# The CASA Approach to Autonomic Applications

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

***Abstract*— In today's world of highly dynamic computing environments, *autonomic* applications are the need of the hour. By an autonomic application, we mean an application that is able to adapt to changes in its execution environment *dynamically* and *transparently*. CASA (Contract-based Adaptive Software Architecture) provides a framework for enabling the development and operation of autonomic applications. CASA helps in significantly reducing the complexity involved in developing autonomic applications by separating the adaptation concerns of an application from its business concerns. CASA further provides a runtime system for dealing with the adaptation concerns. In order to meet the adaptation needs of a broad and diverse set of applications, CASA supports adaptation at various levels of an application – from lower-level services to application code. In CASA, the adaptation policy of every application is defined in a so-called application contract, which is external to the application and is specified using an XML-based language, thereby facilitating changes in the adaptation policy at runtime.**

## I. Introduction

Mobile wireless computing environments provide immense *flexibility* and *value* to users. With the growth of these environments, many new and innovative applications are being conceived and developed for such environments. However, in order to cultivate the benefits offered by these environments, the applications need to successfully face the challenge of frequent and usually unpredictable changes in the execution environment, which are invariably associated with such dynamic environments.

A change in the execution environment can present an *opportunity* or a *threat* for a running application. Let us look at the threat first. A threat is when a change in the execution environment results in a loss of certain resources that are required by an application. This may force the application to run at a degraded performance or functionality. On the other hand, a change in the execution environment in the form of a change in contextual information (such as user's location or identity of nearby objects etc.) may present an opportunity for an application to provide a more relevant context-dependent service. In either case, the application should be able to adapt to changes in its execution environment, preferably *dynamically* (i.e. at runtime, without requiring to stop and restart the application) and *transparently* (i.e. without requiring user's intervention).

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

An application that is able to adapt to changes in its execution environment dynamically and transparently is called an *autonomic* application.

Consider a hypothetical *Collaborative Working* scenario, where a number of participants are collaborating on a common mission. The participants may be geographically distributed, and some of them may even be mobile at any given time. For the collaboration to work, each participant runs a client module of the collaborative working application, which includes a video display showing other participants, a shared drawing space and a discussion board. Some of the resources available to a participant, such as communication bandwidth and battery power, are likely to vary over time because of the mobility and other constraints. Similarly, the contextual information related to a participant (in a meeting, at home etc.) is likely to vary over time. The runtime changes in resources and contextual information demand an appropriate and non-disruptive adaptation of the collaborative working application. For example, in response to a small drop in the bandwidth available to a participant, the application may reduce the quality of the video display accordingly. Whereas for a large drop in the bandwidth, the application may remove any video content altogether. A change in the contextual information may also influence the application behavior, e.g. only important updates may be sent while the participant is in a meeting.

The concept of dynamically and transparently adaptable applications is not entirely new. Several approaches have been proposed that try to adapt the lower-level services used by an application at the middleware level, and thereby change the application behavior. Some other approaches try to adapt an application by dynamically weaving and unweaving aspects into / from an application. The approaches for runtime software evolution, involving a dynamic recomposition of application components, also are closely related to software adaptation.

However, none of these adaptation techniques are individually sufficient to meet the adaptation needs of different kinds of applications. In fact, these techniques can complement each other in order to meet the adaptation needs of a broad and diverse set of applications executing in dynamic environments.

For example, if there is a small drop in the available bandwidth, an adaptive middleware may use a lower-level compression service to compress the data before transmission. But for a large drop, the compression alone may not be sufficient and a change in application code is required to reduce the throughput of the application. Similarly, dynamic changes in aspects and components are complementary to each other, depending on whether the crosscutting or the core functionality of an application needs to be changed in response to a particular change in the execution environment.

The adaptation techniques can be classified according to the level where the adaptation takes place, as follows:

- dynamic change in lower-level services,
- dynamic weaving and unweaving of aspects,
- dynamic recomposition of application components,
- dynamic change in application attributes.

Ideally, an autonomic application should be able to use any combination of the above adaptation techniques, depending on its adaptation needs.

Another challenge facing autonomic applications is that the development process of these applications has largely been *ad-hoc*. In particular, the adaptation concerns of an application are intertwined with its business concerns. This increases the complexity involved in developing autonomic applications. We believe that a framework-based approach for developing autonomic applications can help in coping with this challenge.

CASA (Contract-based Adaptive Software Architecture) provides a framework for enabling the development and operation of autonomic applications. The key features of CASA are:

- Separation of the adaptation concerns of an application from its business concerns,
- A runtime system for dealing with the adaptation concerns,
- Support for adaptation at various levels of an application, as identified above,
- A contract-based adaptation policy, facilitating changes in the adaptation policy at runtime.

A preliminary version of the CASA framework has been presented in [10]. In this paper, we present the detailed design, working and evaluation of the framework, including several enhancements such as support for more adaptation mechanisms.

The rest of the paper is organized as follows. Section II gives an overview of CASA. Section III discusses the design of the CASA framework. Section IV describes the application contract specification. Section V discusses the overall working of CASA. Section VI presents some details of a CASA prototype implementation and its performance evaluation. Section VII gives an overview of related work. Finally, Section VIII concludes the paper.

## II. Overview of CASA

Following the principle of *separation of concerns*, CASA separates the adaptation concerns of an application from its business concerns. Separating the adaptation concerns has an obvious advantage of reducing the complexity involved in developing autonomic applications. Additionally, it facilitates reuse and sharing of the adaptation mechanisms among applications. This, in turn, enables CASA to provide a runtime system for dealing with the adaptation concerns.

Every computing node hosting autonomic applications is required to run an instance of the CASA Runtime System (CRS). The CRS has two responsibilities: firstly, it monitors the execution environment on behalf of the running applications. Secondly, in case of significant changes in the execution environment, the CRS carries out the adaptation of the affected applications.

The adaptation policy of every application is defined in a so-called application contract. The application contract is external to the application and is specified using an XML-based language, thereby facilitating changes in the adaptation policy at runtime. This ensures that the user / administrator has a control over the adaptation policy, although the adaptation is carried out in a user-transparent manner.

As illustrated in Figure 1, the CASA adaptation process involves three steps.

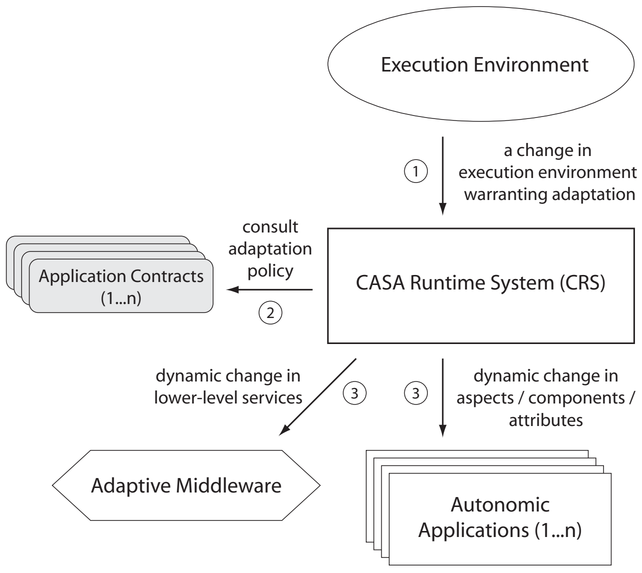
Execution Environment
1
a change in execution environment warranting adaptation
consult adaptation policy
2
Application Contracts (1...n)
CASA Runtime System (CRS)
dynamic change in lower-level services
3
3
dynamic change in aspects / components / attributes
Adaptive Middleware
Autonomic Applications (1...n)

Fig. 1. Working of CASA

Every time the CRS detects a change in the execution environment (step 1), it evaluates the application contracts of the running applications with respect to the changed state of the execution environment (step 2). If the CRS discovers a need for adapting certain applications, it carries out the adaptation of the affected applications, in accordance with the adaptation policies specified in the respective application contracts (step 3).

In the following sections, we discuss details of the CASA framework.

## III. Design of the CASA framework

Figure 2 depicts the CASA framework. The entities within the dotted area represent the CASA Runtime System (CRS). These entities are responsible for monitoring the execution environment and adapting applications. In the following, we discuss details of each of these entities.

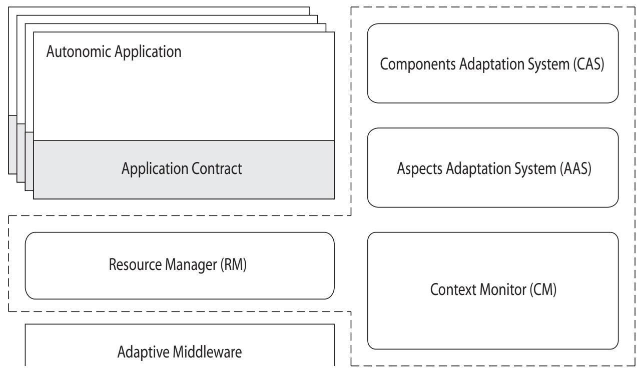

Fig. 2. The CASA framework

### A. Monitoring execution environment

The execution environment of an application can be divided into: *contextual information* (user's location, identity of nearby objects or persons etc.) and *resources* (bandwidth, battery power, connectivity etc.).

Contextual information refers to (purely) the *information* about the context of an application that may influence the service provided by the application (such as locational information, temporal information, atmospherical information etc.), while the resources form the *physical infrastructure* available to the application for providing this service (such as communication resources, data resources, computing resources etc.).

*Monitoring contextual information:* The Context Monitor (CM) in CASA is responsible for monitoring contextual information.

Monitoring the contextual information relevant to an application consists of the following steps:

- acquiring the data related to contextual information,
- structuring the acquired data based on an application domain-specific ontology, and
- deducting the final knowledge, i.e. the contextual information relevant to the application, from this data.

Each of these three steps is implemented at a separate individual level in the CM, as shown in Figure 3.

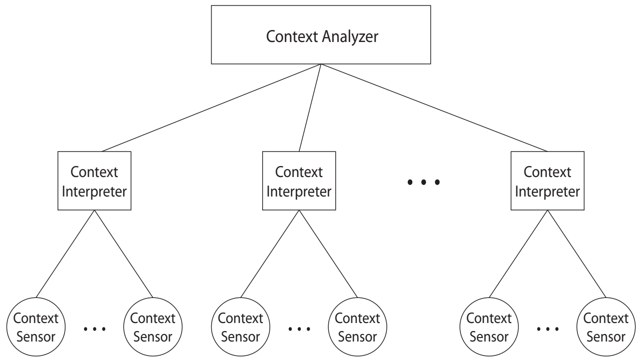

Fig. 3. Context Monitor

At the lowest level, a variety of Context Sensors are used to acquire the data related to contextual information. At the middle level, the acquired data is structured based on an application domain-specific ontology by a collection of Context Interpreters, each one responsible for a different context parameter. At the top level, a Context Analyzer is used to deduce the contextual information relevant to the application from this data.

*Monitoring resources:* For monitoring local and network resources, several resource monitoring services have been developed that operate at the platform (operating system, network) level, where resources can be monitored efficiently (e.g. Remos [8], Dproc [1]). Therefore, for monitoring local and network resources, CASA relies on an external resource monitoring service.[1] The Resource Manger (RM) in CASA is responsible for interacting with the underlying resource monitoring service.

For monitoring any remote resources required by an application, the RM includes a Remote Resource Coordinator (RRC) entity. Remote resources can be discovered using a variety of means. For example, certain standard data resources required by an application can be discovered by the RRC using a remote resource discovery infrastructure such as Jini [4] etc. On the other hand, any specific specialized services, provided by some peer applications, can be discovered explicitly by the application itself and communicated to the RRC.

### B. Adapting applications

Application adaptation can be realized using one or more of the following adaptation mechanisms supported by CASA, depending on the adaptation needs of a specific application.

*Dynamic change in lower-level services:* Lower-level services here mean the underlying services required by an appli-

[1]As discussed later, several adaptive middleware systems have built-in resource monitoring services that can be made available to CASA.

cation for its execution, e.g. data transmission, compression, caching, video coding/decoding etc.

An application adaptation through a dynamic change in lower-level services is typically required in response to a change in the availability of resources. It is usually in the form of a resource vs. resource tradeoff, or a resource vs. quality of service tradeoff. An example of a resource vs. resource tradeoff is to compress the data being transmitted over a communication channel (by invoking a lossless data compression service), in response to a drop in the bandwidth available on the channel. Compressing the data will result in increased CPU consumption, but at the same time it will also result in saving bandwidth. Note that the quality of service (here, the quality of data) is not affected by this adaptation. An example of a resource vs. quality of service tradeoff is to reduce the frame-rate or resolution of the video being transmitted over a communication channel, in response to a drop in the available bandwidth.

Several adaptive middleware systems have been developed that are capable of adapting an application by dynamically changing the lower-level services used by the application. Some of these middleware systems are reflective in nature, i.e. support external regulation of their adaptation strategy, and thus can be integrated with CASA.

Many of the adaptive middleware systems have a built-in resource monitoring service for monitoring local and network resources. Thus, an adaptive middleware system can serve the dual purpose of monitoring local and network resources, and dynamically changing lower-level services.

The RM (Resource Manager) in CASA is responsible for interacting with the underlying adaptive middleware system.

One such adaptive middleware system that can be used for monitoring local and network resources, and dynamically changing lower-level services is Odyssey [12].

Odyssey is able to monitor local and network resources, and provide information about the availability of these resources to the RM. Any runtime changes in the availability of resources can be communicated instantly by Odyssey to the RM. Resource monitoring is carried out at the operating system level in Odyssey, thus inducing efficiency and enabling the RM to respond swiftly to any changes in the availability of resources.

Odyssey is able to further invoke appropriate processing of the application data at the middleware level, according to the instructions of the RM. Data processing at the middleware level is carried out using specialized code components called *wardens*. For every type of data processing, various alternative wardens may be implemented, differing in the data fidelity levels and resource requirements. Depending on the current execution environment conditions, the RM may switch between alternative wardens at runtime.

CASA can even be integrated with more than one adaptive middleware system offering different adaptation capabilities.

However, an application adaptation through a dynamic change in lower-level services is applicable only in limited scenarios – limited kinds of applications, execution environment conditions, and the amount of variations therein. In order to deal with the other scenarios, one or more of the following adaptation mechanisms may be used.

*Dynamic weaving and unweaving of aspects:* AOP (aspect-oriented programming) [6] enables separating the crosscutting functionality of an application from its core functionality. Many times a change in the execution environment requires a corresponding change in the crosscutting functionality of an application, without really affecting the core functionality of the application. Examples of such adaptation are changing the security behavior of an application when it moves from a low-risk area to a high-risk area, changing the persistence behavior in response to a loss of connection with a data storage etc.

Thus, the ability to dynamically weave and unweave aspects into / from an application, generally referred to as dynamic AOP, presents a powerful adaptation mechanism.

For dynamic weaving and unweaving of aspects, CASA relies on a dynamic AOP system called PROSE [11], which is a flexible and efficient Java-based system developed for this purpose.

The Aspects Adaptation System (AAS) in CASA is responsible for interacting with PROSE. The AAS can pass the appropriate aspect details (name and location of the aspect file) to PROSE at runtime for dynamically weaving the aspect into an application. The aspect file contains the information about the join-points (the execution points where the aspect needs to be weaved) as well as the actual aspect code to be executed at these points. PROSE is able to intercept the join-points in a running Java application, and invoke the corresponding aspect code at these points. At the end of the execution of the aspect code, the control is returned to the next execution point in the application. Similarly, the AAS can instruct PROSE to unweave an already weaved aspect.

However, if a change in the core functionality of an application is required in response to a change in the execution environment, rather than in the crosscutting one, then the following adaptation mechanisms may be used.

*Dynamic recomposition of application components:* Modern software applications are composed of components, where each component implements a subtask of the application. In a component-based application development, the components encapsulate their implementation details and interact with each other only through their well-defined interfaces.

This makes it possible and convenient to dynamically change the core functionality of an application through dynamic recomposition of application components. A change in the core functionality is most likely required in response to a change in contextual information, but may also be required for significant variations in resource availability.

For example, if the contextual information related to a *Tourist-Guide* application changes from *shopping mall* to *open-air cinema*, the application needs to provide relevant information about the weather conditions and show-timings, in place of the information about the availability of the items in the user's shopping list in the shopping mall. This kind of a change in the core functionality can be accomplished through dynamic recomposition of application components. Needless to say that the possibilities for application adaptation through dynamic recomposition of application components are enormous.

For dynamic recomposition of application components, CASA follows an indigenous approach.

The Components Adaptation System (CAS) in CASA is responsible for dynamic recomposition of application components. A dynamic recomposition of application components may involve addition, removal and/or replacement of the components at runtime. The CAS takes care to ensure that the consistency of the application is not compromised as a result of dynamic recomposition. Ensuring the consistency involves, among other things, transferring the state of an outgoing component to its successor (in case of a dynamic replacement), and maintaining the integrity of the interactions ongoing at the time of dynamic recomposition. More details on this approach can be found in [9].

However, sometimes only a dynamic change in certain attributes of an application may be required, rather than a dynamic recomposition of application components. For a dynamic change in application attributes, the following adaptation mechanism may be used.

*Dynamic change in application attributes:* Examples of application adaptation through a dynamic change in application attributes include changing the value of a certain timeout period, frequency of data transmission, size of each transmission, or some other threshold parameters affecting an application's behavior in response to a change in resource conditions.

For a dynamic change in application attributes, the concerned application needs to provide appropriate callback methods that can be called by the RM (Resource Manager) at runtime. This allows the application to decide the appropriate timing and order of changing these attributes.

In addition to the above adaptation mechanisms, certain resource-level adaptations are carried out transparently at the level of the RM. Such adaptations involve dynamically negotiating and selecting among the alternative resources available. For example, if the availability of a remote resource being used by an application drops, the RRC (Remote Resource Coordinator) may dynamically switch to an alternative remote resource providing the same type of service but with better availability.

The overall working of the CASA framework is described in Section V. In the next section, the specification of an application contract is discussed.

## IV. The application contract

The adaptation policy of every application is defined in a so-called application contract. The application contract is external to the application, and is specified using an XML-based language. This facilitates easy modification, extension and customization of the adaptation policy at runtime. The contract-based adaptation policy also plays a key role in separating the adaptation concerns of an application from its business concerns.

An excerpt of an application contract of a *Tourist-Guide* application is shown in Figure 4.

```
<app-contract name="Tourist-Guide">
    <context id="1">
        <params>
            <par name="vicinity" value="museum"/>
        </params>
        <config id="1">
            <resources>
                <res name="bandwidth" unit="Kbps" value="56-35"/>
                .
                .
            </resources>
            <components .../>
            <aspects .../>
            <callbacks .../>
            <llservices .../>
        </config>
        <config id="2">
            .
            .
        </config>
        .
        .
    </context>
    .
    .
</app-contract>
```

Fig. 4. Application contract

The application contract is divided into <context> elements, where each <context> element represents a state of contextual information of interest to the application. The parameters characterizing this state are specified within <params> element.

The <params> element contains one or more <par> elements, with each <par> element corresponding to a distinct context parameter. Every <par> element contains a "name" attribute specifying the name of the context parameter, an optional "unit" attribute specifying the unit of measurement, and a "value" attribute specifying the corresponding value of the parameter. The names of context parameters are standard and unique for the corresponding application domain. Examples of the parameter names for the *Tourist-Guide* application are "vicinity" (referring to a place of interest nearby), "time" (referring to the time of day) etc.

Each <context> element further contains a list of alternative configurations of the application, suited to the particular state of contextual information. These configurations vary in their resource requirements, and are listed in an ordering that reflects their user-perceived preference.

Each <config> element, representing a configuration, specifies the resource requirements of the configuration (<resources> element), the components and aspects constituting the configuration (<components> and <aspects> elements), the callback methods to be called for the configuration (<callbacks> element), and the lower-level services related to the configuration (<llservices> element).

The <resources> element contains a number of <res> elements, with each <res> element representing a distinct resource. A <res> element may represent a local, network or remote resource required by the corresponding configuration. Every <res> element contains a "name" attribute specifying the name of the resource, an optional "unit" attribute specifying the unit of measurement, and a "value" attribute specifying the corresponding resource value. The resource names are standard and unique, i.e. every resource is uniquely represented by a standard resource name. For a quantifiable resource, the resource value is usually specified in terms of a range of numbers separated by a hyphen, with the number on the left being most preferred and the one on the right being least preferred.

Resource requirements can also be specified at a high level of abstraction such as in terms of throughput and packet size, instead of directly specifying them in terms of actual resources such as bandwidth. If the resource requirements are specified at an abstract level, then the RM needs to convert these into actual resource values to be allocated.

The <components> element contains a list of adaptable application components, i.e. those components that may differ from one configuration to another. The non-adaptable components, which remain the same across all configurations, are not specified in the <components> element. In the same way, the <aspects> element contains a list of adaptable aspects. Every component and aspect is specified along with its namespace location, as this is required by the CRS (CASA Runtime System) for activating a configuration. Similarly, the <callbacks> element contains a list of methods to be called, and the <llservices> element contains a list of lower-level services related to the configuration.

All the elements specifying the constituents of a configuration (i.e. the <components>, <aspects>, <callbacks> and <llservices> elements) are optional. That is, any of these elements may or may not appear in a configuration specification, depending on the adaptation requirements of the corresponding application. For example, if an application has no adaptable components, but only adaptable aspects, attributes and lower-level services, then the <components> element will not appear in the specification of an application configuration.

Similarly, if an application needs to respond only to the changes in contextual information, but not to the changes in resource availability, then the <resources> element can be omitted from the configuration specification (e.g. if all the resources required by an application are guaranteed to be available sufficiently). In this case, every <context> element will contain only a single configuration, suited to the corresponding state of contextual information. On the other hand, if an application needs to respond only to the changes in resource availability, but not to the changes in contextual information, then there will be no <context> element in the application contract. In this case, the application contract will contain a simple listing of the alternative configurations of the application, ordered according to their user-perceived preference.

Depending on the current state of the execution environment (contextual information and resources), the appropriate configuration from the application contract is selected and activated by the CRS, as explained in the next section.

## V. Working of CASA

When starting up, an application registers itself with the CRS (CASA Runtime System). As a part of the registration, the application contract is passed to the CRS and is accessible to all the entities of the CRS.

Next, the CM (Context Monitor) discovers the contextual information relevant to the application. The contextual information discovered by the CM is matched with the <context> elements specified in the application contract, in order to determine the currently *valid* <context> element. This is done by matching the values of parameters of the discovered contextual information with the corresponding values specified in the <params> element of every <context> element in the application contract.

It is possible that more than one <context> element specified in the application contract is eligible to be valid simultaneously. For example, one <context> element may be valid if the user is currently in her office-building, and the other may be valid if the user is currently in her office-room. So, if the user is currently in her office-room, then both the above <context> elements are eligible to be valid at the same time. In this case, the ordering of the <context> elements in the application contract is important for determining the currently valid <context> element, as this ordering reflects the relative preference of the <context> elements. That is, the CM identifies the highest listed valid <context> element as the currently valid <context> element. Practically, the CM starts searching for the currently valid <context> element from the top of the application contract, and the search ends as soon as a valid <context> element is found.

There may be a default <context> element in an application contract that is valid when none of the other <context> elements is valid. This default <context> element is listed at the end of the application contract, and need not have a <params> element.

The information about the currently valid <context> element is passed by the CM to the RM (Resource Manager). Recall that every <context> element in the application contract contains a list of alternative configurations, represented by <config> elements, that are suitable for the corresponding state of contextual information defined by the <context> element. As with the ordering of the <context> elements, the <config> elements are also preferentially ordered within every <context> element in the application contract.

Every <config> element contains a <resources> element that specifies the resource requirements of the corresponding configuration. For the discussion here, we assume that CASA is integrated with an adaptive middleware that is able to monitor local and network resources, as well as dynamically change lower-level services.

The RM allocates resources to the application based on the current availability of resources, as informed by the underlying adaptive middleware and the RRC (Remote Resource Coordinator). Since the configurations are listed in a preferential ordering within the <context> element, the RM tries to allocate resources for the first configuration listed in the <context> element. If there are not sufficient resources for the first configuration then it tries the second configuration and so on. The resource allocation phase ends as soon as a match between the resource requirements of a configuration and the current availability of resources is found.

In case of multiple applications contending for the same resources, the RM takes into account the relative priorities (as defined by the user), as well as the adaptation possibilities of the applications for allocating resources. The details of the resource allocation algorithm followed by the RM are out of scope of this paper.

At the end of the resource allocation phase, the RM instructs the underlying adaptive middleware, the AAS (Aspects Adaptation System) and the CAS (Components Adaptation System) to activate the lower-level services, aspects and components related to the selected configuration, respectively. The RM also issues any callbacks specified for this configuration. Recall that the lower-level services, aspects, components and callbacks related to the selected configuration are specified within the corresponding <config> element.

If there is a runtime change in the contextual information relevant to the application, then this change is detected by the CM. The CM communicates the new <context> element to the RM. The RM allocates resources for a new configuration from the new <context> element, following the same procedure as during the initial allocation. The existing configuration is then replaced with this new configuration.

Similarly, if there is a runtime change in the availability of resources, but not a change in the contextual information, then the RM is informed about the change by the underlying adaptive middleware or the RRC. If the new resource availability makes it impossible for the application to continue with its current configuration, then the RM allocates resources for a new configuration from the current <context> element, based on the changed availability of resources. The existing configuration is then replaced with this new configuration.

A dynamic change in configuration may imply a change in lower-level services, aspects, application components, and/or application attributes. These changes are carried out by the appropriate entities of the CASA framework.

## *A. Service notification / negotiations*

An application executing in a dynamic environment is likely to participate in a distributed software system, i.e. collaborate with other applications for a common mission. An application participating in a distributed software system may need to notify its peer applications before activating a particular configuration (both initially and in response to a change in the execution environment), so that the peer applications may adapt accordingly if required. In some cases, the application may even need to carry out service negotiations with its peer applications, in order to select the appropriate configuration to activate. CASA provides support for service notification and negotiations among peer applications.

For service notification or negotiations, the concerned application needs to implement a Service Coordinator (SC) component. In the following, we describe the changes in the working of CASA that are implied by the inclusion of service notification and negotiations.

When service notification is required, the RM informs the SC about the selected configuration at the end of the resource allocation phase. The SC then informs the peer applications about the changes in the application's functionality and performance characteristics, which are implied by changing to the new configuration.

The peer applications need not be developed according to the CASA framework. However, for a CASA-based peer application, the information about the functionality and performance characteristics of the above application is forwarded to the RRC (Remote Resource Coordinator) of the peer application. This enables the RM (Resource Manager) of the peer application to decide the appropriate adaptation based on the changed value of the remote resource (i.e. the functionality and performance characteristics of the above application). Whereas, for a non-CASA-based peer application, the peer application itself is responsible for deciding the appropriate adaptation based on the change in configuration of the above application.

When service negotiations are involved, the resource allocation phase does not end as soon as a match between the resource requirements of a configuration and the current availability of resources is found. Rather, the RM identifies all the configurations from the given <context> element that can be activated in the current availability of resources. The RM passes the list of identified configurations to the SC. The SC sends the information about the functionality and performance characteristics associated with each of the identified configurations to the peer applications. The peer applications are then required to rank these configurations based on the information about their functionality and performance characteristics, and send the ranked list back to the SC. The SC then selects the most appropriate configuration for activation, based on the rankings given by the peer applications, and informs the RM accordingly.

Different peer applications may be assigned different weights by the SC, so that the rankings by these applications are treated accordingly for the final selection of a configuration. Any ties for the top-ranked configuration are resolved by selecting the configuration listed highest in the application contract.

The peer applications need not be developed according to the CASA framework, but they must provide a component for receiving a list of alternative configurations and ranking them. If a peer application also needs to be adapted due to a change in the configuration of the above application, then the ranking is decided based on the relative preferences of the corresponding adaptations of the peer application itself for each of these alternative configurations.

As an example where service negotiations may be required, consider an application transmitting high-quality multimedia (video + audio) to a number of clients. In response to a drop in the available bandwidth, the application may have an option either to switch to a configuration that reduces the quality of audio but maintains the high quality of video, or to a configuration that reduces the quality of video but keeps the quality of audio high. Before actually changing its configuration, the application needs to carry out service negotiations with the clients. The clients' decision, on the other hand, may be governed by the actual content of the transmission. For instance, if the transmission is that of a soccer match, the clients may choose to reduce the quality of audio while maintaining the high quality of video. Whereas, if the transmission is that of a musical concert, the clients may choose to reduce the quality of video without disturbing the quality of audio.

However, service negotiations are not essential for all kinds of applications that participate in distributed software systems. For instance, if in the above example the application transmits sports events only, then it may have the default adaptation behavior of switching to the first kind of configuration, without requiring any service negotiations with its clients.

At the time of registering with the CRS, an application needs to indicate whether service notification or negotiations are required, and pass a reference of its SC component to the CRS accordingly.

### B. *Runtime changes in the adaptation policy*

As seen from the description of working of CASA, the ordering of the <context> elements within an application contract, as well as the ordering of the <config> elements within a <context> element, effectively define the adaptation policy of an application.

The ordering of these elements can be changed by the user at runtime. In addition to changing the order, the user can also remove certain <context> or <config> elements at runtime. This way the user is able to customize the adaptation policy of the application according to her needs or preferences.

For customizing the adaptation policy, a graphical user interface for the application contract is provided, which explains the significance of the various <context> and <config> elements in user-understandable terms. That is, instead of displaying the list of parameters characterizing a <context> element, it displays what the particular state of contextual information means for the user. And instead of displaying the detailed constituents of a <config> element, it displays the appropriate functionality and performance characteristics associated with the corresponding configuration.

In a similar manner, new <context> or <config> elements can be added to an application contract at runtime, which were not foreseen at the time of application development. And any obsolete <context> or <config> elements can be removed from an application contract at runtime.

## VI. Prototype implementation and evaluation

A prototype *Disaster Control* system, based on the CASA framework, has been implemented in Java. This system consists of two applications: *Observer* and *Support*. *Observer* is responsible for monitoring a disaster-affected area, and sending its observations to *Support* over a wireless link. *Support* is, in turn, responsible for coordinating the rescue operations based on the information received by it. A typical deployment of this system has several instances of *Observer* and one of *Support*. Every *Observer* instance needs to move frequently while surveying the disaster-affected area. Because of the nature of the operation, bandwidth fluctuations between *Observer* instances and *Support* are very common.

*Observer* provides a number of alternative configurations with varying resource (mainly bandwidth) requirements. These configurations differ in the quality of data sent from *Observer* to *Support*. Among the various alternative qualities of data supported by these configurations are: high-quality video, low-quality video, high-resolution images, low-resolution images, detailed textual description and brief textual description. Depending on the current resource availability, the appropriate configuration of *Observer* is selected and activated. A change in configuration may involve only a dynamic change in lower-level services (e.g. when switching from high-quality video to low-quality video, only the video coding/decoding service needs to be changed), or it may involve a dynamic change in application components (e.g. when switching from low-resolution images to detailed textual description, the concerned application components need to be changed).

Performance evaluation tests with this prototype have been carried out, and the results are encouraging. These tests were carried out primarily to evaluate the performance of our components adaptation approach. The performance evaluation of the aspects adaptation approach followed by PROSE can be found in [11], and that of the lower-level services adaptation approach followed by Odyssey can be found in [12].

The detailed performance evaluation of our components adaptation approach is given in [3]. Below are some of the indicative results.

Figure 5 shows the time CASA requires for responding to a change in the execution environment. This time includes the time spent in (a) selecting the appropriate applications to adapt based on the relative priorities and adaptation possibilities of the running applications (including deciding the appropriate new configurations for the selected applications), and (b) actually changing the configurations of the selected applications. For the data in this figure, for every change in the execution environment, only a single application needed to be adapted and a change in configuration involved a dynamic replacement of a single application component. In Figure 5, the X-axis represents the number of applications running, and the Y-axis represents the response time by CASA. The increase in response time with respect to the increase in number of running applications is due to the activity (a) above.

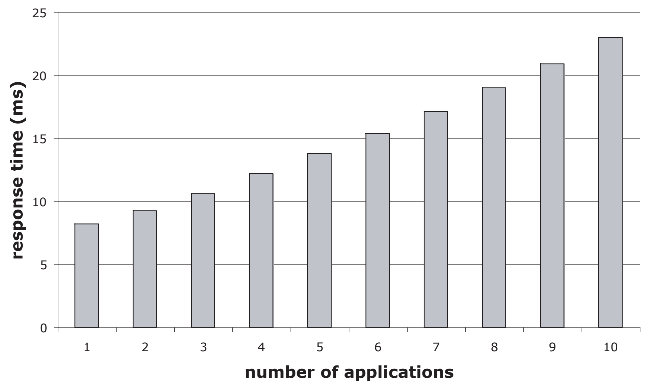

Fig. 5. Response time by CASA for a change in execution environment

Figure 6 shows the increase in execution time of an application because of dynamic recomposition of application components. In Figure 6, the X-axis represents the number of components to be replaced during a dynamic recomposition (time for components addition and removal was found to be lower than that for replacement), and the Y-axis represents the total execution time of the application. The lighter portion of the bars indicates the normal execution time (100 ms, when no recomposition of application components is involved), and the darker portion indicates the overhead due to a one-time recomposition of application components. As observed from this figure, even for a small normal execution time of 100 ms, the overhead due to a recomposition of application components is relatively quite small (please note that this overhead is independent of the actual normal execution time of the application).

For the data in Figure 6, the components were stateless, i.e. no state was transferred from an outgoing component to its successor. However, the overhead due to the state transfer was found to be very small – in the order of a few microseconds (10 $\mu$s for a state with a single parameter to 25 $\mu$s for 10 parameters).

The above tests were carried out on Linux, running on an AMD Athlon XP 1900+ processor machine with 1024 MB RAM. On a much slower machine, the overhead was around 3-4 times higher than the above values.

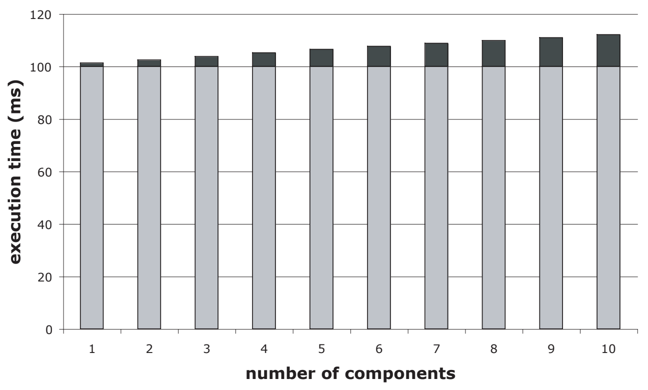

Fig. 6. Overhead due to dynamic recomposition of application components

The overall performance results obtained indicate that the overhead due to the CASA framework is tolerable for most practical applications, and that the benefits of dynamic and transparent adaptation, provided by CASA, far outweigh the performance overhead.

## VII. RELATED WORK

Kephart and Chess [5] envision autonomic computing systems to be able to deal with increasing software and environment complexity, thanks to the self-managing characteristic of these systems. Recently some approaches have been proposed with the aim to turn this vision into reality.

White et. al. [17] propose an architectural approach to developing autonomic systems. In this approach, an autonomic system is made up of autonomic elements, where each autonomic element is self-managing in its own behavior as well as in its interactions with other elements. This work describes the requirements to be satisfied by autonomic elements and systems, but it does not give the details of carrying out self-management.

Liu et. al. [7] present a component-based framework for autonomic applications, where the behaviors and interactions of application components can be adapted dynamically. The adaptation decisions here are governed by the rules associated with every application component. However, in this approach, the number of rules for controlling the behavior and interactions of every application component can potentially be quite large, inducing significant performance overhead due to the execution of these rules at runtime.

David and Ledoux [2] present an approach for runtime adaptation of applications by activating and deactivating certain meta-level components associated with the normal application components, in response to changes in the execution environment. However, the scope of adaptation is restricted here, as the meta-level components are limited to doing some kind of pre or post processing that can be adapted dynamically.

Several other approaches have been proposed for adapting the lower-level services used by applications at the middleware level. Some of these approaches are reflection-based, and thus can be integrated with CASA by applying suitable instrumentation. Examples of such approaches are Odyssey [12], InfoFabric [13], QuO [18] etc.

Some more work has been done in the area of context monitoring. However, most of the approaches have been tightly coupled with the applications for which they have been developed. For example, some systems such as Active Badge [15] and ParcTab [16] identify the user's location and activity etc., and use this information for providing context-dependent services to the user.

The Context Toolkit [14] provides general mechanisms for context monitoring to aid the development of context-aware applications. It employs the concept of *context widgets* that can be used by an application for acquiring the data related to contextual information. This way, the context widgets insulate an application from the mechanics of context sensing. The CM (Context Monitor) in CASA may use the Context Toolkit for acquiring the data related to contextual information, before interpreting and analyzing this data to arrive at the contextual information relevant to an application.

## VIII. Conclusion

CASA provides a framework for enabling the development and operation of autonomic applications. In this paper, the design of the CASA framework and its overall working were discussed.

The complexity involved in developing autonomic applications is significantly reduced using CASA, by virtue of separation of the adaptation concerns of an application from its business concerns. The decoupled architecture of CASA, in particular the contract-based adaptation policy followed by CASA, makes this separation possible. CASA further provides a runtime system for dealing with the adaptation concerns. With a view to provide a general framework that is able to comprehensively meet the adaptation needs of a broad and diverse set of applications executing in dynamic environments, CASA integrates a number of adaptation mechanisms for supporting adaptation at different levels of an application. The contract-based adaptation policy, in addition, facilitates changes in the adaptation policy at runtime.

The implementation and evaluation of a CASA prototype have been discussed in this paper. The implementation of a real-world application based on CASA is a work planned for near future.

## References

[1] S. Agarwala, C. Poellabauer, J. Kong, K. Schwan and M. Wolf. System-Level Resource Monitoring in High-Performance Computing Environments. *Journal of Grid Computing*, 1(3), 2003.

[2] P. David and T. Ledoux. Towards a Framework for Self-Adaptive Component-Based Applications. *Proc. of 4th International Conference on Distributed Applications and Interoperable Systems*, 2003.

[3] A. Gygax. *Studying the Effect of Size and Complexity of Components on the Performance of CASA*. Internship Report, Institut für Informatik, University of Zurich, 2004. http://www.ifi.unizh.ch/req/ftp/papers/casa-perf.pdf

[4] Jini Network Technology. http://www.jini.org/

[5] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1), 2003.

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier and J. Irwin. Aspect-Oriented Programming. *Proc. of 11th European Conference on Object-Oriented Programming*, 1997.

[7] H. Liu, M. Parashar and S. Hariri. A Component Based Programming Framework for Autonomic Applications. *Proc. of 1st International Conference on Autonomic Computing*, 2004.

[8] B. Lowekamp, N. Miller, R. Karrer, T. Gross and P. Steenkiste. Design, Implementation, and Evaluation of the Remos Network Monitoring System. *Journal of Grid Computing*, 1(1), 2003.

[9] A. Mukhija and M. Glinz. Runtime Adaptation of Applications through Dynamic Recomposition of Components. *Proc. of 18th International Conference on Architecture of Computing Systems*, 2005.

[10] 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.

[11] A. Nicoara and G. Alonso. Dynamic AOP with PROSE. *Proc. of 1st International Workshop on Adaptive and Self-Managing Enterprise Applications*, 2005.

[12] 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 Systems Principles*, 1997.

[13] C. Poellabauer, K. Schwan, S. Agarwala, A. Gavrilovska, G. Eisenhauer, S. Pande, C. Pu and M. Wolf. Service Morphing: Integrated System- and Application-Level Service Adaptation in Autonomic Systems. *Proc. of 5th International Workshop on Active Middleware Services*, 2003.

[14] D. Salber, A.K. Dey and G.D. Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. *Proc. of ACM Conference on Human Factors in Computing Systems*, 1999.

[15] R. Want, A. Hopper, V. Falcao and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1), 1992.

[16] R. Want, B.N. Schilit, N.I. Adams, R. Gold, K. Petersen, D. Goldberg, J.R. Ellis and M. Weiser. An Overview of the ParcTab Ubiquitous Computing Experiment. *IEEE Personal Communications*, 2(6), 1995.

[17] S.R. White, J.E. Hanson, I. Whalley, D.M. Chess and J.O. Kephart. An Architectural Approach to Autonomic Computing. *Proc. of 1st International Conference on Autonomic Computing*, 2004.

[18] 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.