A Framework for Dynamically Adaptive Applications in a Self-organized Mobile Network Environment*

Arun Mukhija and Martin Glinz
Institut für Informatik
University of Zurich, CH-8057, Switzerland
{*mukhija | glinz*}*@ifi.unizh.ch*

Abstract

Self-organized mobile networks present a challenging environment for the execution of software applications, due to their dynamic topologies and consistently changing resource conditions. In view of the above, a desirable property for software applications to be run over these networks is their ability to dynamically adapt to changing execution environments. The Contract-based Adaptive Software Architecture (CASA) provides a framework for the development of adaptive applications that are able to adapt their functionality and/or performance dynamically in response to runtime changes in their execution environments. The approach of the CASA framework is to decouple application code from any assumptions about resource availability, while enabling the application to execute under varying resource conditions. The CASA framework relies on specifying adaptation behavior of applications in application contracts, which enables the dynamic adaptation to be carried out in an application-transparent manner.

1. Introduction

Self-organized mobile networks (also known as mobile ad-hoc networks) offer a very flexible way of operation, wherein mobile nodes are free to join or leave a network community, or travel within the network, without any prior warning. This results in varying resource contentions dynamically among applications. Consequently, the task of steady execution of applications over these networks becomes even more challenging – as they have to deal not just with low availability of resources, but also with unreliable availability of resources. In such a situation, the applications should be able to adapt themselves dynamically in response to frequent unpredictable changes in their execution environments.

The problem of unreliable availability of resources is also experienced in conventional mobile networks, but since conventional networks rely on a fixed infrastructure, the problem is not as severe and the solutions not as inadequate as for self-organized mobile networks. Consider an example scenario where the communication between two mobile nodes of a conventional mobile network (such as a GSM network) is taking place through a fixed infrastructure of routers. If, due to some reasons, the bandwidth between one of the communicating nodes and its current base station drops significantly (for example, if the node moves away from its current base station), the node concerned may switch to another base station with which it has a better bandwidth connection (like in a handover). The base stations are usually strategically located throughout the communication area, and their capacities and numbers carefully planned, in order to provide optimal service to mobile nodes. Compare this to an equivalent scenario in a self-organized mobile network, where two mobile nodes are communicating with each other through a route that is made up of a chain of other mobile nodes acting as routers for these two nodes. If the bandwidth between one of the communicating nodes and its predecessor drops (or, for that matter, if the bandwidth on any of the links comprising the route drops), there is rarely an option to switch to some other route in order to avoid the low bandwidth link. This is because the process of route establishment and maintenance is time-consuming and resource-intensive in a self-organized mobile network.

Similarly, techniques for traffic load balancing and congestion control, which are widely available for conventional networks, rarely apply to self-organized mobile networks. So, ultimately, for conventional mobile networks, there is generally a tradeoff between investing in the infrastructure versus getting the quality of service for applications, whereas for self-organized mobile networks, such a trade-

* The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

Proceedings of the 4th International Workshop on Distributed Auto-adaptive and Reconfigurable Systems at the 24th International Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan, March 2004, pp. 368-374.

off does not exist, since there is no fixed infrastructure to invest in.

Nevertheless, the high cost of investing in infrastructure and certain physical constraints have motivated research towards runtime adaptation of high resource-consuming applications, such as real-time multimedia systems. Most of the suggested approaches, as discussed in the related work section, assume a QoS paradigm which is transparent to applications. Such approaches attempt to offer middleware-based solutions, such as modifying the quality of transmitted data or the degree of data compression, adjustments to resource schedules, modifications of replication factors etc., in order to meet QoS requirements of applications. However, these approaches fail in environments where we have a very high variation in the availability of resources. This is frequently the case in self-organized mobile network environments. In such situations, the applications themselves must be reconfigured dynamically in response to changing resource availability.

The Contract-based Adaptive Software Architecture (CASA) framework aims at facilitating this dynamic reconfiguration of applications:

- Every application offers alternative component configurations, differing in their resource requirements and, probably, also in their functionality and/or performance.
- Every application defines its own adaptation policy in an application-independent format, the so-called application contract.
- CASA provides a runtime environment that dynamically adapts the application by reconfiguring its constituent components in response to changes in the execution environment, and in accordance with the adaptation policy specified in the application contract.
- The adaptation mechanism is decoupled from the application itself, thereby enabling the dynamic adaptation to be carried out in an application-transparent manner.

Thus, an application developed according to the CASA framework does not assume any particular resource availability conditions, while at the same time the application offers alternative functionalities and performance levels to deal with different resource conditions that may arise during the execution life of the application.

In this paper, we give a brief overview of the CASA framework, and then concentrate on details of service negotiations among applications, dynamic adaptation and contract specifications. In a previous paper [5], we described the constituent entities of the CASA framework and hence we will not discuss these in detail here.

The rest of the paper is organized as follows. Section 2 gives an overview of the CASA framework, and then discusses details of service negotiations among applications and dynamic adaptation. Section 3 describes a hypothetical example that we will follow to explain concepts in the remaining sections. Section 4 provides details of contract specifications. Section 5 gives details of the service agreement protocol. Section 6 gives an overview of related work. Finally, Section 7 presents some concluding remarks and discusses the future direction of our work.

2. The CASA framework

A self-organized mobile network consists of autonomous mobile nodes with dynamically changing topologies. The mobile nodes are usually owned by independent users, and each node may host any number of applications depending on their utility and user's discretion. An "application", in this paper, refers to a set of components residing on a single network node that are working together for a specific task. Remote applications residing on different autonomous nodes of a network may collaborate with each other at runtime for a mission, thereby forming a "distributed software system".

The overall CASA framework is as shown in Figure 1. In this section, we first briefly describe the constituent entities of the CASA framework, and then illustrate the overall working through a typical interaction between remote applications. For a more detailed description of individual entities of the CASA framework, please refer to [5].

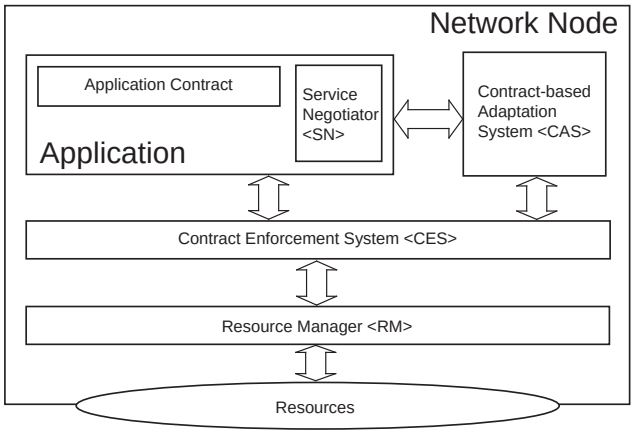

Figure 1. The CASA framework.

An application is composed of components. A "component configuration" (referred to as *config* in this paper) is a set of software components qualified to do the task required of the application. Different *config*s, capable of doing the same task, but with distinct resource requirements and, probably, different functionality and/or performance levels, are referred to as "alternative *config*s". Any two alternative

*config*s may vary in just a few (minimum one) components, while many other components remain the same across both *config*s. CASA requires each adaptive application to offer multiple alternative *config*s. At runtime, depending on the current availability of resources, the appropriate *config* is selected and activated by the CASA runtime system. Thus, the application adaptation, in response to changes in the computing environment, is achieved by executing the most appropriate *config* of the application for the changed environment. The adaptation behavior of an application is described in the application contract, which is expressed in the so-called Contract Specification Language (CSL).

The application contract is divided into zones, which are distinguished by the level of service provided by, or expected by, an application in a given zone (details of contract specifications are given in Section 4). Each zone lists details of alternative *config*s of the application – with the same level of service, but differing in resource requirements. Details of a *config* include the components constituting the *config* and its resource requirements. The zones, as well as the alternative *config*s within a zone, are ordered with respect to their user-perceived preference. The ordering can be changed dynamically to reflect a change in the user's preference.

Each application contains a Service Negotiator (SN) component that is responsible for negotiating the level of service to be provided to, or expected from, its peer applications, using the Service Agreement Protocol (SAP). The SN contains a mapping module that maps the proposed service parameters to the appropriate zone in the application contract. The mapped zone is, obviously, the one that contains alternative *config*s, which are able to satisfy the above service parameters.

Each network node runs an instance of the CASA runtime system, which consists of the Contract-based Adaptation System (CAS), the Contract Enforcement System (CES) and the Resource Manager (RM). As the name suggests, the RM is responsible for managing resources which involves monitoring the availability of resources, and reserving resources for various applications. Resources include the local resources (such as memory, processing capacity and battery power) and the distributed resources (such as communication bandwidth). The RM keeps the CES updated about the current resource status, and makes resource reservations according to the resource allocation decisions of the CES. For the reservation of distributed resources, the RMs of different nodes coordinate with each other using a resource coordination protocol. The CES is a central entity responsible for allocating resources to all contending applications. The CES has global knowledge of current resource availability, applications requirements, priorities and adaptation possibilities. Using this knowledge, the CES makes fair and justified resource allocation decisions, and strives to maximize the overall utilization of system resources. In the event of a mismatch between resources requested by an application and those that can be allocated to it, the current *config* of an application may need to be replaced by a more appropriate *config* for the current resource conditions, according to the adaptation rules contained in the application contract. The CAS carries out dynamic replacement of *config*s in a seamless manner, as and when instructed by the CES. The CAS takes care of state transfer from the old *config* to the new *config*, and maintains the integrity of the existing transactions.

2.1. A typical interaction session between CASA applications

Figure 2 illustrates a typical interaction session between two remote applications: App-A (residing on node A) and App-B (residing on node B). Using this example, we describe how service negotiations and dynamic adaptation work in CASA.

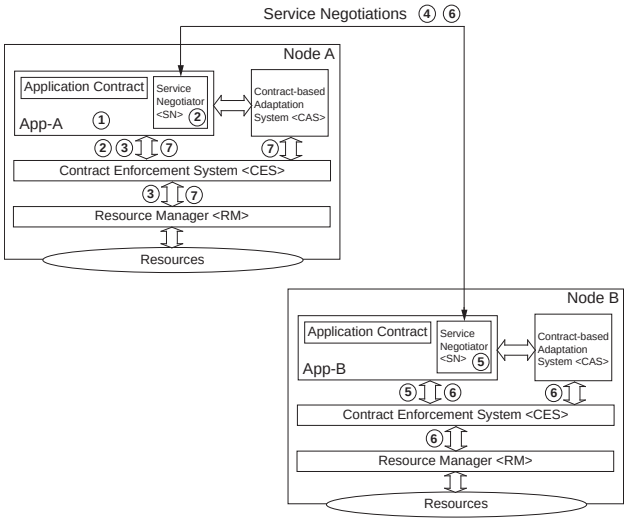

Figure 2. Interaction between applications residing on remote nodes.

Service negotiations: It is assumed that at least one of the applications (say, App-A here) has discovered the other application (here, App-B) through some service discovery system or any other means, and has decided to interact with it. Figure 3 depicts a simplified message sequence diagram for service negotiations between App-A and App-B. The details of service negotiations are explained in the sequence of steps that follows. Circled numbers in Figure 2 correspond to the sequence numbers in the following description.

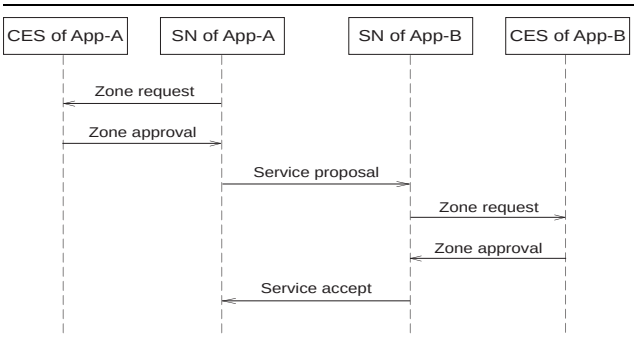

Figure 3. Service negotiations.

1. App-A decides on the service parameters for its interaction with App-B.
2. The mapping module of App-A maps the above service parameters to the corresponding zone in the application contract of App-A. The SN of App-A submits a request for this zone to the CES of node A.
3. The CES decides whether to accept or reject the request of the SN – taking into account the relative priority of App-A and the current resource availability.

   If the requested zone number can be accepted – which means there are enough resources to accommodate at least one of the *config*s in the requested zone – the CES confirms this to the SN. Otherwise, the CES searches for an appropriate *config* that can be accommodated given the existing resource constraints in other zones of the application contract (scanning the application contract from top to bottom, as this is the ordering of *config*s w.r.t. their user-perceived preference), and communicates the new zone number to the SN. If the CES is unable to find any *config* in the application contract that could execute in the current resource conditions, it communicates to the SN that the application cannot execute in the current situation.
4. If the zone number was successfully agreed between the SN and the CES, then the SN sends a "service proposal", with service parameters corresponding to the agreed zone number, to its counterpart in App-B (syntax of "service proposal" is described in Section 5).
5. The mapping module of App-B maps the service parameters contained in the "service proposal" to the corresponding zone in the application contract of App-B. The SN of App-B then submits a request for this zone to the CES of node B.
6. This step is similar to Step 3. The CES of node B decides whether to accept or reject the requested zone number – taking into account the relative priority of App-B and the current resource availability.

   If there are enough resources to accommodate at least one of the *config*s in the requested zone, the CES initializes App-B. That is, the CES reserves resources for the chosen *config*, instructs the corresponding CAS to activate the chosen *config*, and communicates the acceptance of request to the SN of App-B. The SN of App-B then sends a "service accept" message to the SN of App-A (syntax of "service accept" is described in Section 5).

   Otherwise, if the requested zone number cannot be accepted, then the CES looks for an appropriate *config* that has resource requirements compatible with the current resource conditions in other zones of the application contract, and communicates the zone number of this *config* to the SN. The SN passes on the service parameters corresponding to this zone number as a new "service proposal" to the SN of App-A, and the cycle repeats until one of the applications sends a "service accept" or a "service reject" message.

   If the CES is unable to find any *config* in the application contract that could execute in the current resource conditions, then a "service reject" message is sent back to App-A, implying that the interaction between the two applications cannot take place currently (syntax of "service reject" is described in Section 5).
7. Once a "service accept" message is received by an application, the corresponding CES is directed to initialize the application.

The above steps constitute service negotiations and initializations of the two applications, after which the interaction between them can begin immediately. Next we discuss the dynamic adaptation of the applications.

Dynamic adaptation: During the runtime of the above interaction, there are two frequently occurring cases: *Scarcity* of resources and *Abundance* of resources. *Scarcity* of resources implies that the total demand for resources exceeds the total resource availability at one of the nodes. It may occur because of resource failures, activation of new applications, increase in demand for certain resources by existing applications etc. It requires the affected applications to adapt dynamically, and some applications may need to sacrifice their functionality or performance or both. *Abundance* of resources implies that the total availability of resources exceeds the total demand at one of the nodes. It may occur due to reasons such as the recovery of certain resources that failed earlier, freeing up of some resources by some applications etc., and it may enable currently running applications to improve their functionality / performance. We discuss these two cases below.

Case 1: Scarcity of resources (say, at node A): The CES of node A would need to take away some of the resources allo-

cated to some of the applications running on this node, according to their relative priorities. Let us assume that one of the applications that needs to give away some of its resources be App-A, with the result that the currently executing *config* of App-A cannot execute any longer.

The CES of node A looks for an alternative *config* that can be accommodated in the current resource conditions, within the same zone as that of the current *config*. If the CES finds one, it simply instructs the corresponding CAS to replace the current *config* with the new *config*. In this case there is no need to seek approval from peer applications before executing the change, as the alternative *config*s within the same zone offer the same level of service.

Otherwise, if no matching *config* is found in the current zone, then the CES searches for an appropriate *config* in other zones of the application contract. If the CES finds one that can be accommodated given the current resource conditions, then it communicates the zone number of this *config* to the SN of App-A. Service renegotiations between App-A and App-B then take place accordingly.

If the CES is unable to find any matching *config* that could work in the existing resource conditions, then it implies that App-A cannot execute any longer. The CES communicates this to the SN of App-A, which in turn communicates this to the SN of App-B by sending a "service reject" message.

Case 2: Abundance of resources: If the CES of the node concerned had to deny the zone numbers requested by some applications during initial service negotiations, or it had to change the active zone numbers of some applications at runtime (as in Case 1 above), then the CES keeps record of all such applications along with their last requested/active zone numbers. And whenever spare resources are available, the CES tries to allocate the extra resources to these applications, in the order of their relative priorities, such that their original zone requests may be satisfied. Service renegotiations before switching to the new zone take place accordingly.

Several smaller steps and finer details, such as provisional reservation of resources during service negotiations, seeking user's approval, clean-up operations etc. are not mentioned in the above description.

3. A hypothetical example

Before discussing further details of CASA, let us consider a hypothetical distributed software system called the Emergency Coordination System (ECS). The ECS coordinates rescue operations in an earthquake affected area, and is running on a self-organized mobile network since the fixed infrastructure might be destroyed anyway. The area affected by the earthquake is divided into sectors which are identified by their unique sector numbers. Each sector has a camera, recording the damage in the sector. The ECS consists of two applications – *Monitoring* and *Support*. The cameras are connected to *Monitoring*, running on the node M. *Monitoring* sends real-time data about the damage to *Support*, running on the node S. *Support* is responsible for taking appropriate action, such as sending human rescue teams, debris removal teams, medical personnel, fire tenders etc., depending on the type and extent of damage in each sector. *Monitoring* can send data in the following alternative formats, depending on the resource availability:
– Video stream (high quality / low quality)
– Images at frequent intervals (high quality / low quality)
– Textual description of the damage (detailed / brief)

The above example is relevant to our discussions because, due to the nature of operation, the resource fluctuations can be very high, and the system is able to run at different functionality / performance levels depending on the resource availability.

4. Contract specifications

Application contracts are specified using CSL (Contract Specification Language). CSL is an XML-based specification language, developed as part of the CASA framework. The reason for using CSL is so that application contracts are specified in a standard and uniform manner, i.e. independent of the application implementation language and platform. This uniformity helps to achieve transparency in dynamic adaptability of applications. Being XML-based, CSL can express application contracts in a structured and easily extensible format. The syntax of application contracts is as follows.

The `app-contract` tag is a container tag with one required attribute, `name`, which contains the name of the corresponding application as its value. It contains at least one `zone` tag, which is also a container tag. The `zone` tag has just one required attribute, `id`, containing the identification number of that zone. The `zone` tag contains any number of `config` tags (minimum one), specifying alternative *config*s for that zone. Each `config` tag contains two required attributes, namely `id` and `comps`, containing the identification number of the *config* and the names of components constituting that *config*, respectively. In addition, the `config` tag has all the relevant resource attributes, such as `mem`, `cpu`, `bwh` and `pow`, representing the resource requirements of the *config* in terms of memory, processing cycles, communication bandwidth and battery power, respectively.

Only the adaptable components, i.e. those components that differ from one *config* to another, are specified as a part of a *config*, and not the ones that remain the same across all *config*s and provide some common core functionality. Resource requirements corresponding to a *config* can easily be computed using various analytical and probing techniques

available for this purpose. The requirements may also be specified at a higher level of abstraction, such as in terms of throughput and packet size, instead of directly specifying them in terms of resources such as bandwidth etc.

In Figure 4, we present a part of the application contract for *Monitoring* in our hypothetical ECS example.

```
<app-contract name="Monitoring">
     <zone id="1">
         <config id="1" comps="sender(VIDEO-MPEG4-HighRate)"
                     mem="m1" cpu="c1" bwh="b1" pow="p1" />
         <config id="2" comps="sender(VIDEO-MPEG4-LowRate)"
                     mem="m2" cpu="c2" bwh="b2" pow="p2" />
     </zone>
     <zone id="2">
         <config id="1" comps="sender(VIDEO-MPEG1-HighRate)"
                     mem="m3" cpu="c3" bwh="b3" pow="p3" />
         <config id="2" comps="sender(VIDEO-MPEG1-LowRate)"
                     mem="m4" cpu="c4" bwh="b4" pow="p4" />
     </zone>
     <zone id="3">
         <config id="1" comps="sender(IMAGE-COLOR-HighRes)"
                     mem="m5" cpu="c5" bwh="b5" pow="p5" />
         <config id="2" comps="sender(IMAGE-COLOR-LowRes)"
                     mem="m6" cpu="c6" bwh="b6" pow="p6" />
     </zone>
     .
     .
     .
</app-contract>
```

Figure 4. Application contract.

5. The Service Agreement Protocol (SAP)

Service negotiations among applications constituting a distributed software system take place using the Service Agreement Protocol (SAP). Complex service negotiations involving multiple applications are broken into multiple simple service negotiations between just two applications. This helps in accommodating the scalability issue when dealing with complex distributed software systems involving multiple nodes and applications.

The SAP consists of three kinds of messages: "service proposal", "service accept" and "service reject", which are also specified using CSL. The semantics of these messages are as discussed in Section 2.1. Below we describe the syntax of these messages.

Service proposal: The service proposal has only one element called `service-proposal`. It has four required attributes namely `id`, `name`, `user` and `provider`, containing the unique identification number for this service session, name of the service, details of the service user and service provider as their values. In addition, it has several optional attributes specifying various service parameters for the corresponding service. The service parameters are application-domain-specific and not standard, which means that the semantics of these parameters need only to be understood by the applications involved.

Below is an example of a service proposal, as sent by *Support* to *Monitoring* in our hypothetical ECS example:

```
<service-proposal id=''101'' name=''damage-info''
user=''Support(S)'' provider=''Monitoring(M)''
sectors=''1,2,3,4,5,6,7,8'' datatype=''video''
quality=''high'' />
```

In response to the above service proposal, *Monitoring* may send an alternative service proposal to *Support*, as below:

```
<service-proposal id=''101'' name=''damage-info''
user=''Support(S)'' provider=''Monitoring(M)''
sectors=''1,2,4,5,8'' datatype=''image''
quality=''low'' />
```

For certain kinds of services, the cost of using a service may be an important service parameter to be specified as a part of the service proposal. Other common service parameters include precision of results, timeliness etc.

Service accept: The syntax of service accept is similar to that of service proposal. It is sent in response to a service proposal and contains the same service parameters and values as the corresponding service proposal. For our ECS example, if *Support* accepts the last service proposal sent by *Monitoring*, it may send a service accept message like the one below:

```
<service-accept id=''101'' name=''damage-info''
user=''Support(S)'' provider=''Monitoring(M)''
sectors=''1,2,4,5,8'' datatype=''image''
quality=''low'' />
```

Service reject: The syntax of service reject is also similar to that of service proposal, except that it does not contain any service parameters. For our ECS example, at any time either *Monitoring* or *Support* may send a service reject message like the one below:

```
<service-reject id=''101'' name=''damage-info''
user=''Support(S)'' provider=''Monitoring(M)'' />
```

6. Related work

Several approaches have been suggested for providing quality of service guarantees for distributed software systems. Most such approaches have focussed on providing a middleware platform to take care of quality of service concerns of applications. TAO [9], which is an implementation of Real-Time CORBA [8], attempts to optimize real-time method invocations within the ORB, but it does not provide any means for high level adaptations of applications. The approach taken by Quality Objects (QuO) [11] extends CORBA to provide quality of service for CORBA object invocations. However, it does not provide any mechanisms for dynamically reconfiguring the application itself.

There are a few other approaches that attempt to ensure quality of service at the middleware level, such as the Odyssey architecture [7], Reflective Middleware [2] and REal-time Software Adaptation System (RESAS) [1]. Such approaches mostly attempt to carry out adaptations at the middleware level, like adapting the data format, revising resource schedules, modifying replication factors, changing timeout periods etc., in response to external changes. However, they do not provide any means for modifying the application itself if the above mechanisms fail to work.

Adaptive Resource Allocation (ARA) [10] provides models and mechanisms to enable adaptive resource allocation for applications with dynamically changing resource needs. Similarly, approaches like Globus Architecture for Reservation and Allocation (GARA) [4] and the Darwin project [3] strive to provide efficient resource management techniques in order to satisfy quality of service requirements of the applications.

Another approach called $2K^Q$ system [6] talks about functional adaptation in response to QoS changes, and it shares the same goals as our CASA framework. However, this too does not provide any mechanisms for dynamically changing the internal logic of a running application. Moreover, it provides a centralized control over adaptation policy for the complete distributed system, whereas in CASA the applications at every discrete node can adapt individually, according to their own adaptation policies. Since self-organized mobile networks consist of autonomous nodes that form ad-hoc networks, independence in deciding an application's own adaptation policy is significant.

7. Concluding discussion and future work

CASA provides a framework for dynamic adaptation of applications in response to changes in their execution environments. The runtime adaptation is carried out by the application-independent entities of the CASA framework, in accordance with the adaptation policies specified in the respective application contracts. CSL, which is an XML-based specification language used to express application contracts, further enables the dynamic adaptation of applications to be carried out in a transparent manner.

This facility does not come for free, of course. In order to take advantage of the framework, the application developer needs to provide alternative component configurations for the application, and generate an application contract specifying the adaptation policy of the application. Considering the benefits achieved by transparent runtime adaptation, for applications running in highly dynamic distributed environments, it is probably a small price to pay. Moreover, we envisage the development of (automated or semi-automated) tools for analyzing alternative component configurations and generating application contracts, which will help the application developer with his job. The effort spent in developing alternative component configurations will, obviously, be amortized in proportion to the amount of reuse of components comprising these configurations.

The CASA framework was originally developed with self-organized mobile networks as the target environment, although it can be used for all environments that are faced with the challenge of unreliable resource availability.

We have implemented a prototype to demonstrate the feasibility of the CASA framework, and the results have been encouraging. In particular, we have been able to carry out dynamic adaptation of applications successfully in response to (artificially simulated) changes in resource availability. Our next step is to evaluate the performance of the CASA framework in terms of the overheads it causes.

References

[1] T.E. Bihari and K. Schwan, "Dynamic Adaptation of Real-Time Software", *ACM Transactions on Computer Systems*, 9(2), 1991.

[2] L. Capra, W. Emmerich and C. Mascolo, "Reflective Middleware Solutions for Context-Aware Applications", *Proc. of 3rd Intl. Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, 2001.

[3] P. Chandra, A. Fisher, C. Kosak, T.S. Eugene Ng, P. Steenkiste, E. Takahashi and H. Zhang, "Darwin: Customizable Resource Management for Value-Added Network Services", *Proc. of 6th Intl. Conference on Network Protocols*, 1998.

[4] I. Foster, A. Roy and V. Sander, "A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation", *Proc. of 8th Intl. Workshop on Quality of Service*, 2000.

[5] A. Mukhija and M. Glinz, "CASA – A Contract-based Adaptive Software Architecture Framework", *Proc. of 3rd IEEE Workshop on Applications and Services in Wireless Networks*, 2003.

[6] K. Nahrstedt, D. Wichadakul and D. Xu, "Distributed QoS Compilation and Runtime Instantiation", *Proc. of 8th Intl. Workshop on Quality of Service*, 2000.

[7] B.D. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn and K.R. Walker, "Agile Application-Aware Adaptation for Mobility", *Proc. of 16th ACM Symposium on Operating System Principles*, 1997.

[8] Object Management Group, *Real-Time CORBA Specification*, 2002. `http://www.omg.org/`

[9] I. Pyarali, D.C. Schmidt and R.K. Cytron, "Techniques for Enhancing Real-Time CORBA Quality of Service", *Proceedings of the IEEE (Special Issue on Real-Time Systems)*, 91(7), 2003.

[10] D.I. Rosu, K. Schwan, S. Yalamanchili and R. Jha, "On Adaptive Resource Allocation for Complex Real-Time Applications", *Proc. of 18th IEEE Real-Time Systems Symposium*, 1997.

[11] J.A. Zinky, D.E. Bakken and R.E. Schantz, "Architectural Support for Quality of Service for CORBA Objects", *Theory and Practice of Object Systems*, 3(1), 1997.