate mechanisms from policies. Mechanisms have to be implemented inside of fragment implementations. Then, the developer has to define decision points at which a certain decision concerning a mechanism has to be made (e.g., "Which protocol shall I use?", "Which fragment implementation shall I load?"). Instead of encoding this decision into the fragment, the developer formulates a decision request and provides policy rules. The decision request is delegated to a decision subsystem provided by *AspectIX*. The central component of this subsystem consists of a policy engine [14]. The policy engine has access to policy rules and all parameters that may influence a decision. These parameters include the requirements set up by the client via aspect configurations and environment conditions of the system. The actual decision is found by the decision engine consulting the policy rules.

We separate two sets of policy rules: developer and default policies. The first set contains rules defining under which conditions the fragments will operate properly. The second set describes rules for a default behavior of the object. The distinction between the two sets has to be carefully made to allow flexible extensions of the policy system as we will see in the next section. For a decision, the policy engine will first consult the developer policies. Those can decide not to make a definitive decision and to delegate the decision to lower prioritized rule sets instead, e.g., to the default policies. When the delegate rule sets made their decision, these decisions can be checked and possibly overwritten by the rule that initiated the delegation.

Policy rules can access the aspect configurations of a client and they can lookup environment variables of the runtime system. Examples are the currently available size of virtual memory or stable storage, and the available network bandwidth. External services, that hide the platform-dependent issues, provide this information to the policy engine. Other external services that may be needed for policy rule evaluation include naming, directory, location and security services (e.g., DNS [15], Globe Location Service [24], and PolicyMaker [3]).

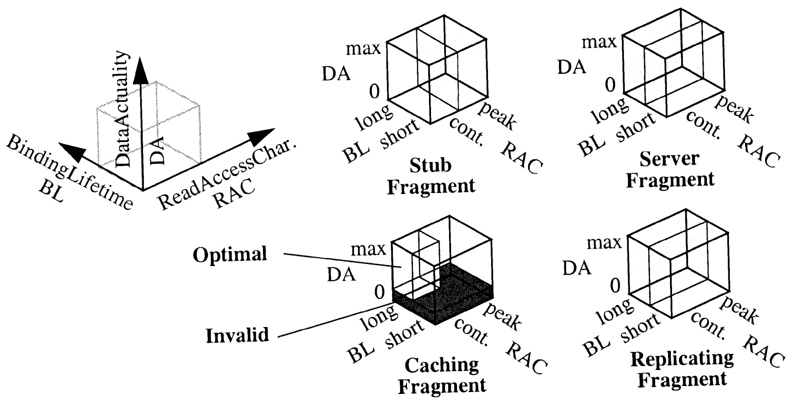

Fig. 3.3 Optimal and invalid aspect configurations for the fragments of a simple information service.

Coming back to our simple information service, we now have four different fragment implementations and three different, supported aspects, of which one is a true QoS requirement. First of all, the developer finds out for which aspect configurations the fragments can work well or cannot work at all. The result for our service is displayed in Fig. 3.3. A stub fragment makes most sense when the client does not use the object continuously. Server and replicating fragment make most sense when their lifetime is not too short. A caching fragment is best suited for continuos usage with data actuality greater than a certain limit. Below that limit the cache would always have to verify returned results. This is not optimal and our developer decided not to allow it, because it implies extra latency. Finally, the cache's lifetime should not be too short.[6]

There are other restrictions that the developer has to set up, e.g., there can be either a single server fragment or several replica fragments. So, the developer has to define how one fragment implementation can be replaced by another. In our case this is easy for most of the possibilities because they do not need any state transfer from old to new fragment implementation. The only exceptions are replacing server and replica fragments by each other, which needs some transfer of state. This is implemented by some internal hand-over interface that both server and replica have to provide.

For an implementation, the developer has to separate mechanisms from policies. For our example we first focus on two decision points and the decision they need:

- Binding of a client to an object: Which fragment implementation should be loaded?
- Replacing the local fragment: Which fragment implementation should be loaded?

The underlying mechanism is the loading of a new fragment implementation. The first decision point is inside of the *AspectIX* middleware, the second inside of a fragment implementation. Both will need the same decision, which is formulated as

[6]A careful reader may have noticed that none of the fragments is optimal for continuos usage for a short time, but we could have expected that because such a configuration would be hardly meaningful.

an appropriate decision request to the decision subsystem: `DecisionRequest (needs FragmentType)`

For supporting this decision the developer has to write policy rules. As *AspectIX* aims at providing generic support for policies, we have decided to use a general purpose programming language to formulate policy rules in our prototype. Specific—and thus problem-dependent—policy description languages would limit the expressiveness to their specific problem domain. For the sake of simplicity, we will only use an abstract and more readable representation of policies in this paper. A policy rule consists basically of a signature describing the possible result of a decision (named “provides” clause), a signature describing the dependencies of the rule (“needs” clauses), and the actual rule. The latter is separated into a “decide” clause containing the decision and a “check” clause that can verify and correct a decision returned by a potential delegation.

A policy decision request only consists of a “needs” clause. The policy engine will search for policy rules that generate the desired decision. Their “needs” clauses are satisfied by recursively searching for other policies that provide the needed

```
  provides:                      FragmentType
  needs:                       FirstFragmentType,
ReplicaAllowed, CacheAllowed
  decide:                      if(  #Fragments  ==  0  )  then
FragmentType             =              FirstFragmentType
                               else delegate
  check:                       if( (FragmentType == REPLICA &&
ReplicaAllowed          ==          FALSE)          ||
                                 (FragmentType   ==   CACHE   &&
CacheAllowed           ==          FALSE)          ||
                                FragmentType == UNKNOWN) then
FragmentType = STUB

  provides:                    FirstFragmentType
  decide:                      delegate
  check:                       if( FirstFragmentType != REPLICA
&&        FirstFragmentType       !=        SERVER        )
                               then FirstFragmentType = SERVER

  provides:                    ReplicaAllowed
  decide:                      ReplicaAllowed = TRUE; delegate
  check:                       if( #Replicas >= MaxReplicas ||
                                 #Replicas    ==    0    &&
this.FragmentType          !=          SERVER          )
                               then ReplicaAllowed = FALSE

  provides:                    CacheAllowed
  decide:                      CacheAllowed = TRUE; delegate
  check:                       if(     aspect(DA.maxAge)     <
MinAgeCache ) then CacheAllowed = FALSE
```

Fig. 3.4 Developer policies for a simple information service.

information. Under-specification and cycles are currently detected and signalled as errors[7]. Over-specification is solved by priorities.

The developer policies describe the capabilities and restrictions of the various fragment types as outlined in Fig. 3.4. The first rule implements the decision requests for a fragment type which is based on the other rules. The second rule is for determining the type of the very first fragment. The third policy rule describes when a replica is allowed. It implements an upper bound for the number of replicas, e.g., imposed by the used consistency model and protocols. The first replica can only be created from a server fragment. The fourth rule takes care that caches are not used when the maximal age required by the client is below a certain limit. To enable the policy engine to give satisfying answers to the decision requests, we have also to provide default policies as outlined in Fig. 3.5. These rules provide a default decision on fragment types. They choose the optimal fragment type for an aspect configuration and are only used when the corresponding developer policy delegates its decision.

So far, the object will start with a server object and clients will bind with a stub or

```
  provides:                     FragmentType
  needs:                      ReplicaAllowed
  decide:                     if(   aspect(BL.lifetime)    ==
_short ) then FragmentType = STUB
                              elsif(    ReplicaAllowed     &&
aspect(RAC.pattern)==rare                                   )
                              then  FragmentType   =   REPLICA
                              else FragmentType = CACHE

  provides:                   FirstFragmentType
  decide:                     FirstFragmentType = SERVER
```

Fig. 3.5 Default policies for a simple information service.

cache fragment, but the object will never deploy replicating fragments. To extend the objects structural self-organization we assume that in most fragment implementations there is a so-called QoS manager running that constantly monitors certain system conditions and the delivered quality of service. In our example, if the QoS manager detects a high load in form of local invocations, it uses additional decision requests to decide on the creation of new replicas, and initially on replacing the server by the first replica. The decision point and the corresponding decisions are:

- QoS manager detects high load: Shall I create a replica? Where?

The server's QoS manager will request a decision on `ReplicaRequired` and if true the server will replace itself by a replica. The replica's QoS manager will request the same decision and if a new replica is required it requests for an additional `ReplicaLocation` decision. The creation of additional replicas is supported by so-called dwelling services. These are ordinary distributed objects that can be requested to bind to another object at a certain place and to set a certain aspect configuration. With this binding, a local fragment will be created which can be a local replica depending on the `FragmentType` decision. The additional developer and default policies are outlined in Fig. 3.6 and Fig. 3.7.

[7]Automatic cycle recovery is subject of current research.

```
  provides:                          ReplicaRequired
  needs:                           ReplicaAllowed
  decide:                          delegate
  check:                           if( ReplicaRequired == TRUE &&
ReplicaAllowed                 ==              FALSE             )
                                   then ReplicaRequired = FALSE
```

Fig. 3.6 Additional developer policies.

```
provides:                          ReplicaRequired
needs:                           LoadTooHigh
decide:                            if( LoadTooHigh == TRUE ) then
      ReplicaRequired                    =                     TRUE
                                   else ReplicaRequired = FALSE

provides:                          LoadTooHigh
decide:                            if( service(SystemLoad) > MaxLoad
      )          then               LoadTooHigh          =        TRUE
                                   else LoadTooHigh = FALSE

provides:                          ReplicaLocation
decide:                            ReplicaLocation     =     service(
      Trader.findNearestDwellingService )
```

Fig. 3.7 Additional default policies.

It is clear that this example is somewhat simplified as otherwise we could not explain
It in this paper. However, our policy system is able to deal with much more complex policy decisions and policy rules as they are necessary in a completely self-organizing globally distributed service. With the definition of developer and default policies, the fragment developer is enabled to separate policies from mechanisms. Decision code is not scattered over the fragment's implementation but collected in form of policy rule sets. This makes the development of fragment implementations much easier and every object instance may encapsulate its own policy decision and rules, which remains transparent to clients.

3.4 Administrative Scalability

The developer alone can not anticipate the optimal service configuration for all possible situations. Thus, we have to introduce further individuals and allow them to use their knowledge for tailoring the service according to their specific needs. To achieve that, we adopt policy concepts that are commonly used in system management [23,13]. We have identified three additional classes of individuals that should be allowed to formulate additional policy rule sets: application administrators, domain administrators and users. As we have adopted role concepts [4] when doing this analysis, we call them *role classes*.

Individuals belonging to the application administrator role class are driven by business goals. They are responsible for the service as a whole and have knowledge about the structure and the characteristics of the data that is processed by the service. Domain administrators have a totally different view. Their influence is limited to their local domain, where they are responsible for all facets of system management. This includes knowledge about the local network topology, available computing resources and security demands. Finally, users should also be enabled to define policies. As it should be avoided that users are required to have some internal knowledge about the distributed service, their influence should normally be limited to the definition of user preferences, i.e. the selection of their favorite text processor or Web browser.

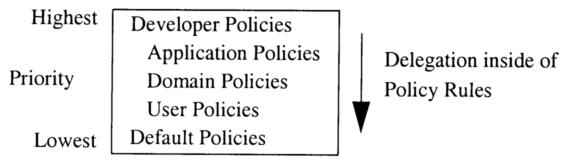

Fig. 3.8 Priority of the policy rule sets.

The role classes define an inherent priority scheme on their rule sets as depicted in Fig. 3.8, e.g., no user is allowed to override domain policies. On the other hand, an application policy is able to delegate decisions to appropriate domain or user policies. Developer policies still own the highest priority within the system whereas default policies can be overridden by any other policy.

```
provides:                     FirstFragmentType
decide:                     FirstFragmentType = REPLICA
```

Fig. 3.9 An exemplary application policy.

```
provides:                   ReplicaAllowed
decide:                     if(    service(LocalStorage)    >
      MinStorage                                            )
                            then ReplicaAllowed  =  TRUE  else
      ReplicaAllowed = FALSE
```

Fig. 3.10 An exemplary domain policy.

In our exemplary information service, an application administrator who knows that a large number of users will use this service can decide that the first instantiated fragment should already be a replica. As shown in Fig. 3.9, he can override the default policy `FirstFragmentType` by an application policy. Similarly, a domain administrator can adapt the service to the local circumstances by requiring a lower limit of available storage for the creation of a replica. He takes advantage of the delegation in the `ReplicaAllowed` developer policy and defines his own policy rule shown in Fig. 3.10.

When applying these concepts to globally distributed services, the system must be capable of supporting a large number of policy rules. This is achieved by exploiting locality. All individuals belonging to role classes are scattered all over the distributed system and their political standpoints often depend on the administrative domain they belong to. Moreover, not all policies are valid for every distributed service. Thus, the validity of most policies is limited to administrative domains or certain classes of services. With the introduction of *validity annotations* for every rule that can be based on principals, on the type of the distributed services, and on administrative domains, the number of policies can be reduced to the required minimum when making decisions. Validity annotations also allow improvements of the distribution strategies that are applied when a large number of policies have to be distributed within the service. Together with this potential for numerical and geographical scalability [18], a large number of different policies—and therefore a large number of different domains—can be supported. *AspectIX* thus achieves administrative scalability by offering political influence on an object's internal decision processes to a large number of individuals.

4. Related Work

As explained in Section 2, *SOS* and *Globe* use a partitioned object model. In SOS, all intra-object communication is either modelled by so-called channels or by other predefined fragmented objects. The *AspectIX* approach of communication end points allows more flexibility (e.g., legacy code can be accessed by standard protocols). Globe and SOS neither consider QoS requirements nor address how a distributed object is organized and maintained. *AspectIX* provides both, QoS requirements in form of aspect configurations and a sophisticated architecture for object-internal decision processing. This allows administrative scalability and flexible, self-organizing objects.

QuO [28, 30] and *MAQS* [1, 2] extend CORBA and support quality-of-service requirements. MAQS can only deal with one-dimensional QoS requirements, whereas *AspectIX* objects can support arbitrary aspect-configuration sets. Both, QuO and MAQS use a local object at the client side to implement the QoS requirements (called delegate or mediator). As this object is only QoS-related but not to the object's semantics it is difficult to integrate functional properties with QoS. This will be necessary if we like to have a client-side local cache to gain performance but less data actuality.

The majority of work within the policy area uses policies as a concept for system management [23, 13]. Much work about the definition of policies [9, 16], policy hierarchies [29], conflict management [11], and the use of roles [4, 10] is available. We used these results as a basis for exploring administrative scalability. The main difference between *AspectIX* and those systems is, that we also integrate the developer into the policy definition process. By supporting the separation of mechanism and policy at a very early stage of the service's software design, we achieve a tighter integration of service design and service management than other approaches.

An IETF workgroup currently defines a policy framework for the management of network systems [27]. They have defined a set of classes, that allow policies to be defined and stored [17] and also employ role concepts to map policies to specific components. The IETF policy framework offers little abstraction for the policy programmer. Due to its concentration on a specific problem domain, it does not provide a generic solution as we do.

5. Conclusion and Future Work

We presented the novel middleware architecture *AspectIX*. Its partitioned object model allows an object developer to partition a single service object into multiple distributed components, which can be deployed for building numerically and geographically scalable distributed services. The *AspectIX* policy subsystem enables a developer to strictly separate mechanisms from policies. The developer has to implement the mechanisms, to identify the decision points and to express the capabilities, restrictions and the default behavior in policy rules. Administrative scalability is achieved by allowing additional administrator and user policies that can influence the object's decision. The policy engine itself can be made scalable as rule sets have some local area of validity and as the relevant rules can be easily found for a decision. Thus, with using *AspectIX* scalable and completely self-organizing, globally distributed objects can be designed and operated.

In this paper we presented only QoS requirements that deal with scalability. In principle, all kinds of requirements can be handled and implemented in multiple fragment implementation. This is subject of further research.

So far, we have multiple prototype implementations validating our concepts: a nondistributed prototype validating the object model and implementing the replacement of local fragments, an implementation of communication end points, and a nondistributed prototype of our policy subsystem that is able to find the appropriate decision with policy rule sets applied. In the near future, we will build an integrated prototype in Java. We are also working on a development environment that will support the developer in designing policy rules and implementing fragments.

References

1. C. R. Becker, K. Geihs: QoS as a competitive advantage for distributed object systems: from enterprise objects to global electronic market. *Proc. of the 3rd Int. Enterprise Distr. Obj. Comp. Conf.*—EDOC (La Jolla, Cal.), 1998.
2. —: Generic QoS specifications for CORBA. *Proc. of the KiVS Conf., Kommunikation in Verteilten Systemen* (Darmstadt, March 1999), Springer, Inform. aktuell, 1999; pp. 184–195.
3. M. Blaze, J. Feigenbaum, J. Lacy: Decentralized trust management. *Proc. of the 1996 Symp. on Security and Privacy*, IEEE, Los Alamitos, Cal., May 1996; pp. 164-173.
4. D. Ferraiolo, R. Kuhn: Role-based access control. *Proc. of the 15th National Comp. Security Conf.*, 1992.
5. F. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, M. Steckermeier: AspectIX: A middleware for aspect-oriented programming. *Object-Oriented Technology,* ECOOP'98 Workshop Reader, LNCS 1543, Springer, 1998.
6. —: *The AspectIX approach to quality-of-service integration into CORBA*. Tech. Report TR-I4-99-09, IMMD 4, Univ. Erlangen-Nürnberg, Oct. 1999.
7. M. Henning: Binding, migration, and scalability in CORBA. *Comm. of the ACM* 41(10). ACM, New York, NY. Oct. 1998; pp. 62-71.
8. M. Kermarrec, I. Kuz, M. van Steen, A. S. Tanenbaum: A framework for consistent, replicated Web objects. *Proc. of the 18th Int. Conf. on Distr. Comp. Sys.*—ICDCS (Amsterdam, May 1998).

9. T. Koch, C. Krell, B. Krämer: Policy definition language for automated management of distributed systems. *Proc. of 2nd Int. Workshop on Sys. Mgmt*, IEEE, Toronto, June 1996.
10. E. C. Lupu, M. S. Sloman: Towards a role-based framework for distributed systems management. *J. of Network and Sys. Management* **5**(1), Plenum Press, 1997.
11. —: Conflicts in policy-based distributed systems management. *IEEE Trans. on Softw. Eng.*—Spec. Issue on Inconsistency Management, 1999.
12. M. Makpangou, Y. Gourhant, M. Shapiro: BOAR: a library of fragmented object types for distributed abstractions. *Int. Workshop on Obj. Orientation in Operating Sys.*—I-WOOOS (Palo Alto, Cal., Oct. 1991).
13. M. J. Masullo, S. B. Calo: Policy management: an architecture and approach. *Proc. of IEEE Workshop on Sys. Management*, UCLA, Cal., April 1993.
14. E. Meier, F. Hauck: *Policy-enabled applications*. Tech. Report TR-I4-99-05, IMMD 4, Univ. Erlangen-Nürnberg, July 1999.
15. P.V. Mockapetris: *Domain names—concepts and facilities*. RFC 1034, Nov. 1987.
16. J. D. Moffet: Specification of management policies and discretionary access control. M. S. Sloman (Ed.): *Mgmt. of Distr. Sys. and Comp. Netw.*, Addison-Wesley, 1994, pp. 455–480.
17. Moore, E. Ellesson, J. Strassner: *Policy framework core information model*—Ver. 1 Specification. Internet-Draft, Work in Progress, Jan. 2000.
18. C. Neuman: Scale in distributed systems: T. L. Casavant, M. Singhal (Eds.): *Readings in Distributed Computing Systems*. IEEE Comp. Soc., Los Alamitos, Cal., 1994; pp. 463–489.
19. Object Management Group, OMG: *CORBA Messaging*. OMG Doc. orbos/98-05-05, 1998.
20. —: The Common Object Request Broker: architecture and specification. Rev. 2.3.1, OMG Doc. formal/99-10-07, Oct. 1999.
21. M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, C. Valot: SOS: an object-oriented operating system. *USENIX Comp. Sys.* **2**(4), 1989; pp. 287–337.
22. M. Shapiro: Structure and encapsulation in distributed systems: the proxy principle. *Proc. of the 6th Int. Conf. on Distr. Comp. Sys.*—ICDCS (Cambridge, Mass., May 19-23, 1986), IEEE Comp. Soc.,Wash., DC, 1986; pp 198–205.
23. M. S. Sloman: Policy driven management for distributed systems. *J. of Netw. and Sys. Management* **2**(4), Plenum Press, 1994.
24. M. van Steen, F. Hauck, P. Homburg, A. S. Tanenbaum: Locating objects in wide-area systems. *IEEE Comm. Magazine* **36**(1). Jan. 1998; pp. 104–109.
25. M. van Steen, P. Homburg, A. S. Tanenbaum: Globe—a wide-area distributed system. *IEEE Concurrency*, Jan.–March 1999; pp. 70–78.
26. M. van Steen, A. S. Tanenbaum, I. Kuz, H. J. Sips. A scalable middleware solution for advanced wide-area Web services. *Dist. Sys. Eng.* **6**(1). March 1999; pp. 34–42.
27. M. Stevens, W. Weiss, H. Mahon, B. Moore, J. Strassner, G. Waters, A. Westerinen, J. Wheeler: *Policy Framework*. Internet-Draft, Work in Progress, Sept. 1999.
28. R. Vanegas, J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz: QuO's runtime support for quality of service in distributed objects. *Proc. of the Int. Conf. on Distr. Sys. Platforms and ODP, Middleware '98* (The Lake District, UK), Springer, 1998.

29. R. Wies: Using a classification of management policies for policy specification and policy transformation. *Proc. of the IFIP/IEEE Symp. on Integr. Netw. Mgmt.* Santa Barbara, 1995.
30. J. A. Zinky, D. E. Bakken, R. E. Schantz: Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Sys.* **3**(1), 1997.