Electronic Communications of the EASST
Volume 11 (2008)

Proceedings of the First International DisCoTec Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2008)

Divide and Conquer – Organizing Component-based Adaptation in Distributed Environments

Ulrich Scholz and Romain Rouvoy

12 pages

Guest Editors: Romain Rouvoy, Mauro Caporuscio, Michael Wagner
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Divide and Conquer – Organizing Component-based Adaptation in Distributed Environments

Ulrich Scholz[1] **and Romain Rouvoy**[2]

[1] European Media Laboratory GmbH
Schloß-Wolfsbrunnenweg 33
69118 Heidelberg, Germany
ulrich.scholz@eml-d.villa-bosch.de

[2] University of Oslo, dept. of Informatics
P.O.Box 1080 Blindern
0316 Oslo, Norway
rouvoy@ifi.uio.no

Abstract: This paper introduces a divide and conquer approach for organizing the adaptation of distributed applications in a potentially large number of interacting middleware instances. In such an environment, a centralistic and static adaptation reasoning *i)* is inadequate and *ii)* gives the same priority to all applications. The divide and conquer method aims at minimizing the interference between running applications, allowing users to weight the priority of applications, and organizing the adaptation and the reasoning about the adaptation in a decentralized and flexible way.

Keywords: Adaptive middleware, distributed adaptation reasoning

1 Introduction

This work is concerned with the task of adapting a number of large, distributed applications in mobile environments subject to frequent context changes. We consider this problem within MUSIC [MUS], an initiative to develop a comprehensive open-source platform that facilitates the development of self-adaptive software in ubiquitous environments. One aim of MUSIC is a large-scale deployment of multiple middleware instances. Some of these instances host two or more applications, some applications are distributed on two or more instances. Furthermore, the topology of middleware instances and applications is transient—*i.e.*, they can appear and disappear at any time. For such a collection of middleware instances, users, applications, devices, connections, and other artefacts related to adaptation, we assign the term *theatre*.

Current solutions to the adaptation problem use a centralized and coarse-grained approach, which is not suited for large theatres: If one part of an application needs adaptation, the whole application—or even a set of applications—is adapted in combination by a dedicated solver. The solver considers all alternative configurations of all these applications at once and chooses the configuration that yields the best utility. This approach is not feasible for large theatres due to the combinatorial explosion of alternative configurations and because it interferes with parts of the theatre even if such interference is not required.

As a solution, we propose the *Divide and Conquer* (D&C) approach to organize the adaptation tasks of a theatre in a decentralized and distributed way. D&C considers units that are smaller than single applications and provides techniques that allow the adaptation of partial applications.

In the following, we describe the adaptation problem and introduce the basic D&C ideas (cf. Section 2). Then we detail these ideas, namely the application packs (cf. Section 3), resource distribution (cf. Section 4), the decomposition tree (cf. Section 5), and the strategies (cf. Section 6). Before listing related work (cf. Section 8) and giving concluding remarks (cf. Section 9), we explain D&C with an example scenario (cf. Section 7).

2 The Adaptation Problem and the D&C Approach

The MUSIC project's focus is the adaptation of component-based applications. Applications are assembled of components—*i.e.*, pieces of code—and several different collections of components, each called a *variant*, can realize the same application. Many variants provide the same function to the user (*e.g.*, participation in a picture sharing community), but often with different non-functional properties (*e.g.*, quality of service and CPU usage). The degree to which the non-functional properties of a variant satisfy the user is called the *utility* of that variant [FHS+06].

Adapting an application means choosing and commissioning one of its variants, while the *adaptation problem* means adapting an application such that it has the highest—or a sufficiently high—utility in a given situation. An adaptation middleware, such as MUSIC, aims at a solution to the adaptation problem such that every application maintains its functionality and a high utility despite of changes in the theatre.

Not all variants of an application are valid: They have to satisfy *architectural constraints* [KRG07]. Two examples for such constraints are the existential dependency between two components that holds if choosing one only makes sense if the other is chosen too, and a dependency where two components are mutually exclusive or require each other. Adaptations can introduce new dependencies to an application and remove existing ones.

Existing adaptive systems usually consider entire applications: If some part of an application requires an adaptation then the whole application has to be adapted. In MADAM [MAD06] the unit of adaptation is even wider, as it comprises all applications on a local device and all parts of these application deployed on other devices. Adapting large units guarantees an optimal utility but has several drawbacks. One of them is the combinatorial explosion in the number of variants that have to be considered. If an application always consists of 5 components and there are 5 variants for each then the application has $5^5 = 3125$ variants. For an application comprising 10 components, or for the combination of 2 applications with 5 components each, there are already about 10 million of them. Although increasing processing power, restrictions on valid choices of variants, and heuristics allow solving large adaptation problems, even medium sized collections of applications are often infeasible for a global method.

Another reason for the inadequateness of current approaches is that they affect parts of the theatre that are better left untouched. Adapting an application involves stopping it and re-starting it after some time. Nevertheless, some application parts, such as video stream receivers, may not support to be suspended and resumed dynamically. In addition, users are willing to accept such interruption of service only if the disruption is very short or they observe a clear advantage. As the adaptation time is often pronounced, adapting large parts of theatres can lead to many such undesired outages for a user.

In contrast, D&C forgoes globally optimal solutions by adopting a more fine-grained approach

towards adaptation. Applications are divided into smaller units, called *application parts*, and D&C organizes the adaptation and distribution of collections of such parts, called *packs*, in a decentralized and flexible manner. Then, the adaptation of each part is treated independently as black box by D&C.

Furthermore, current approaches do not distinguish between the application adaptation and its organization. As a result, the realization of adaptation control requires in-depth knowledge about the logic of its application and the environment to foresee possible adaptation situations. D&C provides a clear separation between both aspects of adaptation reasoning.

In detail, D&C comprises five concepts to adaptation organization: (1) the concept of application parts and packs, (2) the splitting and merging of packs, (3) the reasoning about pack layout, (4) the resource negotiation between packs, and (5) the decentralized, flexible coordination of the adaptation. The first two points are covered by the following section, resource negotiation by Section 4, and the decomposition tree by Section 5. Reasoning about pack layout is not an essential part of D&C and we omit it here for space reasons.

3 Packs – Adapting Collections of Application Parts

The D&C units of adaptation are *parts* and *packs*. Applications are divided into parts that can be adapted independently or in combination; a pack is a collections of such parts. The handling of packs—*i.e.*, their adaptation, division, aggregation, and relocation, as well as the organization of these operations—forms the essence of the D&C approach.

With respect to complexity and autonomy, application parts are positioned between components and full applications. Like applications, they are built of components and have their own utility function. Their overall utility is the product of the ones of their parts. As for components, architectural constraints between parts can restrict their variant space at a certain time.

Note that the division of an application into parts and packs does not increase its adaptation complexity compared to the same application built of components: Adapting all parts in combination takes the same time as adapting all components in combination and choosing to not adapt some parts can only reduce the required effort.

Packs are a purely logical assembly of application parts. Parts in a pack are usually handled as a whole. In particular, all parts of the same application that are in one pack are always adapted in combination, eliminating the need for D&C to handle the architectural constraints between these parts explicitly. Each pack can be adapted independently of other packs and only the applications of the part in a pack have to be stopped during its adaptation.

The division into packs reflects the results of D&C's reasoning about which application parts are likely to be adapted together and which independently. If two application parts a and b of different applications in the same pack adapt in combination then the adaptation mechanism considers each element of the cross product $p_a \times p_b$. The time to adapt them is $t_a \cdot t_b$, where p_a and p_b are the sets of variants of a and b, respectively, and the time to establish these sets is t_a and t_b. In case a and b are in different packs, the elements of p_a and p_b are considered independently and the adaptation time is $t_a + t_b$ in the worst case. In other words, by placing a and b in different packs, we go from an exponential alternative space to a linear one.

The aggregation of application parts to packs allows to organize the adaptation and to adjust

the reasoning effort against the quality of adaptation. On the one hand, the larger the packs on a machine are, *i)* the higher the expected utility after their individual adaptation and *ii)* the easier to find a good resource distribution among them (cf. Section 4). On the other hand, the smaller the packs—-*i.e.*, the higher the number of packs on the machine—the faster the adaptation of individual packs. Thus, changing the composition of packs allows the middleware to balance reasoning time and adaptation quality.

In more details, the motivations for merging two packs are as follows. (1) Optimization of utility: In case an application was distributed over two or more of the initial packs, the merged pack might offer a better utility function for this application. (2) Lower resource usage: The improved utility function can lead to an adaptation that uses less resources, but does not decrease utility. (3) Decrease the need for and increase the quality of resource negotiation: The fewer packs there are, the easier it is to distribute resources between them.

Packs are split for two reasons. (1) To minimize adaptation time: Smaller packs have a potential smaller adaptation time. Prerequisites are that resources are sufficient and the new packs have few architectural dependencies. (2) To change the layout of packs: In a D&C setting, packs are the unit of relocation.

We assume that the realization of splitting and merging of packs is transparent with respect to time and memory. In particular, they do not change the resource consumption of the involved packs and packs resulting from a split have the same combined resource usage like the initial pack. Although splitting and merging packs takes time, we assume that this time is negligible such that packs can be split and merged without interfering with the applications.

The question of where to split a pack, which packs to merge, and when to do so is a question of strategies, which we discuss in Section 6.

4 Negotiation: Balancing the Resource Consumption among Packs

Applications on a machine do not run in isolation: They share the device resources. With *resource negotiation*, D&C tries to prevent the combined adaptation of all applications in case a single one of them has to adapt. The main idea is that each pack is assigned a specific amount of every resource under negotiation. When changes in the theatre trigger an adaptation, the affected packs adapt locally within the allocated resource budgets. The hope is that such a local adaptation of packs result in new variants that are “good enough”, while the other packs can remain untouched. Currently, we assume that an estimate within ±20% of reported utility value suffices.

4.1 Weighting the Priority of Applications

D&C gives the user additional control about the resource distribution. It allows her/him to rate applications according to her/his interests by assigning a number between 0 (irrelevant) and 1 (highly important). The priority is independent of the utility and the current variant of the application, thus an adaptation does not change it. Priority and utility allow the middleware to find the applications that mostly contribute to the user satisfaction: These are those having the highest product of application priority $\times$ current application utility.

Note that the priority of one application is independent of other applications. As an effect, the

	memory		
	< 20	< 40	≥ 40
CPU < 100	0.0	0.0	0.0
CPU < 200	0.0	v_3, 0.3	v_5, 0.4
CPU ≥ 200	0.0	v_3, 0.3	v_6, 0.7

$$happiness^{v_3}_{\text{mem}}(x) = \begin{cases} 0.0 & \text{if } x < 20 \\ 0.3 & \text{if } 20 \leq x < 40 \\ 0.4 & \text{if } 40 \leq x \end{cases}$$

$$happiness^{v_3}_{\text{CPU}}(x) = \begin{cases} 0.0 & \text{if } x < 100 \\ 0.3 & \text{if } 100 \leq x \end{cases}$$

Figure 1: Example of happiness functions. The table on the left give the utilities (and associated variants) of an application part for different assignments of resources. The two functions on the right are the happiness functions for variant v_3 with CPU less than 200 units.

sum of *priority* values for a user's applications can result in a number higher than 1. The reason is that we expect the user to manually set the priority and we do not expect her/him to normalize the values.

4.2 Happiness – Estimating Utility

Resource negotiation is based on estimating the utility of packs given an allocated resource budget. Optimal utility values and assignment of resources require global adaptation reasoning. Because the aim of resource negotiation is to decide whether adaptation reasoning is necessary, it can only be based on estimates. D&C estimates utility by *happiness functions*.

Let us first detail happiness functions for application parts. Assume that a part p depends on resource types R that D&C negotiates and other context information C that is not negotiated by D&C. If the context in C remains unchanged, then the utility of p depends on the amount of resources in R that it can consume. Each variant v of p has a fixed utility u_v if its resource needs are met by the assignment to R; if not, the part p is not able to run properly. To find the optimal utility of p for a specific resource assignment, we have to explore all variants of p, rejecting the invalid ones, and pick the one with the highest utility among the remaining.

Consider Figure 1 as an example. Assume that R consists of the resources memory and CPU, and an application part p has six variants that all require different amounts of these resources and yield different utilities. With the current context C, three of these six variants are realizable and are not dominated by another variant, *i.e.*, there is no other variant that can be realized with the same resources and that has a higher utility. If, for example, p is assigned 50 units of memory and 150 units of CPU then v_3 is the optimal variant of p yielding an utility of 0.3. Now, if the availability of resources changes, then finding the new optimal variant requires recalculating the table or its stored copy. But, if we assume that only one resource can change, then it suffices to store all the different values for memory, given CPU $= 100$, and all different values for CPU, given memory $= 40$. These values are combined to piecewise define the *happiness functions*.

Happiness functions of parts can be calculated as by-product of adaptation reasoning. In particular, if the adaptation reasoning enumerates all variants, then the happiness functions can be constructed during the search.

Happiness functions for packs are the combination of those of parts. While happiness functions for parts always give the correct results, those for packs only approximate the real utility. The reason is that two or more parts can adapt individually without an adaptation of the whole

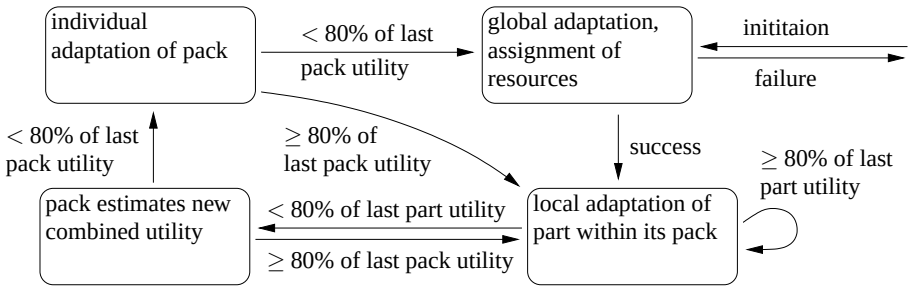

Figure 2: Life cycle of a resource negotiation node. Note that only the regular state transitions are shown. The figure does not show transitions taken in exceptional situations, *e.g.*, when new packs are established and when large amounts of free resources become available. Transitions labeled with "80%" refer to estimates compared to the last calculated utility.

pack. Although each new happiness function is correct for the given resource assignment, a change of this assignment itself is not considered. Therefore, the pack happiness functions after a global adaptation can differ from the locally adjusted ones.

4.3 Distributing Resources

The life cycle of a resource negotiation node is shown in Figure 2. It is an interplay of global and local adaptation. The automaton starts with a global adaptation of all participating packs. If it fails then local adaptation cannot find a solution either and the resource use on the considered device has to be reduced, *e.g.*, by pack relocation. On success, the resulting initial resource assignments are handed to the packs. The automaton changes to the lower right state, where parts that are affected by a context change adapted individually. If the estimate by the happiness function indicates that the result is insufficient, then the automaton visits the states clockwise and the scope of adaptation widens: First the pack utility is estimated, then the whole pack adapts, and finally a global adaptation is performed. If in any of the states the current utility estimate is "good enough", then the automaton goes into the lower right state again.

For reasons of readability, Figure 2 does not show three transitions of the automaton that connect the upper right state from the remaining three. These state changes are taken if an increase in the amount of free resources results in a combined expected utility of the packs that exceeds 120% of the current value, if a pack is relocated onto the machine, and after starting up an application. For reasons of brevity, we also left out the handling of priorities in the explanation of resource distribution.

5 Decomposition Tree: Controlling the Organization of Packs

D&C organizes the adaptation of applications in a theatre in a distributed and self-organized way without using a central controller. This organization is performed by a so-called *decomposition tree—i.e.*, a weakly-connected, *directed acyclic graph* (DAG) with a single root. We use the term "tree" instead of DAG because, usually, we do not regard pack nodes as part of the graph.

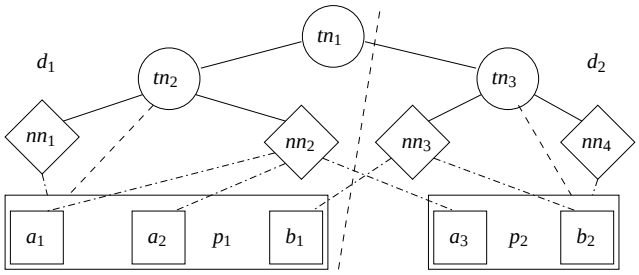

Figure 3: A decomposition tree. Square nodes denote packs while negotiation nodes have a diamond shape. The remaining tree nodes are depicted circular. The packs and the nodes are hosted by two devices d_1 and d_2, separated by the dotted border.

Physically, the nodes of the tree are data structures manipulated by the middleware instances; some part of the data constitutes the internal state of the node. The tree is distributed and there is no central middleware instance that has complete knowledge about all nodes. Nodes are controlled by rules and a node being active means that its rules are matched against its internal state. The physical decomposition of the nodes allows several nodes to be active simultaneously.

Logically, the decomposition tree represents the result of the organization at a certain point in time. The nodes of the tree are annotated with local information about the organization process, which is the internal state of the node, such as the node's parent and children, information about the node's device, and past decisions. The collection of all such internal states is the current state of the D&C method.

Operationally, the nodes dynamically change the tree according to rules. On a certain context change, *e.g.*, the appearance of a new middleware instance, nodes become active and update the tree until a new final form is reached and activity ceases. Thus, the partitioning reflects the evolution of the application configuration. Usually, only a few nodes are active and activity is deliberately handed from node to node.

5.1 Nodes of the Decomposition Tree

The nodes of the DAG are of one of the three types: *tree node*, *negotiation node*, and *pack node*. The root is always a tree node and tree nodes can have children of all three kinds. Negotiation nodes are restricted to have pack nodes as children and pack nodes are always leaves of the DAG.

Each device has one negotiation node that distributes its system resources; a pack on a device is automatically child of this node. D&C uses a predefined list of resource types that a device can offer, such as memory and CPU usage. Each application has a direct negotiation node that holds information about the dependencies between the application's parts. Different resource and application negotiation node have no direct relation to each other.

Figure 3 illustrates a decomposition tree that controls five application parts belonging to two applications: Application a is divided into parts a_1 to a_3 and application b is divided into parts b_1 and b_2. Parts a_1, a_2, and b_1 form pack p_1 that is under control of the node tn_2 while the pack p_2, consisting of parts a_3 and b_2, is handled by node tn_3. Negotiation nodes nn_2 and nn_3 handle the direct dependencies between the respective packs. Node tn_1 is the root of the decomposition

tree, tn_2 and tn_3 are leaf nodes (negotiation nodes and pack nodes are not considered regular nodes of the distribution tree). The packs and the nodes are distributed over two devices d_1 and d_2, denoted by the dotted border. The resources of d_1 are negotiated by node nn_1, while nn_4 does the same for device d_2.

5.2 Working of the Decomposition Tree

The operations of the decomposition tree can be divided into three classes: Atomic operations, complex operations, and strategies. Atomic operations are realized directly by the middleware. Each node has available a set of atomic operations used to examine and alter the decomposition tree. Examples are the migration of its children to another node or the participation in a process for electing a common root node. Strategies decide which complex operation to perform in which situation (cf. Section 6).

Complex operations correspond to reactive actions to apply on the decomposition tree. They are composed of basic operations of one or more nodes. Examples are "Join two decomposition trees" and "React to a failing negotiation node". Different compositions can result in similar complex operations (*e.g.*, for the first example) and for the same purpose there can be complex operations with different outcomes (*e.g.*, for the second).

We have realized four classes of complex operations. (1) Operations that organize the adaptation of applications: These are the splitting and merging of packs and their re-distribution. (2) Operations resulting from changes in the status of an application: A starting application gets its own pack on the machine it is started on. On termination, application parts are removed from their packs. If a new application part is created as a result of an adaptation—*i.e.*, not by the replacement of one part by another—then it is placed in a new pack. If a pack becomes empty because of the termination of an application or because of an adaptation, then it is removed. (3) Operations that handle the sudden disappearance of a connection or of a device. (4) Decomposition tree maintenance operations: These balance the tree, split nodes that have too many children, and merge those with too few.

6 Strategies: Selecting the Adaptation Heuristics

The previous sections explained techniques that allow to reason about and to control the organization of the adaptation of theatres, the principles of this reasoning are given by *strategies*. They are implemented by rules and each node has its own rule-based control loop. Strategies are currently under investigation and we develop a simulator to identify and compare different strategies.

In detail, each node has an internal state that holds all the knowledge that the node has about the theatre. For example, a tree node knows about its parent, its children, its hosting middleware instance, and its past decisions. The internal state is updated either by one of its rules or by the middleware. The rules are condition-action pairs that match the internal state to state changes and to atomic operations as explained in Subsection 5.2.

Strategies control the organization of the adaptation—*i.e.*, which packs to split and where, which packs to merge, and when to relocate a pack. Regarding the decomposition tree, strategies

have to determine the overall structure of the tree and the location of the tree nodes. They also handle effects that result from the localness of the reasoning within the decomposition tree: For example, independent decisions can yield to a deadlock. Here, strategies have either to prevent these situations or have to provide a mechanism that overcomes the problem. Other problems are desired states of the tree that are not reachable by other states and oscillating tree behavior.

An example for a strategy is the following: When splitting a pack, we have two choices: (1) Do not separate parts of the same application, *e.g.*, $\{a_1, a_2, b_1, b_2\} \rightarrow \{a_1, a_2\}, \{b_1, b_2\}$. This option can result in a linear reduction of the adaptation time. It does not increase the structural negotiation effort and does not decrease the utility of the applications. (2) Separate parts of the same application, *e.g.*, $\{a_1, a_2, b_1, b_2\} \rightarrow \{a_1, b_1\}, \{a_2, b_2\}$. This option can result in an exponential reduction of the adaptation time but might decrease the utility of the applications. One strategy is to always try the first option if possible and try the second one only with packs where all contained parts are from the same application.

7 Use Case: The InstantSocial Scenario

The following *InstantSocial* (IS) scenario [FHS08] demonstrates the capabilities of the D&C approach in organizing the adaptation in larger theatres. One general aim of adaptive middleware systems, and thus an aim of D&C, is to be transparent to the applications and to the user. Therefore, we report two views: the user view typed in normal font and *system view typed in cursive*.

Paul is visiting a large rock festival. During a Björk concert, he is not able to take a good shot, others could have done better.

An adaptation is triggered and restructure the IS decomposition tree with other discovered IS nodes using a strategy which maximizes the quality of the pictures.

Paul is willing to share his pictures with others. He instructs his PDA to look out for other visitors with the same interest. Unfortunately, the Internet connection is down and there is no immediate success.

The decomposition tree is configured with a strategy that maximizes Björk-related multimedia content. For the time being, the decomposition tree is restricted to a single node (Paul's PDA) due to the lack of connectivity.

Back at his tent, Paul listens to some music when his PDA notifies him about the presence of a media sharing group. He happily joins, gives high priority to this application, and a moment later his display shows a selection of pictures, each representing a collection of shots. He browses through the content, selects the ones he likes, and begins to download.

The PDA runs a MP3 player with high priority and IS with low. After a picture sharing community becomes available, the priorities get reversed. Thus, the negotiation group sharing resources between the MP3 player and IS is updated and the adaptation process allocates more resources to IS in order to list and download the content provided by the community. Among the adaptations performed, the media replicator and ontology components of IS are replaced by similar services provided by two other PDAs.

Suddenly, the current download aborts prematurely: One of the group members has left without prior notification. But only some of the pictures of this user disappear, others are still available.

The connection to the weak PDA is lost and the IS decomposition tree adapts with the help of the MUSIC middleware. The current download aborts, but some of the Björk pictures have been seamlessly replicated, so their availability does not change.

Some time later, Paul notices that the selection he sees becomes more precise: Some topics he does not care about are no longer shown and some others, unusual but interesting ones, appear. He checks the Internet connection and yes, the festival's Wifi network is up again.

The Internet connection is re-established. The decomposition tree adapts IS by replacing the ontology component on Paul's PDA by a much accurate one at a remote server. The selections are re-evaluated. Unnoticed by him, Paul's PDA now hosts a media replicator component.

He decides to see the next concert and indicates his wish to leave the group. The PDA asks him to wait a few seconds. After getting the acknowledge, Paul returns to the stage for some more good music.

Some of the pictures kept on Paul's PDA are moved to the remaining media replicator. After success, Paul is notified and his community service terminates.

The potential gains of using D&C in this example are numerous. Its distributed nature allows applications to handle the frequent changes in the theatre gracefully: First, Paul's InstantSocial application cannot provide useful service because it is alone. But after other InstantSocial nodes become available, the social group is set up automatically. Arbitrary, unanticipated changes alter the offered functionality but keep the group operational. A centralized control mechanism would fail in such situations.

The use of packs allows InstantSocial to make full use of the available resources by controlling which application parts are to be adapted in combination and which are not: Some part of the weak PDA's InstantSocial application is hosted by Paul's PDA but they run with little interference.

The fine-grained control over the priority of applications allows Paul to search for a picture sharing group while listening to high quality music. When the group becomes available, the music is played with less quality. But Paul is distracted anyway, so he does not realize it.

8 Related Work

The Greedy approach [BHRE07] is another way of improving the adaptation organization. It adapts applications one by one, beginning with the one that offers highest expected utility. Each application is given the remaining available resources until they are used up. The Greedy approach reduces the overall adaptation time by adapting individual applications and has the potential of yielding a high overall utility.

DACAR [DM07] uses rule-based policies to monitor an environment, to deploy applications, and to react to changes in the environment. The use of generic rules allows the developer to formulate fine-grained policies that allow to reason about and verify the rule base. Nevertheless, the control mechanism of DACAR requires an entity with complete knowledge about the environment and the applications, which poses an error-prone bottleneck in dynamic theatres. In contrast, D&C builds on distributed, incomplete knowledge that is more suitable in this case.

SAFRAN [DL06] is a framework for building self-adaptive, component-based applications that separates the application logic from the adaptation. It is very high level and, in principle, allows

for the implementation of techniques similar to the distribution tree. Although SAFRAN supports distributed adaptation by allowing each component to decide upon which reconfiguration to operate, it does not support the coordination of adaptations that are carried out and can lead to unstable behavior, in certain cases.

In [BT08], authors introduce a distributed architecture for coordinating autonomous agents. The proposed approach defines *supervisors* as coordinating entities for *clusters* of autonomous agents. Supervisors can interact to aggregate and analyze context data retrieved by agents. Each supervisor is responsible for implementing system-wide adaptations on agents associated to its cluster. According to authors, the clusters can be dynamically created and updated using dedicated techniques [EDN07]. If, similarly to D&C, this approach tackles the coordination of large theatres, the proposed decomposition is rather static and does not support application driven organization of the topology.

According to [MK], D&C is a combination of meta-level control-based planning and social law-based design: Applications adhere to the distribution of resources provided by the negotiation nodes because of social laws. It is also control-based planning because each node in the decomposition tree "is guided by high level strategic information for cooperation". Minsky *et. al.* [MU00] develop principles of law-governed interactions, of which many hold for the D&C approach, too. The main difference to D&C is that they assume independent agents with their own priorities—*i.e.*, whose decisions have to be controlled whether they are within the law—while in D&C the agents—*i.e.*, the nodes—follow the law by design.

9 Conclusions and Future Work

This work proposes the Divide and Conquer (D&C) approach for organizing the adaptation of a theatre—*i.e.*, of a number of large, distributed applications in mobile environments with frequent context changes. This organization is independent of the application logic and relieves the application developer from providing the organization himself.

D&C considers packs—*i.e.*, collections of parts of applications—and thus gives the middleware a more fine-grained control over the adaptation than what is achievable by operating with full applications. By dividing the overall task of adapting the theatre into the tasks of adapting individual packs, D&C allows the adaptation middleware to parallelize the required work.

By allowing the user to assign priority to applications, D&C enables to balance the perceived quality of service according to her/his needs. The realization of this balance is independent of the application logic and does not require provisions by the developer.

D&C uses distributed reasoning activities to decide upon and to change the division, as well as to react to expected and unexpected changes in the theatre. This approach yields a more decentralized and flexible organization of the adaptation as achievable by centralized reasoning.

Although the work on D&C is ongoing and the algorithms and heuristics of D&C are not rigorously validated, we believe that they will overcome the shortcomings of a global adaptation approach. We currently develop strategies that tell how and when the different options of organizing the adaptation should be applied. We plan to investigate if and how the results of other rule-based approaches to distributed adaptation could be applied in a D&C setting, *e.g.*, the rules of DACAR and SAFRAN. Currently, we are developing a simulator that allows to investigate and compare different strategies.

Acknowledgements: The work is funded by the Klaus Tschira Foundation and by the European Commission for the MUSIC project (# 035166). The authors would like to thank Yun Ding, Frank Eliassen, Gunnar Brataas, Eli Gjørven, and Bernd Rapp for their helpful comments.

References

[BHRE07] G. Brataas, S. Hallsteinsen, R. Rouvoy, F. Eliassen. Scalability of Decision Models for Dynamic Product Lines. In *Proceedings of the International Workshop on Dynamic Software Produc Line*. Sept. 2007.

[BT08] L. Baresi, G. Tamburrelli. Loose Compositions for Autonomic Systems. In *7th International Symposium on Software Composition (SC)*. LNCS 4954, pp. 165–172. Springer, Budapest, Hungary, Mar. 2008.

[DL06] P.-C. David, T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *5th International Symposium on Software Composition*. LNCS 4089, pp. 82–97. Springer, 2006.

[DM07] J. Dubus, P. Merle. Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-Based Systems. In Kühne (ed.), *International MoDELS Workshop on Models @ Runtime (MRT'06)*. LNCS 4364, pp. 242–251. Springer, 2007.

[EDN07] R. M. Elisabetta Di Nitto, Daniel Dubois. Self-Aggregation Algorithms for Autonomic Systems. In *2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems (BIONETICS)*. Budapest, Hungary, Dec. 2007.

[FHS+06] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, E. Gjørven. Using Architecture Models for Runtime Adaptability. *IEEE Software* 23(2):62–70, Mar./Apr. 2006.

[FHS08] L. Fraga, S. Hallsteinsen, U. Scholz. "Instant Social" – Implementing a Distributed Mobile Multi-user Application with Adaptation Middleware. In 1st *DisCoTec Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS)*. EASST, this volume. 2008.

[KRG07] M. U. Khan, R. Reichle, K. Geihs. Applying Architectural Constraints in the Modeling of Self-adaptive Component-based Applications. In *ECOOP Workshop on Model Driven Software Adaptation (M-ADAPT)*. Berlin, Germany, July/Aug. 2007.

[MAD06] MADAM IST. Theory of Adaptation. Deliverable D2.2 of the project MADAM: Mobility and adaptation enabling middleware, Dec. 2006.

[MK] A. Mali, S. Kambhampati. Distributed Planning. unpublished.

[MU00] N. Minsky, V. Ungureanu. Law-governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems. *TOSEM* 9(3):273–305, 2000.

[MUS] IST MUSIC project. www.ist-music.eu.