Decentralized, Adaptive Services: The AspectIX Approach for a Flexible and Secure Grid Environment

Rüdiger Kapitza[1], Franz J. Hauck[2], and Hans Reiser[1]

[1] Dept. of Comp. Sciences 4, University of Erlangen-Nürnberg, Germany
{rrkapitz,reiser}@cs.fau.de
[2] Distributed Systems Lab, University of Ulm, Germany
hauck@informatik.uni-ulm.de

Abstract. In this paper we present EDAS, an environment for decentralized, adaptive services. This environment offers flexible service models based on distributed mobile objects ranging from a traditional client-server scenario to a fully peer-to-peer based approach. Automatic, dynamic resource management allows optimized use of available resources while minimizing the administrative complexity. Furthermore the environment supports a trust-based distinction of peers and enables a trust-based usage of resources.

1 Introduction

In the past few years, many research activities have been initiated to create new infrastructures for distributed applications. The primary goals are to overcome limitations of traditional client/server-structured systems, to increase flexibility, to reduce administrative cost, and to enable a better utilization of all available resources, possibly distributed world-wide.

The utilization of client-side resources is specifically addressed by the increasingly popular peer-to-peer systems. In such systems every peer has similar responsibilities and provides resources for the whole system. Frequent changes of participating nodes are supported by the protocols. However, severe degradation of the overall system performance or even a collapse of the system may happen if too many nodes participate only for very short periods of time. More severely, using all available peer resources means that all peers have the potential of attacking the system. In many systems, a single attacker may even counterfeit multiple identities in the system and thus control a significant part of the whole system. As a result, the advantage of using peer resources easily is being paid for with severe difficulties in controlling security and privacy concerns.

Infrastructures for grid computing aim at virtualizing a group of computers, servers, and storage as one large computing system. Resource management is a key issue in such systems, needed for an efficient and automated distribution of tasks on the grid. Such grid infrastructures are often deployed at enterprise level, but projects like SETI@home [1] have demonstrated the feasibility of more decentralized grids as well. Research on ubiquitous computing, autonomous computing, or spontaneous networks concentrates more on problems caused by the dynamicity in a network with a large number of devices, many of them mobile, and with huge differences in networking facilities, architecture and computational resources.

M. Jeckle, R. Kowalczyk, and P. Braun (Eds.): GSEM 2004, LNCS 3270, pp. 107–118, 2004.
© Springer-Verlag Berlin Heidelberg 2004

There is a tendency towards growing overlaps between all these different areas. Consequently future systems will demand a generic infrastructure that is able to fulfill the requirements of all such application types. Such next generation infrastructure will be confronted with the following challenges:

- First of all, faced with large systems with constantly growing complexity, it must keep the administrative complexity at an easily manageable level. A high degree of automation of management tasks is necessary, without loosing controllability of the system.
- Second, it has to provide mechanisms for a controlled usage of resources. On the one hand, it should be possible to make use of all peer resources available anywhere the distributed system. On the other hand, security and confidentiality concerns must be respected.
- Furthermore, it should allow easy and flexible adaptation to changing circumstances. One example of such dynamic reconfigurations is migration of service provision between rapidly changing participants (e.g., mobile devices). Similarly one can consider compensating reactions to failures, changes in available resources, or varying utilization of a service. The aspects of adaptation and resource control cannot be solved each on its own, but influence each other mutually.

In this paper, we present our *Environment for Distributed, Adaptive Services (EDAS).* This environment allows the usage of client-side resources in a controlled, secure fashion. It supports dynamic adaptation at run-time, provides a management infrastructure, and offers system-level support for scalability and fault tolerance. The environment is built upon our AspectIX middleware infrastructure, which directly supports QoS-based, dynamic reconfiguration of services. It supports flexible service models, including a fully centralized client/server structure, completely peer-to-peer based systems, and various configurations "in between" that allow a controlled use of peer resources. The overall goal is do provide a generic service architecture that allows to implement the service functionality once, and then, ideally, run this service with different service models and adapt it at run-time.

We support explicit management of available resources via a *home service.* Using this home service, domain administrators can provide resources for application services or service classes. For simplicity of administration, a set of nodes within one administrative domain is managed jointly. Furthermore, the home service is responsible for local resource monitoring (e.g., currently available memory, CPU resources, and network bandwidth) and notification about resource-specific events (e.g., addition or removal of resources, node shutdown).

The second key component is the *service environment.* Its task is to provide the environment in which services can be hosted. It manages the available execution locations, depending on resource offers by home services and trust specifications of the administrator of the service environment. It also reacts to notification from the home services, and suggests, for instance, that a service should be migrated to another available node as a reaction to a shutdown notification. The service environment is also able to consider different trust levels for the service. For example, the core of a service (e.g., all of its primary data replicas) might be located at highly trusted nodes only, whereas some caching or secondary read-only replicas might be placed on other available nodes as well.

This paper is structured as follows: The next section gives a short overview over the AspectIX middleware infrastructure. Section 3 presents the core architecture of EDAS. Section 4 illustrates the structure and properties of the environment with a sample service. Section 5 surveys related work. Section 6 summarizes our contribution and gives some concluding remarks on the status of our prototype implementation.

2 Basics Middleware Infrastructure

The EDAS environment is based on our AspectIX middleware [2]. At its core, it provides a CORBA-compliant ORB and, as such, supports heterogeneous distributed systems. There are extensions which allow direct interoperation with Java RMI or .NET applications. These extensions may be encapsulated in a transparent way for any client or service implementation. Our fragmented object model, which we will explain in the next subsection, provides a generic abstraction for services with arbitrary internal structure. Furthermore, AspectIX provides useful basic mechanisms for distributed adaptive services. A dynamic loading service (DLS) allows loading of service-specific code at the client side respecting local platform dependencies [3]. A generic architecture with state transfer mechanisms supports migration and replication of service fragments. These will be explained afterwards.

2.1 The Fragmented Object Model

In a traditional, RPC-based client-server structure, the complete functionality of an object resides on a single node. For transparent access to an object, a client instantiates a stub that handles remote invocations (Fig. 2.1 A). The stub code is usually generated automatically from an interface specification.

In the fragmented object model, the distinction between client stubs and the server object is no longer present. From an abstract point of view, a fragmented object is a unit with unique identity, interface, behavior, and state, like in classic object-oriented design. The implementation of these properties however is not bound to a specific location, but may be distributed arbitrarily on various *fragments* (Fig. 2.1 B). Any client that wants to access the fragmented object needs a local fragment, which provides an interface identical to that of a traditional stub. However the local fragment may be specific for exactly that object. Two objects with the same interface may lead to completely different local fragments. This internal structure allows a high degree of freedom on where the state and functionality is provided, and how the interaction between fragments is done. The internal distribution and interaction is not only transparent on the outer interface of the distributed object, but may even change dynamically at runtime. This allows the fragmented object model to adapt to changing environment conditions or quality of service requirements.

In the context of EDAS a decentralized, adaptive service is modeled as fragmented object. This offers the possibility to change the service model on demand from traditional client-server to a peer-to-peer based approach and all kind of intermediate stages by migrating and exchanging fragments.

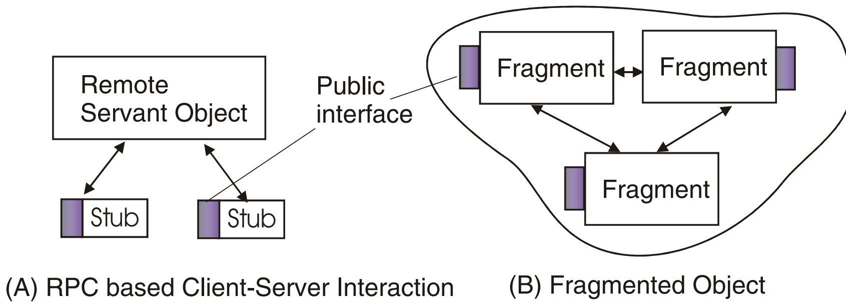

Fig. 2.1. RPC-based Client-Server Interaction vs. Fragmented Object

```
interface Checkpointable {
     void setState(in byte[] state);
     byte[] getState();
};
```

Fig. 2.2. `Checkpointable` Interface

2.2 AspectIX Services for Migration and Replication

The migration and replication of fragments or fragment implementations can be divided into three steps: First of all, an appropriate target system has to be found. After that, unless the necessary code is already available at the target system, it has to be loaded on demand. Finally, the corresponding fragment has to be instantiated and the state has to be transferred to the new location.

To solve these tasks the AspectIX middleware provides an extended version of the CORBA lifecycle service [4]. If a fragment has to be replicated or migrated a factory finder is consulted. Such a factory finder is responsible for locating factories within a defined scope. In the context of the AspectIX middleware, this factory finder consults special factory finder objects that reside on each node of the scope. These factory finders cooperate with the Dynamic Loading Service (DLS) to instantiate a factory for the needed fragment on demand, provided that the requirements of the fragment are met and the fragment code could be loaded and executed in the context of the target system. To achieve this, the DLS queries an implementation repository for all available implementations of the needed fragment. Then, the DLS checks the requirements of each implementation and selects the implementation that fits best to the local node. With the help of the factory a new fragment is instantiated.

After creation, the new fragment needs to be informed about its state. Our state transfer mechanism adheres to the according elements in the FT-CORBA standard [5]. That is, any service object that can be relocated or replicated has to implement a `Checkpointable` interface (see Fig. 2.2).

This interface is used for all kinds of state transfer, both for exchanging the local fragment implementation with a different one, and for remote state transfer for migration or replication. Special care has to be taken about concurrency between method invocations at the service and state transfer actions. Appropriate synchronization mechanisms are provided at the middleware level.

In the simplest case, the state is completely encoded into an octet sequence with get_state and decoded with set_state. Please note, however, that more sophisticated models are possible. get_state might, e.g., simply return a FTP address where to get the state, and set_state could use this address for the actual state transfer. For an exchange of the local implementation, get_state might simply encode the location on the local disk where the state resides, and set_state just feeds this information to the new local implementation.

3 Architecture of EDAS

Our environment for decentralized, adaptive services (EDAS) aims at providing a generic platform for services in a distributed system. Any EDAS-based services may be spread over a set of peers and combines available resources for service provision. Administrative complexity is minimized by automation of management tasks, like reconfiguration in reaction to failures, high load, or system policy modifications. Mechanisms for migration and replication of service components are available. The process of selecting execution locations considers trust metrics of peers that offer resources, to fulfill reliability requirements of a service.

The EDAS platform has three major components (Fig. 3.1): The *home service* is provided by every peer that actively supports decentralized, adaptive services. It basically manages resources of one or more peers belonging to the same administrative domain. The *service environment* is spread over a set of domains that support a group of decentralized, adaptive services run by one organization or community. Finally, the *decentralized, adaptive service* is dynamically distributed within the scope of an associated service environment.

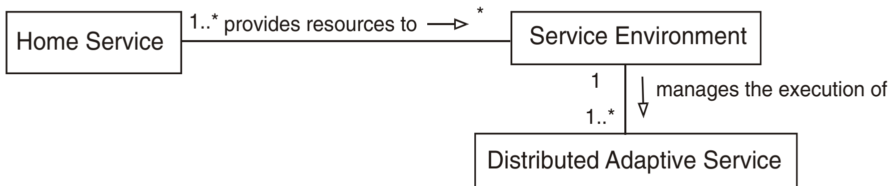

Fig. 3.1. Core Components

3.1 Home Service

The home service represents a mediator between the peers of an administrative domain and one or more service environments, each running a set of decentralized, adaptive services (Fig. 3.1). Fig. 3.2 shows three domains each running a home service which spans all peers of the respective domains. Every peer provides a set of resources. These resources are combined and monitored by the associated home service. Each domain has a manager who can use that home service to assign resources to service environments and to revoke them. Furthermore, the home service provides system information about each peer to the involved service environments and to the domain manager. This includes system load and all kinds of resource usage information but also the notification about important system events. For example, if a peer is shut down all affected service environments are notified and can migrate affected service components as needed.

3.2 Service Environment

A service environment represents a scope of distribution for one or more decentralized, adaptive services. Usually, a service environment is owned by one organization or community and managed by an individual called service manager. Such a manager can start, stop, and configure services through the interface of the service environment and decides which resources provided by home services are accepted (Fig 3.3). The main goal of the service environment is to support the seamless and easy distribution and management of distributed, adaptive services.

In most cases a service environment is spread over more than one administrative domain as shown in Fig. 3.2. One task of the service environment is to collect the system and resource information of the supporting home services. Another task is to manage the migration of services or service components, based on available resources, the needs of the services, and the policies provided by the service manager. The migration of service components can be necessary for various reasons, like peer shutdown, load balancing, or the growth or shrinkage of a service environment. For this purpose the service environment implements a generic factory for service components. If a new replica has to be created or a service component needs to be migrated, the service asks the service environment for a new service component instance. The service environment now has to determine which node provides sufficient resources and fulfills the system requirements of the service component. Further basic requirements have to be taken into account, like not to place replicas of the same component on the same node. To achieve this, a component specific factory has to be instantiated on all suitable hosts. The factory provides information about the resource requirements of the service component and checks in co-operation with the service environment if additional runtime requirements are fulfilled.

The expansion or shrinkage of a service environment depends on the offered resources and trustworthiness of the resource provider. Each domain manager has the possibility to offer resources to a service environment. The service manager can accept the offer and instruct the service environment to expand and use the offered resources. Furthermore, the service manager can assign a trust level to the administrative domain. This rating of the resource provider allows an explicit resource usage based on the trustworthiness. Up to now the rating is based on the knowledge of the service provider but we currently evaluate how and based on what information this could be done automatically. The shrinkage of a domain can be caused by an administrative domain revoking the usage permission or simply by decision of the service administrator. If a service component is migrated or a new replicate is instantiated, this is done in a trust-level conform way. A new component will always be placed on nodes with the same or higher trust level.

It is obvious that there could be situations where the available resources are not sufficient or severe problems occur like a network partition. In these cases, where the service environment cannot autonomously solve the problem, the service environment notifies the service administrator. The service administrator can now fix the problem manually (e.g., accept resource offers by other domains). The service environment also detects failures like a crashed node. In such a case the affected services are notified. Then it is up to the service to cope with the situation; for example, a service could request a new replicate.

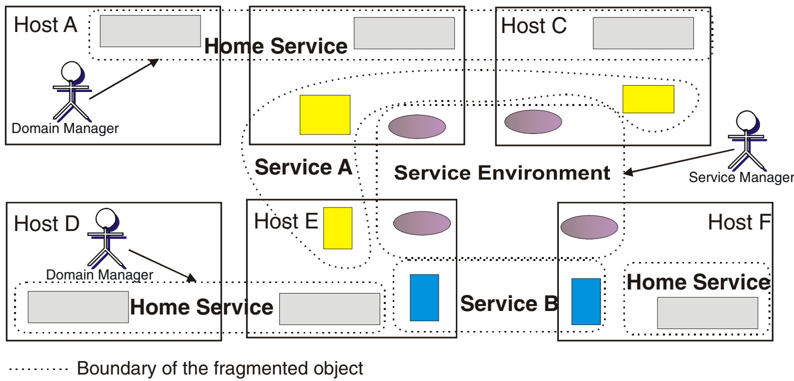

Fig. 3.2. EDAS Scenario

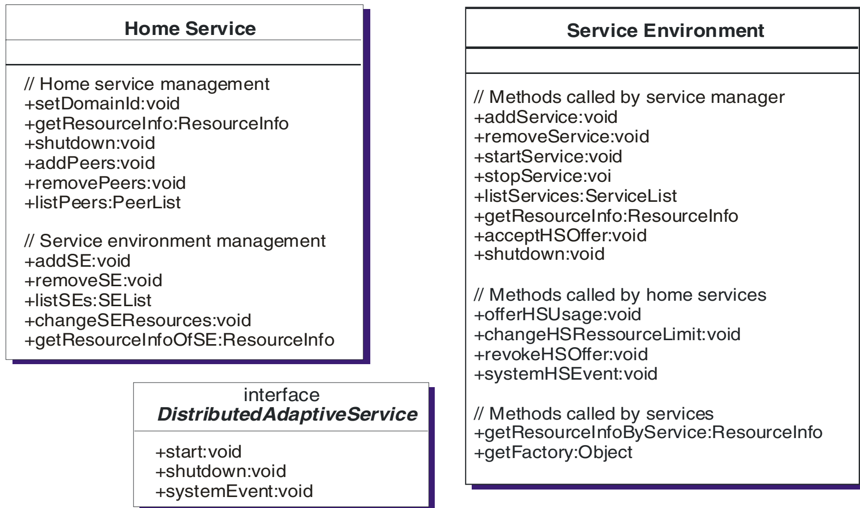

Fig. 3.3. Interfaces

3.3 Decentralized, Adaptive Services

In EDAS a decentralized, adaptive service normally matches a traditional service accessed by users like a web server, an instant messaging server or a source code repository. Such a service is represented by a fragmented object. This object expands or shrinks in the scope spanned by the associated service environment depending on the service demands and for fault-tolerance reasons. Usually every part of the object is mobile and is migrated if necessary. Each service has at least two interfaces: one for management tasks and another service specific for the end user. The management interface offers methods to start, stop, and configure service instances.

In the preceding section we already mentioned that each supporting domain has an assigned trust level. This level is used for a secure and fault tolerant distribution of a

service. As mentioned above a decentralized, adaptive service consists of different parts. In a typical service implementation each part of a service has equal security requirements. However, if parts of the fragmented object are replicated and the changes to the replication group and the replicated data are managed by a fault tolerant algorithm, the usage of only partial trustworthy peers is possible. The service has only to ensure that the maximum number of permitted faults is never exceeded.

Another possibility is the usage of something that we call *verifiable operations*. For example, a service provides a set of files. These files can be placed on a number of less trustworthy hosts. If a client requests a file from the service, it is transferred to the client and a signed checksum is provided by a part of the service residing on a fully trusted host. The client can now verify the integrity of the file with the checksum.

A third possibility is the distribution based on the self-interest of the resource provider. If the service can be structured in a way that parts of the service are only responsible for request issued by clients within a domain, then these parts should be hosted by peers of the associated domain whether they are trustworthy or not.

4 Sample Application

4.1 Overview

For illustrating how our environment for distributed, adaptive services works in practice, we use a CVS-like data repository as a sample application. Methods are available for adding new files or directories, committing changes, etc. We will show how this sample service can be deployed in different environment configurations. In the simplest case, one central server hosts the repository. Optionally, transparent service migration to a different host may be supported. For higher availability, the repository service could be replicated on a fixed set of nodes within one administrative domain. Automatic relocation or recreation of replicas in reaction to, e.g., failures may be considered as well. You might even want to distribute the repository over a less homogeneous set of individual nodes. For example, all developers using the repository might offer resources for the repository service, possibly distributed world-wide. Furthermore, available resources could be divided into different roles: Fully trusted nodes are allowed to host primary replicas, and other nodes are available for "mirrors", i.e., secondary read-only replicas which simply copy the state of the primary ones. In the ultimate case, you might want to use the data storage of a peer-to-peer network for hosting the repository.

4.2 Central Repository Implementation

For a central client-server implementation, only a very simple variant of the home service and service environment are needed: The home service is a simple, local fragment that allows the use of the local resources (disk, network) to a local service environment, which in turn hosts the repository service locally. No interaction with other nodes is necessary. The repository service itself is implemented in exactly the same way as one would do with traditional client/server middleware infrastructures.

If such basic service implementation additionally implements our standard interface for state transfer (see Section 2.2), migration is automatically supported in EDAS, without any modifications to the service implementation. Figure 4.2 illustrates the steps necessary for a service migration.

First of all (step 1), a second node has to offer resources for the service. This might be the case automatically, if Node 2 is managed via the same home service. Otherwise, the administrator of the home service of the second node has to offer its resources explicitly.

As a second step, the service environment needs to be expanded to the second node. This may happen automatically as soon as the service environment is notified of the resource offer, provided that the home service is sufficiently trusted. Otherwise, an explicit administrator action of the service environment can accept the new home service as a potential execution location for its services.

In a third step, the migration itself has to be triggered. This is either done explicitly by an administrator, or it is initiated automatically based on notifications ("*Node 1 will be shut down for maintenance*") or policies ("*Trigger migration if load on Node 1 exceed limits and another Node with significantly less load is available*"). The migration itself is done by accessing a factory on Node 2, which will use the DLS to load the repository service implementation on that node. After that, the service environment controls the state transfer using the mechanisms described in Section 2.2.

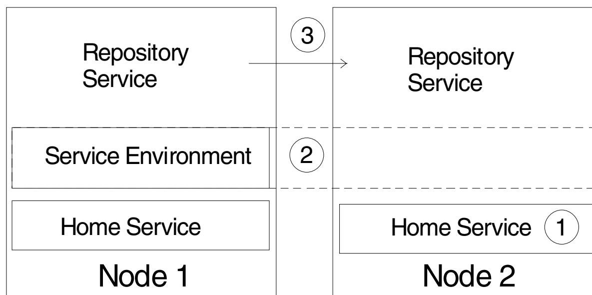

Fig. 4.1. Migration of a Central Repository Service

4.3 Replicated Service

It might be desirable to replicate the repository service for fault-tolerance or performance reasons. The AspectIX middleware provides basic services for passive or active replication strategies. These again relay on the `Checkpointable` interface described above. The EDAS environment is responsible for the management of the replication.

A sufficient number of nodes have to offer resources to the service environment of the repository service. The administrator of a service environment has to decide, which home services are to be considered when locating resources for service fragments. Furthermore, he may express policies for preferred locations and can define a desired degree of replication for a specific service.

Based in this information, the service environment is able to select execution locations for the service automatically. Using the factory and state transfer mechanisms already described before, the necessary number of replicas will be created. In contrast

to the migration example in 4.2, the DLS needs to load enhanced fragment code that uses some consistency protocol to ensure replica consistency. Such protocols are provided in the fault tolerance framework of the AspectIX middleware.

The service environment may perform dynamic reconfigurations – like migration or creation of replica – automatically. Such actions are initiated by notifications about shutdown or crash of a node, by overload of nodes, or when the service administrator adjusts the desired number of replicas.

An extension to such basic replication infrastructure is the introduction of a second trust level. In such a scenario, the service environment will always place the core parts of the repository service at fully trusted nodes. In addition, secondary replicas or caches might be created automatically in the system for load balancing reasons.

4.4 Peer-to-Peer-Based Repository

A general peer-to-peer based data store might be used for storing the repository. This implies all the security and confidentiality problems outlined in the introduction. To some extent, these may be overcome with adequate cryptographic mechanisms and massive replication. We are working on a Byzantine fault tolerant replication service that might be used in such situations. From an EDAS point of view, this is a rather simple variant: No state transfer is required as all data is available anywhere in the peer-to-peer network, and the resource usage is also fully under control of the P2P network, so that each node can act on its own (local home service and service environment on each node without interaction with other nodes).

5 Related Work

Grid infrastructures like the Globus-Toolkit [6] provide services and mechanisms for distributed heterogeneous environments to combine resources on demand to solve resource consuming, computation intensive tasks. Due to this orientation, grid infrastructures lack techniques for using resources of partially trustworthy peers and the usage of client-side resources in general. These systems also do not provide support for mobile objects or mobile agents. JavaSymphony [7] and Ibis [8] provide object mobility but are limited to the Java programming language and provide no distributed object model.

The CORBA middleware offers the interaction of objects across heterogeneous environments via generated stubs. The CORBA Life-Cycle Service [4] and the Fault Tolerant CORBA [5] extension provide a basic support for decentralized, adaptive services, but CORBA also lacks a distributed object model which enables the usage of client-side resources.

Previous mobile agent systems like Aglets [9] , Moa [10] or Mole [11] offer state and code migration of objects and special communication mechanisms for agent interaction. To support the implementation of distributed adaptive services, these systems lack a distributed object model and also flexible mechanisms for the utilization of client-side resources provided by partially trusted peers.

Peer-to-peer systems like CAN [12], Chord [13], or Pastry [14] construct an overlay topology and force each peer to collaborate if using the system. This has certain drawbacks if many peers participate only a short period of time or don't behave well.

The proposed concept of distributed, adaptive services offers each peer the possibility to provide resources or not. Moreover the participation of peers is controlled by a service manager. The peer-to-peer infrastructure JXTA [15] partly addresses this necessity through membership protected peer-groups. This offers the possibility to control participation, but also strongly limits the usage because only group members can use the services of a peer-group. Furthermore JXTA lacks any mechanisms for mobile objects.

6 Conclusions

We presented the architecture of EDAS, an environment for decentralized, adaptive services. This environment makes the following contributions. It provides a generic platform that allows using flexible service models ranging from a traditional client-server scenario to a fully peer-to-peer based approach. Based on the fragmented object model, it supports scalable services and mobility of service fragments. The administrative complexity of EDAS-based applications is minimized by its management infrastructure. It allows domain-based joint management of resource offers, supports an inter-domain resource selection taking into account assigned trust levels, and automates reconfigurations in reaction to events like failures, resource shortages, or explicit policy changes.

The EDAS prototype is based on our AspectIX middleware, which has been published under the LGPL licenses. It is available for download, together with further information, at `http://www.aspectix.org`. We plan to make the EDAS system software available as well. A first public release is scheduled for the end of 2004 after internal verification. Further steps will a better notion of trust and to provide mechanisms for trust aware distribution at the middleware level.

References

1. W. T. Sullivan, III, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, D. Anderson: “A new major SETI project based on Project Serendip data and 100,000 personal computers.” In *Proc. of the Fifth Intl. Conf. on Bioastronomy*,1997.
2. H. Reiser, F. J. Hauck, R. Kapitza, A. I. Schmied: “Integrating Fragmented Objects into a CORBA Environment.” In *Proceedings of the Net.Object Days*, Erfurt, 2003
3. R. Kapitza, F. J. Hauck: “DLS: a CORBA service for dynamic loading of code.” In *Proc. of the OTM Confederated International Conferences*, Sicily, Italy,2003
4. Object Management Group: *Life Cycle Service Specification.* Ver. 1.1, OMG Doc, formal/ 00-06-18, Framingham, MA, April 2002.
5. Object Management Group: *The Common Object Request Broker Architectur and Specification*. Ver. 2.6, OMG Doc. formal/01-12-35, Framingham, MA, Dec. 2001.
6. I. Foster, C. Kesselman, S. Tuecke: “The Anatomy of the Grid: Enabling Scalable Virtual Organizations.” In *International J. Supercomputer Applications*, 15(3), 2001
7. T. Fahringer: “JavaSymphony: A System for Development of Locality-Oriented Distributed and Parallel Java Applications.” In *Proceedings of the IEEE International Conference on Cluster Computing CLUSTER 2000*, Chemnitz, Germany, Dec. 2000.
8. R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, H. E. Bal: “Ibis: an efficient Java-based grid programming environment.” In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grand*, Washington, USA, 2002.

9. D. Lange, M. Oshima: “Programming and Deploying Java? Mobile Agents with Aglets.” Addison Wesley, 1999
10. D. Milojicic, W. LaForge, D. Chauhan: “Mobile Objects and Agents (MOA).” In *Proc. of USENIX COOTS ’98*, Santa Fe, 1998
11. M. Strasser, J. Baumann, F. Hohl: “Mole - A Java based Mobile Agent System.” In: *M. Mühlhäuser: (ed.), Special Issues in Object Oriented Programming*, dpunkt-Verlag 1997
12. S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker: “A Scalable Content-Addressable Network.” In *Proceedings of the ACM SIGCOMM ’01 Conference*, San Diego, August 2001
13. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan: “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.” In *Proceedings of the ACM SIGCOMM ’01 Conference*, San Diego, August 2001
14. A. Rowstron, P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems." In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November, 2001.
15. L. Gong. “JXTA: A network programming environment.” In *IEEE Internet Computing*, v. 5, 2001