Distributed Interaction for Activity-Based Computing

Jonathan Bunde-Pedersen

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Distributed Interaction for Activity-Based Computing

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Jonathan Bunde-Pedersen
January 27, 2009

Abstract

As we engage in an increasing number of activities involving computational devices distributed in our environment, organizing and managing these activities becomes harder. This dissertation presents an approach to activity-based computing focusing on the distributed interaction between software systems and the individuals using them. The work involves three topics: (i) user-interfaces and interaction techniques for working in, and with, activities, (ii) infrastructures responsible for the runtime distribution of data and events, and (iii) mechanisms to enable application developers to expose their applications to activity-based integration.

These three topics are treated each in a chapter in Part I of this dissertation. The first involves how to present activities in user-interfaces such that single individuals can better manage and interact with their activities, but also how groups of individuals are aided in their collaboration around one or more computational activities. The second topic, the software infrastructure, deals with the underlying technical issues of distributing and managing the runtime state of activities, and which type of software architecture to base a well-performing, but also flexible and extensible infrastructure on. The last topic revolves around how to enable external applications to be integrated into activities, and which handles for doing so are provided to developers of these applications.

The contributions are within the field of activity-based computing and includes both designs and implementations of prototypical, but nevertheless complete systems which interoperate to form distributed applications, enabling individuals to use the activity-based computing concepts to support e.g. interrupted work and synchronous as well as asynchronous collaboration.

Acknowledgements

I owe thanks to a lot of people which have supported and encouraged me during my period of study, and who in some way or another have contributed to the creation of this dissertation.

First and foremost, my advisor, Jakob Bardram, for his insightful guidance and his committed involvement in my work. In addition I would like to thank Henrik Bærbak Christensen as well as my colleagues and friends; Martin Mogensen, Thomas Riisgaard Hansen, and Mikkel Baun Kjærgaard for great discussions and collaborations. And to the Pervasive Healthcare group at Aarhus University and all past and present participants in the Activity-Based Computing project.

Thanks also goes to Keith Edwards and members of his pixi lab for making my visit at Georgia Institute of Technology a really great experience. The same gratitude goes to Jeffrey Pierce and John Barton and their colleagues at IBM Research Almaden for letting me work on some really interesting and fun projects there.

Lastly I especially want to thank Astrid for her invaluable support and help, love and encouragement. And to Birgit, Erik and Sophie and the rest of my family for their support.

Jonathan Bunde-Pedersen,
Aarhus, January 27, 2009.

Contents

Part I

Overview

Chapter 1

Introduction

1.1 Motivation

As we surround ourselves with more and more computers, the overhead in managing the work and the tasks we do with these machines increases dramatically. Resuming a previously interrupted task on a computer, for instance, can be arduous – finding the files, applications and other resources involved in this task must be done manually. The effort involved in this process is only exacerbated when we try to resume the task at a different computer than where it originally began. Files must be transferred and suitable applications started to handle these files. A possible solution to this problem is to add a layer which abstracts away the handling of individual files and applications, and which may also add functionality to leverage other issues of managing the tasks of users.

Activity-based computing is such an approach. It seeks to provide a further abstraction level on applications and the data which they access. The primary concept which is used to this effect is the *activity*. In activity-based computing, there is a distinction between human and computational activities. *Human activities* are the goal-oriented tasks which individuals seek to accomplish in the physical world. These are then modelled as *computational entities* in software and made available for end-users and developers of activity-enabled applications. The computational activities may span several applications and require disparate data-sources, and they may be engaged in by one or even several people at any one time. The concept of an activity is thus something which pervades activity-based systems, it is made available from the user-interface to the infrastructure as a computational concept representing a human activity. The overall goal of activity-based computing is to provide more and better support for work characterized by frequent interruptions, a high degree of mobility, and work which is inherently collaborative. Examples of such work are found in many and diverse environments, from office work to work such as that found in hospitals or nursing homes.

Building systems which provide this kind of computational support for human

activities is a complex task on multiple levels and involves both the layers of presentation and infrastructure. For instance, there is the question of how to model the activities themselves: Which kind of structures best support the notion of a human activity, and which capabilities is model this instrumented with? There is the user-interface, the interaction with the activities, and their presentation to users. The complexity increases further if we take an approach where activities may span several devices and may be composed of, or interact with, disparate data-sources or applications. Or, where multiple – distributed or co-located – individuals collaborate and interact within an activity.

This dissertation revolves around the concept of "distributed interaction" and has as its main theme the question of how to realize distributed interaction in an activity-based computing system. The concept of distributed interaction is concerned with the interaction between (i) computational resources; applications, files and activities – their distribution across devices; between (ii) individuals and resources; and between (iii) collaborating individuals. The software layers involved in realizing these three levels of interaction are *presentation*, *infrastructure*, and a layer for *integrating external applications*. The following section places this dissertation within a research context by giving a brief overview of related work and related projects, before the research questions of this dissertation, each concerned with a separate layer, are presented.

1.2 Research Context

This research falls within the field of activity-based computing, which in itself is a compound research field building on pervasive or ubiquitous computing, computer-supported coorporative work and distributed computing. Within this field a number of related systems for supporting human activities exist. The research which form the context for this dissertation, and which the notion of distributed interaction spans, can roughly be divided into three areas. The first is on activity-based computing in general, and deals with the concepts of *activities* or *tasks* and how they are used as a tool for e.g. application- and information-integration and collaboration. The second area is the representation of activities in the user-interface. Here activities are often seen as an abstraction on top of existing applications and data, and this is used to e.g. provide an overview of the applications involved in the activity or to support interruptions. The third area is that of infrastructure support for activities in distributed environments. It is also here we find the support for integrating applications with the infrastructure and mechanisms for building applications to use with activity-based systems.

When looking at related work, which deals with representing the *concept* of human activities in a computational form, the primary reference in the context of this dissertation is version 3 of the ABC project by Bardram [17]. Here activities are presented in a fullscreen Java MDI environment, in which custom Java windows represent applications. Each application is built using custom controls, on which the system can get and set the internal state (e.g. the scroll-position for a scroll-

bar). The activity here is then the combined state of all windows. The Aura project by Garlan et al. [75] explores a "task"-concept which is similar to an activity, and which provides a framework for creating applications for pervasive computing environments. In the Unified Activity Management by Moran [124] and Moody et al. [123] an activity is an entity which is "represented as an association of properties and as relationships to other entities" and backed by an infrastructure. It serves as a unifying concept by including many disparate computational items such as webpages, todo-notes and calendar entries. The work of Muller et al. [125] and Vogel et al. [177] describes the concept of an activity as the mediator of collaboration between users and presents the Activity Explorer prototype which gives access to the contents of activities. Similarly, the TaskMaster system by Bellotti et al. [30] organizes and presents email-contents and -attachments in task-centric collections. Common to these research projects is that they are full-scale activity-based systems whose objectives are to provide computational support for human activities beyond those found in individual applications.

Activities as they are presented to the user in the *user-interface* is as mentioned another area of research within the broader task-centered or activity-based computing research fields. Treating activities as first class objects and making them visible for user-interaction is the focus of the GroupBar system by Smith et al. [163], in which applications can be grouped into something alike activities which may then be switched between like "virtual desktops" [137]. Other projects can be seen to support human activities in the user-interface as an abstraction on top of existing files and applications by relying on spatial or temporal memory as done by e.g. moving groups of windows to the desktop periphery as done by Robertson et al. [145] in the Scalable Fabric system, extending the user's desktop with additional virtual screen space [24] or peripheral displays [109]. The same is accomplished using 3D in the TaskGallery window management [143], using time as the main organizing principle in task management [139], or to allow for hierarchical window organization and elastic stretch and resize of windows in the Elastic Windows system by Kandogan and Shneiderman [96].

The research within *infrastructures* for pervasive and activity-based computing is the third area in which we find related work. The role of the infrastructure is to provide a foundation of software components, which in turn provide services extending or augmenting the functionality of applications. In the case of activity-based computing the primary role of the infrastructure is to manage and distribute activities, as well as events relating to those activities. Another important aspect of infrastructures is the role that they play in application composition, i.e. that they enable applications to take advantage of the possibly distributed nature of the environment by placing their application-components or -state within the infrastructure. The Gaia project of Hess et al. [86] and Chetan et al. [46], bases its infrastructure component on distributed objects, which roam "active-spaces". Applications are collections of distributed objects which use the underlying infrastructure to provide e.g. persistency and migration of applications. The infrastructure of Aura [75] is a *meta*-operating system which services applications with low-level functions such as a distributed filesystem, a resource monitoring component, and an engine for remote execution based on an RPC mechanism. But it

also provides the higher-level support for task- or activity-management through its PRISM component which houses explicit representations of user-tasks as coalitions of abstract services. The *one.world* infrastructure by Grimm [81] focus on a nesting scheme in which applications may be composed of and execute in coordination with a collection of services. An application manager is then used to control a collection of applications, thus providing an interface in which to define activities. Similarly, the i-LAND project by Streitz et al. [168] runs on the *BEACH* infrastructure which creates shared information spaces and provides the functions necessary for distributing, synchronizing and replicating information objects which are the core entities which applications use for communication and state-sharing.

The main research themes of this dissertation and those of leveraging distributed interaction within activity-based computing are thus overall: The presentation of – and interaction with – activities in distributed environments. Presentation is realized in user-interfaces which provides overview of one or more activities, and which enables the user to e.g. switch between different activities or to maintain an awareness of the activities of others. A second theme is that of infrastructure support in the form of first; a software construction in which activities may be stored, manipulated and which handles distribution of activities and second; tools and methods for building activity-enabled applications and for integrating external applications and components. These themes form the background for formulating the research questions in this dissertation.

1.3 Research Questions

The overall research question addressed by this dissertation is: *How is the notion of distributed interaction in activity-based computing realized in the presentation of activities, their infrastructural distribution and management, as well as the tools and methods for building applications that interface with activities?* This generic research question is in the following split into three concrete research questions.

Question I Which interaction mechanisms are needed in a distributed activity-based system(?), and how are activities presented for end-users? These questions has as their focus the appearance and the functionality of user-interfaces through which activities are manipulated and made visible for the end-user. A further aspect of this question, is how the distribution of activities is handled by multiple user-interfaces, how this affords participants an overview of one or more activities, and enables them to collaborate.

Question II What are the (software) infrastructural components of supporting distributed interaction in an activity-based system? How are activities modelled and what kind of architecture accomplishes this modelling best? This question deals with the importance of providing the specific functionality needed to distribute information in a given environment – such as those found in wearable computing or those populated by multiple-displays. The perspective of this dissertation is on how to 'encode' the requirements of the

environment into the system, to ensure that the software is made adaptable, reflecting the environment in which it runs.

Question III How are external applications activity-enabled? This question revolves around the need to put application developers in a position where they are able to integrate their applications into an activity-based system and take advantage of the capabilities given to them through activity-based computing. This includes functionality which enables the internal state of applications to be shared through activities hosted in a distributed infrastructure.

The three research question all revolve around the concept of distributed interaction, but each represent a separate aspect of this topic. Answering the three questions involves an approach which deals with multiple fields of research, but an approach which can be embedded in the construction of one or more prototype implementations. The objective of the work presented in this dissertation is therefore to address these three questions by building and evaluating prototypes demonstrating possible solutions for one or more of the questions. The evaluations but also the prototypes themselves will be the eventual test of and evidence backing this objective. The research approach taken to accomplish this is described next. It is aimed at describing the scientific framework which provides a means of – and methods for – finding these solutions.

1.4 Research Methodology & Approach

The classical research methodologies of the natural sciences are according to March and Smith [113] aimed at understanding reality, at explaining "why" and "how" things are. It is concerned with developing concepts or specialized languages to characterize phenomena in order to first *theorize*, and then later *justify* the theories through observation and experimentation. The majority of computer science or engineering research does not easily lend itself exclusively to this classical methodology, but rather falls within the paradigm of *design science*. This is also the case for the work presented in this dissertation.

Design Science

Design science "attempts to create things that serve human purposes. It is technology-oriented. Its products are assessed against criteria of value or utility – does it work? is it an improvement"[113, pg. 253]. The classical natural science research approach can be seen as *analytical* where design science is *experimental* in nature. Design science as a research framework is driven by the distinction between *research outputs* and *research activities*. The activities; build, evaluate, theorize, and justify produces outputs of the following types: constructs, models, methods, and instantiations. Constructs are the concepts used to describe problems, the vocabulary of a specific domain. Models express relationships among constructs, they represent situations as problem- and solution-statements. Methods are sets of steps required

to perform a task, such as those of an algorithm. Instantiations are realizations of an artifact in its environment, an operationalization of constructs, models and methods. The build and evaluate activities are experimental in nature, and hence at the core of design-science, while the theorize and justify research activities are seen as more in terms with classical analytical research methodologies. The act of theorizing is concerned with formalizing ideas, while justifying means proving or dis-proving them. Applying this framework to the research represented by the papers in Part II of this dissertation, results in Table 1.1 below. The papers are placed in the table according to their main focus. Take for instance Paper I; the constructs presented here include among others "activity-based roaming", while the instantiation is the implementation of the software system which allows users to actually perform the roaming, and includes both the user-interface and the underlying infrastructure layers. The evaluation looks at both the interaction mechanisms and the user-interface, but also focus on the applicability of the constructs.

The analytical component in the work presented here mainly lies in the technical application of algorithms and software architectural constructions to achieve non-functional goals such as *performance* and *scalability*, while the experimental approach is visible in the construction, deployment and end-user evaluation of the prototype implementations presented in later chapters.

	Build	Evaluate	Theorize	Justify
Constructs	I, IV			
Models	II, III, VII	III	V	
Methods	II, V	I, V	V, VI	VI
Instantiations	I–III, IV, V, VII	I, III	V, VI	V

Table 1.1 Design science framework applied to the papers of Part II.

The distribution of papers in Table 1.1 reveals that the main research activity falls within the experimental design science research approach of "build and evaluate", while the more analytical natural science approach is less applicable. This can be explained from the research activities involved in this dissertation, which were based on a "prototyping" approach. A full description of research activities and outputs for each paper – a verbose version of Table 1.1 – is not included in this dissertation. However, the research outputs, the contributions, are described in more detail in Section 1.5. Moving from research methodology to approach the next section is dedicated to describing prototyping and an iterative research cycle.

Prototyping

The overall research approach applied in this dissertation follows the pattern of asking research questions, formulating hypothesis, and finally providing evidence, either confirming or rejecting the hypothesis. The evidence is given in the form of experiments. More concretely the hypothesis is often contained in the implementation of prototypes, and the evaluation of these prototypes is then the validating

experiments. An evaluation in this manner is according to Zobel [187]: "...a full test of the hypothesis, based on an implementation of the proposal and on real – or at least realistic – data" [187, pg 176]. The prototypes are for the most part vertical [72], i.e. more or less complete implementations of the functionality under development. The evaluations are technical where an analytical research approach has been taken and where the contribution is of technical nature, such as it is the case in Paper V. The major part of contributions however, falls within the realm of the experimental part of design science, and as such the evaluations here are actual, but experimental, deployments of the prototypes, e.g. Paper III, and end-user studies of these deployments.

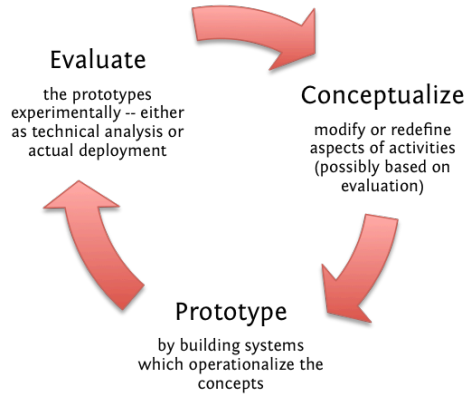


Figure 1.1 A cycle of an experimental research approach. Prototypes are operationalizations of concepts which are evaluated experimentally.

The concrete evolution and crystallization of ideas into prototypes were done using an iterative approach. The arguments in favor of prototyping are numerous [72, 78, 55], and one of the main benefits are, that the prototypes themselves may serve as vessels for the articulation of underlying concepts, models and methods [113]. As such, the descriptions of abstract concepts often take an outset in concrete implementations. The cycle depicted in Figure 1.1 represents the research activities used. The focus of each cycle may differ as the prototypes focus on different research questions. For example, in Paper I we started out with an idea of letting a desktop-based user-interface be the focal point for interacting with distributed activities. This led us to build a prototype in the Windows XP operating system in which end-users could, among other things, interact with activities, store them in a central server, and resume them on any connected device. Finally, the prototype was evaluated by doing a user-study. Another paper, Paper VII, focusing on infrastructure issues then represents another cycle of research activities. The research problems are thus treated by multiple iterations of the research activity cycle. The concrete research activities, as previously mentioned, therefore bear witness to a design science oriented approach in which the "build" and "evaluate" activities are used.

1.5 Contributions

The contributions, or in design science terms; the research outputs, are split into three main themes; Distributed user-interfaces for Activity-Based Computing, Infrastructure, and Stateful Applications. The three themes are based on the three research questions of Section 1.3 and matches the contents of chapters 3, 4, and 5. Each paper in Part II of this dissertation may represent a contribution in more than one theme, since the prototypes presented in the papers are often instantiations of more or less complete activity-based systems, and as such deal with and present solutions to more than one single research questions. Figure 1.2 is an illustration of this point, mapping the papers onto the three research questions. The following sections gives a more detailed thematic overview of each included paper.

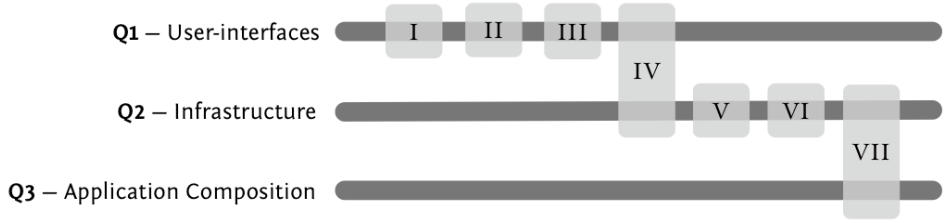


Figure 1.2 The contributions of each paper mapped onto the overall themes and research questions.

Distributed user-interfaces for Activity-Based Computing

This theme deals with activities as they are made first level entities, visible for the end-user. The main effort is represented in Paper I in which an implementation and evaluation of an a user-interface extending the Windows desktop is presented. The overall contributions are in the form of a new user interface for activity-based computing mimicking that of the operating system and mechanisms for using activities on the world-wide-web. Two other prototypes are presented as well; the first introduces how interaction may be distributed in a wearable computing prototype, while the second introduces the concept of a distributed multiple-display environment supported by activities.

- In Paper I we show how to support activities as part of a desktop-based interface. The work represents both an effort to provide constructs such as e.g. the notion of *activity-roaming*, and instantiations of these constructs. The prototype itself represents and instantiation where the standard desktop interface is augmented with an activity-list from which activities can be resumed and suspended, and a spatial layout mechanism designed to provide an overview of a single activity. The usefulness of activity-based computing in normal desktop work is demonstrated through an evaluation.

- Paper II is an application of activity-based computing onto the domain of wearable computing. The system illustrates how interaction within one activity may be distributed on multiple displays, and how uneven collaboration is realized using distributed activities.
- Paper III presents the CCAB system for distributed multi-display environments. It is a system in which activities are presented in a fullscreen mode, and where each display is continuously updated to match its current context. Displays can be both fixed and mobile and configured to present relevant activities for any type of context, e.g. a given user or a given location. The paper shows how many activities may be presented simultaneously in an environment with multiple displays.
- Paper IV presents the activity-links construct which is basically a URL scheme for referring to activities. This allows activities to be stored online and enable links to activities to be included in e.g. web-pages and emails. It also represents an instantiation of a further integration between local desktop applications and activity-based web applications.

Infrastructure

The role of the infrastructure components presented in this dissertation, is primarily to provide the means for modelling, distributing, and managing the runtime state of activities on multiple devices. An important property of the infrastructure is that it coordinates the access to activities and provides functionality for applications to listen for changes in activities on the local system and in the surrounding environment. The infrastructure also serves as the medium in which activities may be integrated with external context, for instance information describing who is currently engaged in collaboration through the activity, or the locations where the activity is resumed – in other words it is where the integration of disparate data-sources into activities happens. The papers describe the architecture of the infrastructures as well as the protocols which enable their distribution.

- Paper V focus on a framework for using activities as a collaboration mechanism. It presents a hybrid extension of the client-server architecture employing a novel differentiation mechanism to distinguish between events and route them using appropriate distribution mechanisms. The distinction is made between the constructs of ephemeral and accountable events, and justified through a technical evaluation.
- Paper VI presents a fusion of an ad-hoc peer-to-peer and a centralized architecture and demonstrates its use through a scenario involving emergency workers. The research outputs are a method for switching configuration based on context, its justification through a scenario, and an implementation of the adaptive fusion architecture.
- The focus of Paper VII is on providing a lightweight *meta*-infrastructure designed for activity-based and pervasive environments. The paper present the

implementation of the infrastructure component which is used to drive an activity-based multi-display environment as well as pervasive meeting space, and thus demonstrates a flexible infrastructure approach to enable distributed interaction in a variety of environments.

Stateful Applications

An activity may span several external applications. These applications are joined in the activity by sharing a certain amount of their internal state; we talk of applications as being activity-enabled. The purpose is to make applications aware of the activity in whose context they execute, and to provide them with means to interact with the activity itself as well as other applications that are part of the same activity. The contribution in this area is a method for application developers to annotate specific parts of their application to denote the "state" of their application and have it shared and distributed through an infrastructure.

- The ASPI programming model for application composition is introduced in Paper VII. This model allows developers options for synchronizing state and for modelling complex data-structures using an annotation scheme. The technical details of how the programming model fits into the infrastructure and the handles which are made available to the developer, are also presented here, along with a prototypical example of its use.

1.6 Dissertation Overview

This dissertation is organized into two main parts. The rest of Part I is an overview of my work. Chapter 2 gives a conceptual foundation focusing on the notions of "activity-based computing" and "distributed interaction". The next three chapters each in turn describe the components of a system supporting distributed interaction in activity-based computing, and as such, each chapter deal with one of the research questions posed earlier. Chapter 3 is concerned with research question I; how end-user interaction with activities is accomplished, as well as the presentation of activities. Chapter 4 focus on research question II; the infrastructure components for distributing activities and events within activities. Chapter 5 is on research question II; mechanisms for integrating external applications, both on the programing model made available to application developers, and on the technical implementation of this integration. Related work is described in turn in each of these three chapters, such that each chapter is responsible for the related work within its own topic. Part I is concluded in chapter 6. Part II contains a collection of papers numbered I-VII and referenced as such. Paper I, II, IV, V, and VI have been published, while III and VII are under submission. Lastly, a full list of all my published papers is included.

Chapter 2

Conceptual Foundation

The purpose of this chapter is to introduce the concepts behind *distributed interaction for activity-based computing*. It is divided into two sections, the first focuses briefly on the theoretical and practical constructs important to activity-based computing, while the second section is on three connotations of distributed interaction and the components needed to realize them.

2.1 Activity-Based Computing

Activity-based computing has its theoretical origins in activity-theory[180, 13, 11]. Activity-theory considers the relationship between human subjects and their goals or objects as mediated by cultural means, tools and signs [180]. An activity is undertaken by a human agent, the *subject*, who is motivated toward the solution of a problem or purpose, the *object*, and mediated by tools (*artifacts*) in collaboration with others, i.e. a *community*. The structure of the activity is constrained by cultural factors including conventions (*rules*) and social strata (*division of labor*). These relationships are depicted in Figure 2.1.

The mediation is the central concept in activity-theory, and especially the mediation of computational work by computational *artifacts* is important for activity-based computing. In fact, the artifact not only makes interaction possible, but also enhances it – the more advanced tools we have, the more advanced interaction is made possible [66]. Computational activities can be seen as constructs in which a number of the relationships of Figure 2.1 can be modelled. For instance, the relationship between the subject and the community can be expressed within a computational activity as relationships between the participants and their roles within the activity.

The theory of distributed cognition by Hutchins [90] is similar to activity-theory in the fact that it deals with the mental models of tasks or activities. A *distributed system* here is similar to the concept of an activity, and distributed cognition

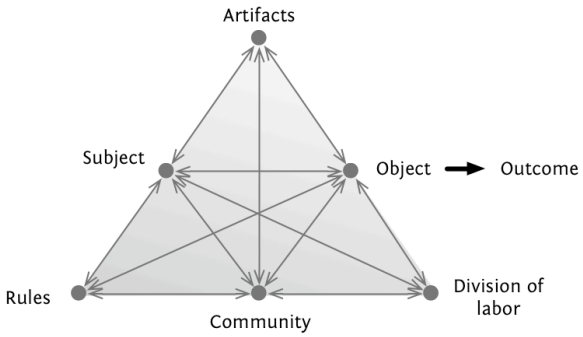


Figure 2.1 The relationships and mediations of activity-theory. After Engeström [66], pg. 78.

is concerned with the internal (in the mind) as well external (presented through artifacts) representations of cognitive structures and processes. As in activity-theory, it "extends the reach of what is considered cognitive beyond the individual to encompass interactions between people and with resources and materials in the environment" [88]. As the term suggests there is a focus on the distribution itself, the coordination among individuals and artifacts to understand "how individual agents align and share within a distributed process" [90].

An alternative to these cognitive theories is situated action proposed by Suchman [169]. She argues that activities and their objectives are highly dependent on the given situation – that the situation and the context subsumes any pre-defined plans or objectives. The point here is that people's actions are influenced, if not determined, by the context of their present situation. In the words of Suchman [169]:

> *...every course of action is highly dependent upon its material and social circumstances focusing on moment-by-moment interactions between actors, and between actors and the environments of their action.*

This suggests that the tools which mediate the interaction between individuals and objects need to be flexible, and that situational knowledge and adaptation to context plays a significant role.

It is within these theoretical frameworks that activity-based computing as a research area emerged. The work presented here is influenced by these theories but does not represent a rigid implementation of the ideas and of the mechanisms described by them. The computational construct of an activity rather represents an abstraction designed to provide more support for human (work) activities such as those found in e.g. hospitals.

Computational Activities

The idea of providing better support for human activities is at the core of activity-based computing. The activity is a mechanism with which focus can be moved

from the details of finding and executing applications and data, supporting *interruptions* in human activities. Instead activities then become the focus when switching between e.g. work tasks. Activities are a way to accomplish *mobility*. They can be moved from device to device and will maintain their structure while doing so. Activities are inherently *collaborative*, and can be shared between multiple participants. These principles are the fundamental concepts behind activity-based computing [17], as summarized here:

Activity-Based Human activities are supported through the concept of a computational activity, which collects a range of services and data into a coherent set. The activity is among other things a method of abstracting away the interaction with files and the instantiation of applications. The modelling and presentation of computational activities to individuals requires software which "overlays" this information on existing applications and files. The concepts from *activity-theory* forms the foundation for modelling this interaction, and the concrete technical and design-oriented effort towards supporting it is the focus of chapter 3.

Suspend and Resume The action of *resuming* an activity starts the process of returning the machine to the state, when the activity was last *suspended*. This typically involves somehow instantiating the activity (or the individual components or services it contains). We then say that the activity is "resumed", "executing" or "running" (these three terms are used interchangeably in this dissertation). The end-user is then able to alternate between multiple activities by suspending the current and resuming another. The suspend and resume operations support the interruptions of human activities, and are, as such, based on the notion of *situated action*, allowing individuals to react to changes in their environment.

Roaming An activity is stored and modelled in an infrastructure. It can thus through the infrastructure be distributed to machines that either resume the activity or otherwise manipulate it. The act of suspending an activity on one machine, transferring it to another, and resuming it here is the act of roaming. This allows individuals to use multiple devices, and as such supports mobility. Both *activity-theory* and *situated actions* emphasizes location as well as context.

Sharing Multiple individuals may be given access to the same activity. This activity is then said to be *shared* among the collaborating participants. The participants of an activity can then either resume it and continue the work of another previous user, or if other participants are simultaneously running the activity, changes may be synchronized on all participating devices. This supports collaboration through activities. Of the theoretical approaches above *activity-theory* and *distributed cognition* both deal with collaboration and sharing of responsibilities.

Both roaming and sharing requires an infrastructure or some method of transferring activities and runtime changes with activities between machines. The un-

derlying infrastructure for doing so is within chapter 4 while the interaction handles for end-users are found in chapter 3. Activities, as we have chosen to represent them computationally in the systems presented in this dissertation, are abstractions on top of services and data (or applications and files). Figure 2.2 is an illustration of this. The right-hand side is an example activity. The human activity is here the treatment of a patient, the concrete services, the applications, are an x-ray viewer and an application for accessing test-results. The data which these applications in turn access are for the former; x-ray images associated with the treatment and for the latter; the most recent blood-test results.

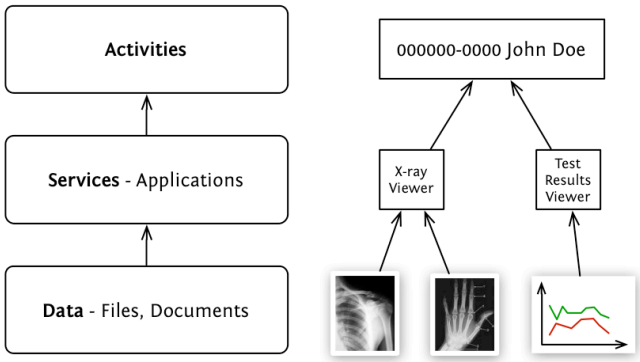


Figure 2.2 The hierarchy of a computational activity. Right; the abstractions used when modelling an activity. Left; an example activity.

This computational model af a human activity is used to realize distributed interaction. It binds applications together and allows them to use the state of the activity as a means of communication, and is also presented to user in user-interfaces which additionally provide handles for manipulating the activity. The activity is therefore a means of realizing interaction distributed between applications and users.

2.2 Distributed Interaction for Activity-Based Computing

Three connotations of distributed interaction within activity-based computing are pursued here. The first is the interaction which takes place between distributed processes. This interaction happens through protocols and is mediated by the infrastructure. The second is the interaction between an individual and one or more computational processes mediated by the user-interfaces of those processes. Here, the concept of an activity is used to provide multiple perspectives on applications and data. The third is the interaction between multiple individuals – mediated by user-interfaces and distributed processes. This interaction happens in the collaboration between individuals, and is accomplished by transferring state information between distributed applications. A generic definition of distributed interaction

is therefore that; *distributed interaction is the actions that entities, individuals or computational resources, perform through interfaces on distributed data.*

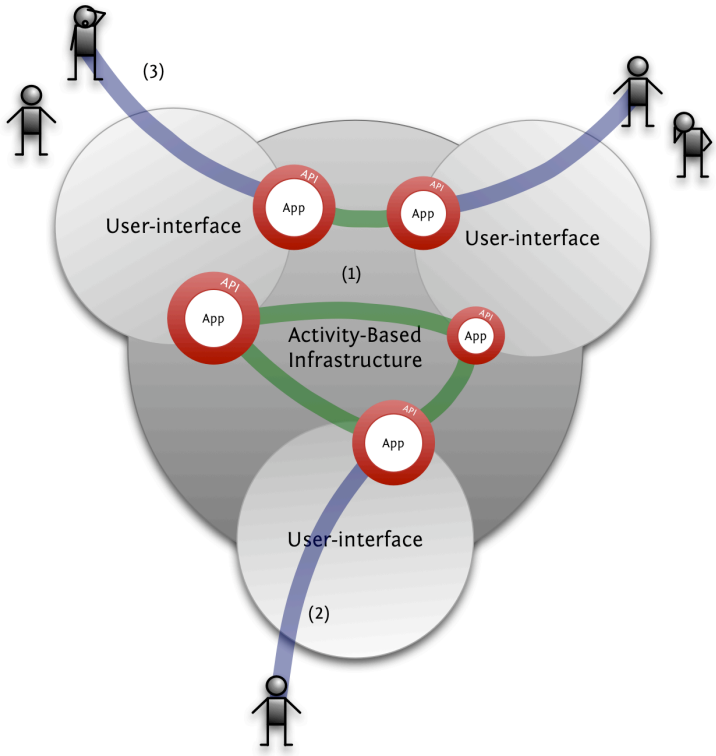


Figure 2.3 The components in distributed interaction. Interprocess interaction (1), interaction with processes or applications through user-interfaces (2), and interaction between users (3) are mediated by the these components. The colors indicate the software elements in play; (blue) for the user-interface technologies of chapter 3, (green) for the interaction through a software infrastructure as described in chapter 4, and (red) for application integration and composition mechanisms of chapter 5.

The software components needed to realize these notions of interaction are directly related to the research questions of Section 1.3. Figure 2.3 illustrates the interplay between the components and how distributed interaction is mediated through them. Activities are here represented in the infrastructure as the connections between applications marked in (green). In the following a brief discussion of the three connotations is presented.

Interprocess Interaction

This interaction happens either directly between processes or, as is the case here, is mediated by a software infrastructure layer. The interaction processes here are either the applications which individuals manipulate or independent software components executing as part of the infrastructure. There are several established tech-

nical approaches for interprocess interaction or communication; message passing is one, shared memory is another. The approach taken in the infrastructures presented here rely on shared artifacts, the activities, and use message passing over custom protocols to keep these artifacts updated and synchronized. The interaction between processes therefore happens by state-changes – i.e. one process reacts to the change in state of another. The term "distributed interaction" is often applied to the behaviour of and communication between software components in e.g. autonomic agent-based systems [153, 155, 93]. The processes in this dissertation however are not imbued with any autonomic capabilities, but still exhibit some of the same dynamic properties. One example is the ability of applications within activities to be moved between devices. Almeida et al. [6] describes the concept of an "interaction system" for distributed applications as a system which supports the interaction between two or more system parts. An interaction system represents an abstraction on e.g. middleware- or protocol-centered infrastructures and corresponds to the concept of connectors [5, 4, 120]. Connectors are architectural entities used for connecting disparate parts of distributed systems. They abstract the low-level details of communication away and allows the developer to focus on the overall interaction of system components. Interaction systems or connectors provide the abstractions which here are represented by the infrastructure and which enables the interaction between distributed applications and components. The approaches to "connect" applications, either via interprocess communication or architecturally via e.g. connectors form the contexts for the technical solutions in chapter 4. The structure in which interaction, or state sharing, between applications in an activity-based system takes place is naturally the activities. As Figure 2.3 suggests in (red) the externalization of application state is an essential element and is covered in chapter 5.

Interaction through User-Interfaces

The activity-concept is used in all interfaces as the concept through which interaction between users and computational processes happens and through which a perspective on applications and data are provided. Four different *perspectives* are made available through the user-interfaces to an activity-based system and they cover the single application, the single activity – both represented on a finite two-dimensional surface as well as an infinite – and lastly multiple activities. The single application perspective is used to focus the interaction on one application. This perspective is a remnant of the days before the introduction of windowed applications and is what you get when you e.g. maximize an application in Windows. Here, the single application perspective is similarly used as a focusing mechanism. The single activity perspective is then the presentation of multiple windows on a single screen. In most window management systems the surface on which applications may be placed is finite, however in games where larger virtual worlds cannot fit on the screen an infinite (or at least very large) surface is used. The last perspective, that of multiple activities, presents the user with several surfaces on which applications can be placed. These four perspectives, as listed in Table 2.1, are used to provide focused as well contextual information supporting end-user interaction

with activities.

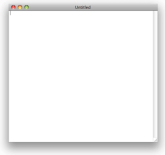
Perspective	Example	
Single application *The application fills the entire screen*	WordPerfect for DOS	
Single activity *Multiple applications are visible at the same time*	OS desktop	
Single activity – infinite space *Multiple applications are visible, and some execute outside the viewport*	Scalable fabric, Platform games	
Multiple activities *Multiple sets of applications are available to the user*	Virtual desktops	

Table 2.1 Four perspectives on activities and their representations on the desktop.

When dealing with the perspective of multiple activities the notion of concurrency becomes important. Either the activities can execute sequentially or concurrently. Sequential execution can be accomplished using the 'suspend' and 'resume' operations to switch between activities. Concurrent execution is accomplished by interleaving the execution of two or more activities on one machine. However, when we enter distribution into the equation, a number of further considerations must be taken. Two such considerations are those of synchronicity and fragmentation. Synchronicity here, is looking at whole activities and the choice between executing an activity synchronously on multiple devices, or asynchronous execution where an activity may migrate from device to device. Fragmentation on the other hand is concerned with the inner structure of an activity and the choice between having an activity execute in its entirety on one machine or having it run as fragments spanning multiple devices. These concepts are applied in the prototypes of chapter 3.

Interaction Between Users

Collaboration between multiple participants in an activity is defined as the actions that take place when interaction within one shared activity is distributed between multiple individuals. The interaction happens, as the case with a single user, through one or more user-interfaces but the software infrastructure also plays a central role as the distribution mechanism for the interaction as illustrated by the involvement of a (green) connection in Figure 2.3(3). This distribution may be subjected to non-functional requirements such as concurrency and access control which is also the responsibility of the software infrastructure. The architectural properties of the infrastructure determine how the actual distribution of interaction happens – as well as which distribution and collaboration mechanisms are made available for developers of applications. These aspects are covered in chapter 4 and 5.

The distribution of user-interfaces when individuals collaborate asynchronously or synchronously within an activity is also an important issue as shown with (blue) in (3). The user-interface is an important factor in how responsibilities are divided between participants, and so are the tools which are made available allowing them to experience the actions of others and to help them decide on their own actions. This end-user "distribution" of responsibility is realized by the user-interfaces for collaboration in chapter 3.

Chapter 3

Distributed User-Interfaces for Activity-Management

This chapter is concerned with the interaction on distributed user-interfaces for activity-based computing. It describes user-interfaces for managing activities in three different domains. The first is desktop-based and aimed work characterized by frequent interruptions and a high degree of mobility. It integrates existing applications on the desktop into activities and provides mechanisms for local interaction with activities and for distributed interaction to support collaboration. The second domain is that of wearable computing. Here two user-interfaces provide distributed interaction between an emergency field-worker and a central call-center using shared activities. The third domain is a hospital, where a generic interface for browsing multiple concurrent and distributed activities are presented.

The concepts from chapter 2 and especially those found in Section 2.2 under Interaction through User-Interfaces are used in this chapter. The different organizational perspectives on activities are used to e.g. provide overviews of multiple work tasks and in collaborative scenarios. The notion of synchronous and asynchronous execution of multiple activities are also used to describe the distribution of interaction across user-interfaces, and how the user-interfaces presents multiple concurrently executing activities.

3.1 Interaction in Desktop Environments

The desktop user-interface found in most modern operating systems is a user-interface based on a spatial layout [112]. It provides a two-dimensional surface on which files may be placed and in which applications may have their windows hosted. The surface itself is often of finite size, but the operating itself may provide multiple desktops (Spaces in OSX, virtual desktops in Windows) on which applications can then be distributed.

This section presents a system with a user-interface for interacting with dis-

tributed activities based on the desktop-metaphor. Activities may be distributed using a fixed infrastructure component – a custom-built server, via the internet using standard HTTP servers or manually by having users save the activity locally and distribute it via e.g. email. The system runs on and extends the desktop of Windows XP by providing an activity-bar similar to the built-in task-bar, and provides an infinite surface for spatial organization of applications represented in an activity. Two mechanisms are provided to navigate this infinite space; the radar-view and the zoom functionality. The activity-bar does not replace, but rather augments the built-in task-bar and displays activities instead of single applications. The bar provides a simple interface for switching between a limited number of activities. The activities can be stored remotely in a software infrastructure component which also takes care of synchronizing activities and distributes collaboration events. In the terms of the perspectives of Table 2.1 this user-interface affords the user an interface to interact within a single activity on an infinite workspace. Activities may roam or migrate between devices, and may also be distributed and synchronized across multiple machines when a collaboration is started, but they are not fragmented in the sense that all applications of an activity must run on the same machine. Multiple activities cannot run concurrently either, however the system allows the user to alternate between them via the activity-bar. The following sections describes the user-interface components of this system and the interaction which can be achieved through it.

The Activity Bar

Activities are presented and interacted with through the *activity-bar* as illustrated in Figure 3.1. In order to facilitate an intuitive understanding of how the bar works, the activity-bar is deliberately designed to resemble the Windows task-bar, however activities replace the application instances of the task-bar. Since multiple activities cannot be resumed at the same time on the desktop, the activities placed on the bar is the currently active and then the most recent, but suspended, activities.

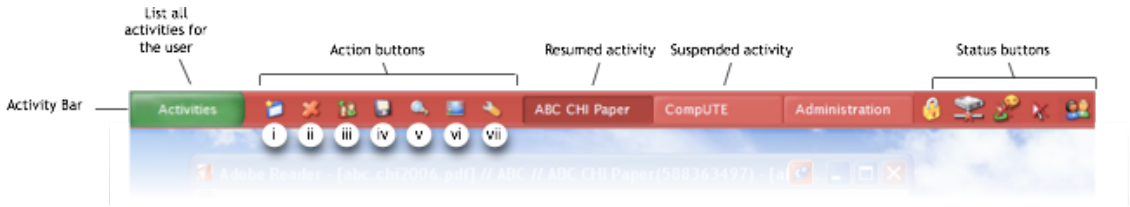


Figure 3.1 The activity-bar user-interface.

The "Activities" button is used to list the activities of the current user as shown in Figure 4. A screenshot of each activity is shown to provide a visual cue to the user. Next to this button a number of action buttons are placed, which provides functionality to; (i) create a new activity, (ii) suspend the current activity, (iii) invite other participants to join a collaboration in the current activity, (iv) save the activity locally in a file which may e.g. manually be transferred to another machine and resumed, (v) zoom the activity to get an overview of all applications within it,

(vi) to show the radar view for an overview, and finally (vii) to show a control panel with user preferences. On the left hand side, a number of status icons provide contextual clues when the user is engaged in a collaborative session. The clues include information about the participants and options for controlling access to the activity, as well as to configure the available collaboration widgets. The features for enabling collaboration are further described in the Collaboration subsection.

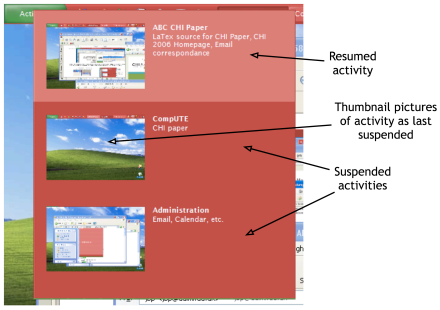


Figure 3.2 The list showing available activities.

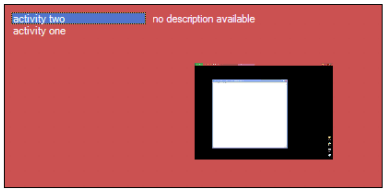


Figure 3.3 The Ctrl+Tab activity-switcher.

The system does not automatically detect or create new activities for the user, but rather lets the user explicitly state when a new activity should be created, or when to suspend or resume an existing activity. Once an activity is resumed applications may be included in it by using the "pin icon" in the application title bar as illustrated in Figure 3.4. Applications can thus be moved between activities by first removing them from the current one, suspending the activity and then resuming the new one, and subsequently adding the application to this activity. The system does not allow an instance of an application to be part of more than one activity at any given time. Activities are represented as the combined "state" of the participating applications. The state of a single activity is composed of a number of independent component states. All applications have their window-size and position automatically included. A single browser-window, for instance, may then have a state consisting of the current location (URL), and the location on the page (the current scroll value). This is enough information to be able to restore the state of the browser when an activity, in which it is included, is resumed. When an application is part of a resumed activity its state is then polled at intervals by the system.

Quickly switching between activities can be accomplished using the Ctrl+Tab keyboard combination displays the user-interface shown in Figure 3.3. The Alt+Tab combination is still active and is used to toggle between application instances within an activity. The task-bar is also still available and serves its original purpose. The activity-bar is thus not a replacement of the standard Windows XP user-interface, but rather complements it by providing interaction with activities of spanning existing applications.

When switching between activities the system queries all participating application for their state and ensures that it is persisted in the infrastructure. The infrastructure provides functions for storing and synchronizing access to activities and is the subject of chapter 4. Resuming an activity is then a matter of query-

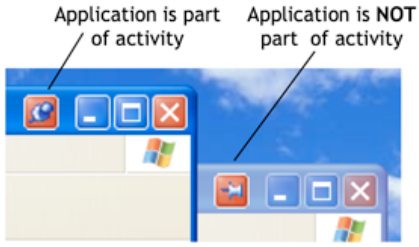


Figure 3.4 The pinning mechanism for adding and removing applications from an activity.

ing the server for the serialized activity-state again and starting applications to set their state such that it corresponds to that of the serialization. This process can take some time and depends on waiting for applications to start, so an optimization in which a fixed number of recent activities may be kept "on standby" in memory is also available. The applications are in this case hidden (but still kept running) and switching between two activities is achieved by hiding the current activity's applications and showing the ones in the activity being resumed.

Activity Links

The address-field shown in Figure 3.5 is a part of the activity-bar which enables activities to be distributed on the world-wide-web. It basically adds the functionality necessary for resuming activities stored on arbitrary web-servers. When the user enters a URL in the bar, the given server is contacted to download the serialized activity. Currently, three protocols are supported; the well-known `HTTP` and `FTP` but also the custom `ABCP` which is the protocol used natively by the infrastructure. When either a `HTTP` or a `FTP` address is specified, the activity is downloaded to the local machine and resumed from this file. This means that the downloaded activity is a personal copy which cannot be used for synchronous collaboration until it is stored in the infrastructure. `ABCP` addresses refer to servers and specifying such an address allows both synchronous and asynchronous collaboration to take place within the activity.

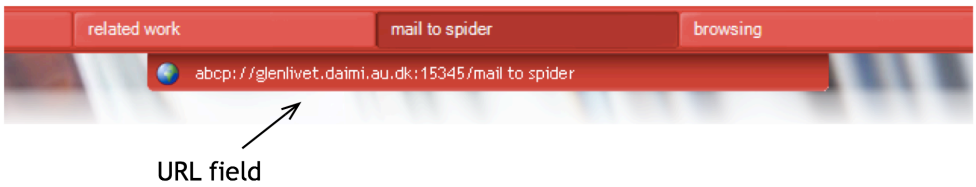


Figure 3.5 The activtity bar extended with a URL field.

The address-field is complemented with a protocol handler which is an OS extension that registers the activity-bar as a handler for `ABCP` URLs. This ensures that whenever a link prefixed with `abcp://` is activated, the activity-bar is notified and can respond by resuming the given activity. Users are thus given more options

for communicating activities. For instance; a link to an activity can be included in an email or posted on a webpage. Dedicated link management sites can also be used to e.g. index, tag and search larger collections of activities as seen in Figure 3.6 showing activity-links in the delicious [56] service.

Figure 3.6 Using del.icio.us to organize activities.

To further the integration of activities onto the web an HTTP bridge for the server was built. This bridge serves views on activities stored in the infrastructure as HTML pages which can be viewed in a browser as depicted in Figure 3.7 where a single activity is shown. The bridge can show individual activities by referring to them by name, but can also be used to filter the shown activities by using the `find?` operator as seen in Listing 3.1. Here the list of all public activities created by the "jbp" user is returned. Each activity is formatted to show a screenshot from when it was last resumed and a link to resume the activity.

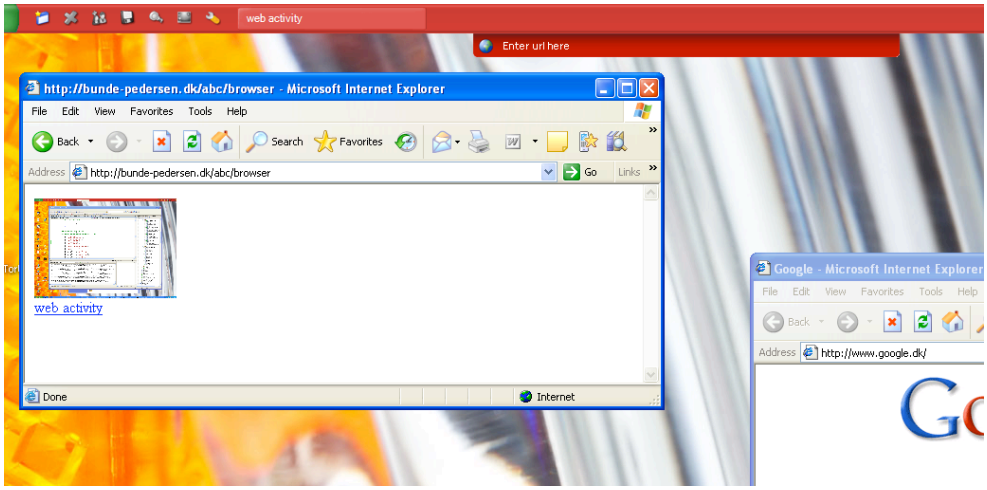


Figure 3.7 The browser in the top left corner shows the output served by the HTTP bridge.

```
1 abcp://glenlivet.daimi.au.dk/iwi2006paper
```

```
abcp://glenlivet.daimi.au.dk/find?creator=jbp&type=public
```

Listing 3.1 The URL format of activities.

Zoom & Radar

One of the mechanisms to manage the infinite workspace is the zoom functionality. This mechanism enables the user to "zoom in" on activities in order to focus on one or more applications and to "zoom out" to provide context as illustrated in Figure 3.8. The zooming functionality is also used as an adaptation tool. When an activity moves from a device with a large display to one with a smaller, some applications may be beyond the boundaries of this smaller display and thus out of reach. By zooming out the user can see all applications and choose another focal point to zoom in on a specific application. Therefore the mechanism supports the principle of activity roaming or migration between heterogenous devices.

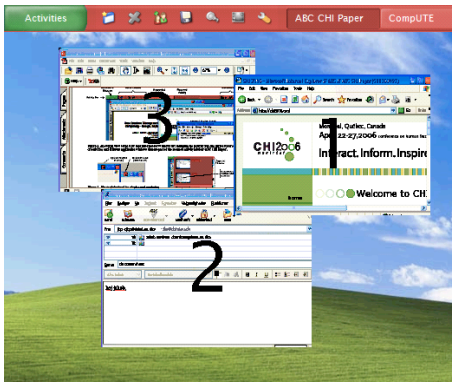


Figure 3.8 The context state of an activity – "zoomed out". The numbers are used when a speech-command should select the application on which to focus.

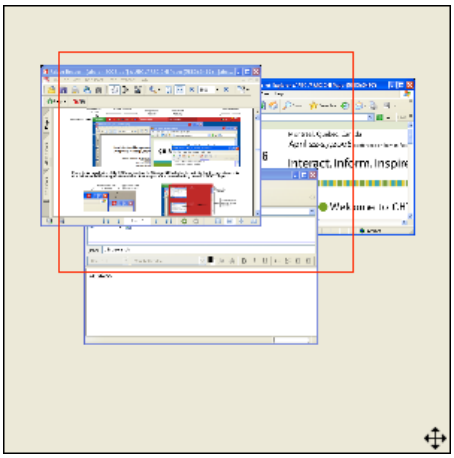


Figure 3.9 The radar view - the red square indicates the boundaries of the current view.

The shortcut Ctrl+1 can be used to toggle between the focused and context zoom states and enables the user to quickly get an overview of an activity and focus on details by clicking on the window or, if speech-interaction is enabled, by uttering the zoom command followed by the number displayed on the application. The functionality which the zoom component provides also proved itself useful in single-device activity handling as it helped users manage activities with many windows in a spatial 2d metaphor that conserve the arrangement of windows. The position of the applications can be manipulated when the activity is "zoomed out" allowing a user to rearrange the windows in the infinite surface of the activity. In contrast to other zoomable user interfaces, we maintain the window size and position, and do not automatically rearrange windows (like Exposé in Mac OS X) or introduce new layout metaphors using e.g. 3d metaphor as those by Robertson et al. [143]

Another option for navigating within the surface of a desktop-based activity, is the radar view [82] as illustrated in Figure 3.9. This radar resides transparently on top of the application windows and a red square indicates the current viewport. By dragging the red square in the radar view, the current viewport on the desktop is adjusted, moving all windows in and out of view. Individual applications can also be dragged to rearrange their individual positions. Radar views have been shown to outperform other types of overview mechanisms when the manipulation is done discreetly [154], therefore we also support simply clicking in the view to move to that location. The viewports are local, which means that other collaborating users will not be able to see when the view is changed on one machine and it also means that two participants may work in their own subspace, with their "own" applications, while still maintaining a peripheral awareness of the actions of the other participant.

Collaboration

An activity can be shared among a group of collaborating users. To this effect it includes a list of participants who can access and manipulate the activity. Consequently, all participants of an activity can resume it and continue the work of another user. This is the principle of asynchronous collaboration. Furthermore, if two or more users resume the same activity at the same time on different devices, they will be notified. If their devices support it, they will engage in an on-line, real-time desktop-based synchronous collaboration. This in effect is a synchronous distribution of interaction, where all users are presented with the same information. When two or more participants engage in the synchronous collaboration mode, the state of the activity is synchronized across all devices giving the users access to the same applications and data as depicted in Figure 3.10. This means that as one user e.g. moves a window it will be moved on all devices since the position is included in the state of an application.

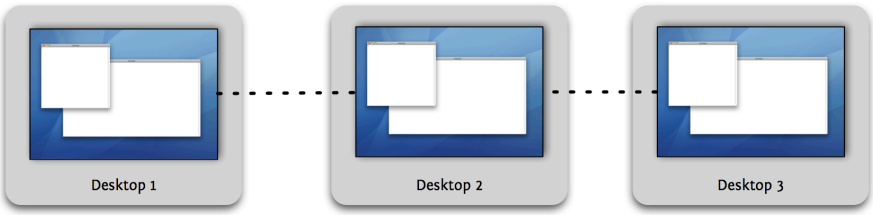


Figure 3.10 When in a synchronous collaboration the interaction is distributed equally among all participants.

The viewport, as previously mentioned, is not synchronized. This enables the zoom and radar views (depicted in Figure 3.8 and Figure 3.9) to a support peripheral spatial awareness by allowing a user to overview a greater area of desktop-space than is available to see on the screen. Both the zoom and the radar functionality only manipulate the *local* viewport as depicted in Figure 3.11, which means that two (or more) users collaborating in the same activity can partition the virtual canvas

of the activity, work in different applications and maintain a peripheral awareness of what the other users are working on. Thus, even though the interaction is synchronized, the infinite 2d-space on which the activity is placed affords an individual mode of collaboration as well.

Two collaboration widgets are also included in the system; tele-pointers and audio-links. The tele-pointers present a color-coded mouse-pointer from all participants on the local desktop, and as such also supports the notion of peripheral awareness. Audio-links are audio-channels providing an external communication channel for participants.

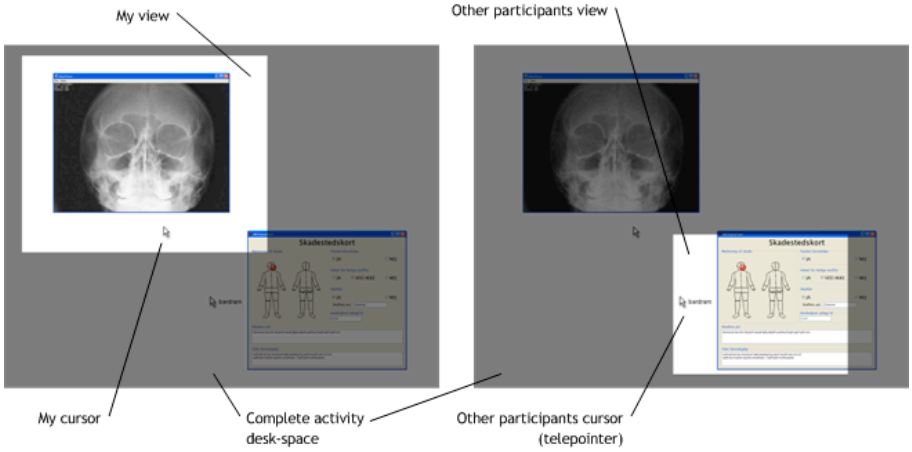


Figure 3.11 The space in an activity can be partitioned between collaborating users. Here, two users with two different viewports.

3.2 Interaction with Multiple Displays

The collaboration mechanism of the activity-bar provides the means to keep distributed activities synchronized and to enable synchronous collaboration within activities. The focus in this section is on using multiple displays to distribute interaction between several participants. The two domains to which this is applied is first that of wearable computing, and second the more heterogenous environment of a hospital in which we have several different displays of varying size and capability. The wearable system consists of two user-interfaces, the first is based in a call-center, which services a number of field workers, each equipped with a head-mounted display which runs the second user-interface. The system which is based in a hospital environment, has one flexible and generic user-interface which is used on multiple displays, some fixed and some mobile.

Wearable Computing

Wearable computing is an emerging technology which enables the use of computers and access to digital material in working situations that are physically and con-

ceptually far away from the office desk. Examples of environments in which wearable computing is often applied is police work, military warfare, maintenance work, and paramedic work. The requirements for interaction on wearable computers are therefore very different from that of desktop-based computing. One major difference is that the interaction with wearable devices must be done with as little use of the users hands as possible. Keyboards and mice are infrequently used in any traditional sense; eye-tracking [92] or speech-interfaces [146] is more the norm. The actual hardware for which the user-interface was built, was a head-mounted display with a relatively low resolution as seen in Figure 3.12. A wrist-held keyboard and a hand-held mouse was used, but the primary interface was speech. The call-center used a wall-sized rear-projected display with a very high resolution.

Figure 3.12 The head-mounted display and the two hand-held input devices.

Despite an intensifying research endavour, wearable computing still faces a number of challenges some involving basic hardware issues [71, 160]; other challenges involve more basic software architectures and operating systems for wearable computing [57]; and yet other challenges involve how users can use such computers for information retrieval, overview, data input and collaboration [26, 37, 159, 101].

The interfaces for distributed interaction for wearable computing presented here are aimed at the latter; enabling users to navigate, overview, and interact with complex information on a small wearable screen, on how to make a bridge between the user's current work activity and the working context, and on how collaboration in a wearable computing setup could be accomplished. The interplay between the two user-interfaces represent an emergency scenario in which one user, sitting at a call center monitors and guides multiple field-workers by virtue of sharing the activities of each field-worker. This scenario is illustrated in Figure 3.13 where the call-center user is simultaneously monitoring three activities of three remote paramedics.

The user-interface for the wearable computers covers the single application and single activity perspectives and is designed to consume very limited screen

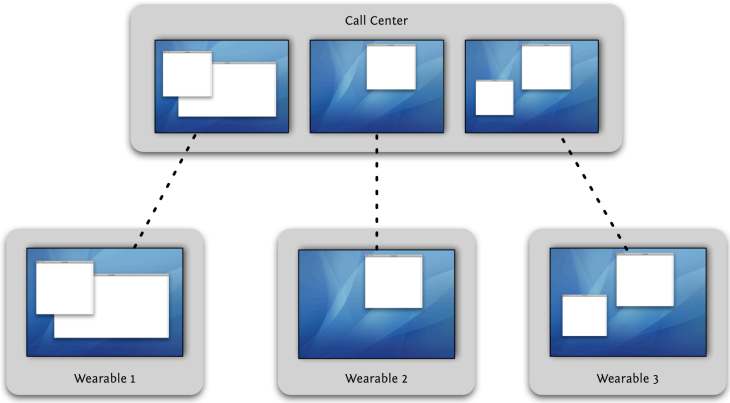


Figure 3.13 The interaction is distributed between two user-interfaces, the call-center interface and that of the wearable. The call-center can execute multiple activities concurrently while the wearable only has one activity resumed at one given time.

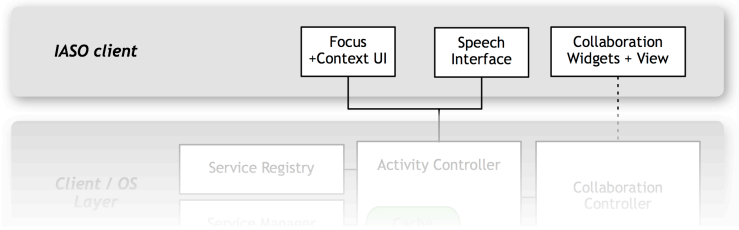


Figure 3.14 The *Iaso* client built on top of the activity-bar core.

real-estate. It is called *Iaso* and is a custom frontend built on the activity-bar base as depicted in Figure 3.14, but does not include the activity-bar user-interface itself. Instead it provides the user with a speech-interface for switching between activities and perspectives within the current activity, i.e. the "context" voice-command zooms out on the activity as described in 3.1 while the "focus" command, followed by a number, zooms in on the numbered application as shown in Figure 3.15. Many core functionalities are directly inherited from the activity-bar base software. These are features such as support for activity management, storage, activity sharing, context-awareness, collaboration awareness, and the support for activity-adaptation to multiple devices and displays.

The user-interface for the call center exhibits a further overview perspective by displaying multiple concurrent activities (Figure 3.16). Since all activities are synchronized, the user at the call center can monitor multiple persons engaged in multiple activities, and can easily switch between them. The layout of activities are maintained across all views, such that a common frame of reference between the collaborating parties always exist. The smaller frames shown in Figure 3.16 (a) are actually live-previews of activities. These previews enable the call-center user to overview multiple activities at the same time. The user-interface is meant to run on a wall-sized display and each preview is thus large enough to provide a

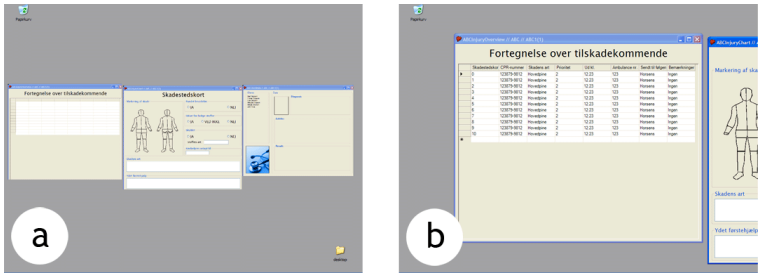


Figure 3.15 *(a)* Overview of the entire activity. *(b)* A detailed view of a single service.

glanceable overview.

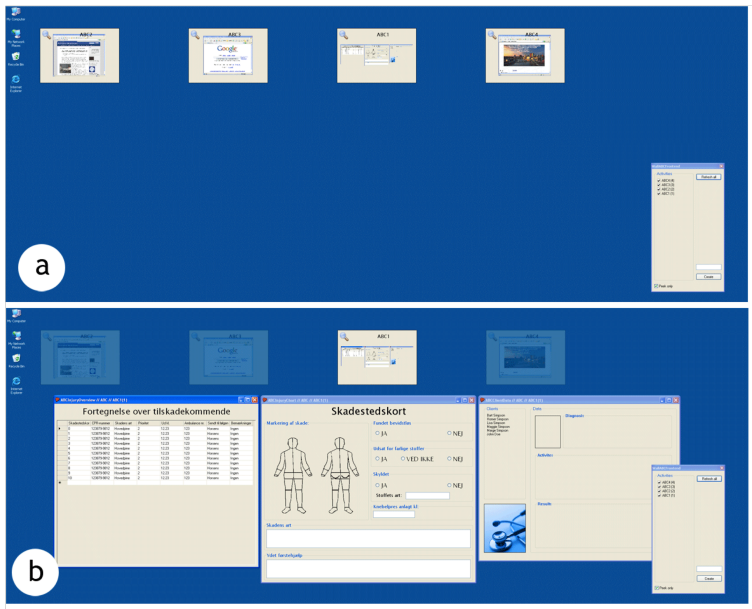


Figure 3.16 *(a)* Overview of multiple activities. *(b)* Overview of a single activity.

In the combined system we therefore have three of the perspectives on activities listed in Table 2.1, which can be acted upon by the principle of *focus+context*; the single application, the activity and the multiple activities perspectives. The wearable interface supports an application as well as an activity perspective while the call-center prototype supports all three levels as shown on Figure 3.17.

Support for the last perspective, multiple concurrently executing activities, is only supported by the call-center user-interface and is employed to provide a call-center user an overview of several distributed activities. This uneven distribution of interaction is useful in this specific situation where the responsibilities of the participants in the activities are fixed beforehand. The next prototype has support for a more fluid distribution of interaction.

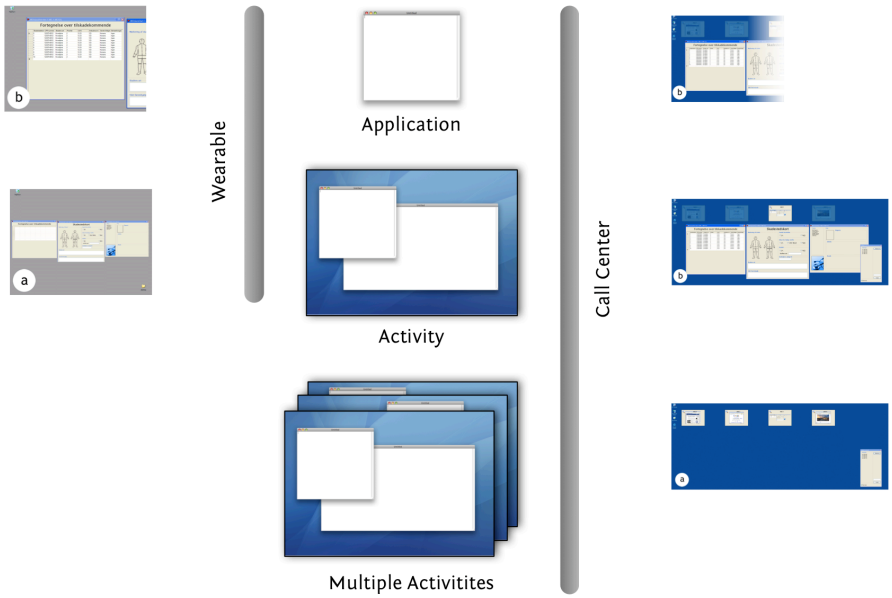


Figure 3.17 The two user-interfaces for the wearable system cover different perspectives on one or multiple activities.

Distributed Multiple Displays

Traditionally, a multi-display environment (MDE) is made up of multiple and heterogenous devices. These devices may be personal or public devices, and range from small devices (e.g. PDAs, laptops, and tablet PCs) to large wall-based displays, and are networked to form an integrated workspace [36]. The goal of such MDEs is to foster co-located group work utilizing all available devices. This includes the ability to fluently move all information between the devices; to place it on shared displays for presentation, comparison, discussion and reflection; to jointly create and modify information during problem solving; and to enable easy transitions between different devices in personal or public modes.

The goal of the Clinical Context-aware Activity Browser (CCAB) [111,164] system is to provide distributed interaction across multiple displays. The approach differs from that taken with the wearable computing system,where two customized user-interfaces were built. The CCAB is a single, but generic user-interface which can be adapted according to the context in which it executes. It supports execution of multiple activities simultaneously but also provides multiple perspectives to be used in the same user-interface. Figure 3.18 shows how multiple activities can run concurrently on each instance supporting a notion of distributed interaction which spans both multiple devices but also multiple activities on each device. This affords a greater flexibility in use than e.g. the fixed constellation of the wearable/call-center system.

The CCAB system is a novel approach for supporting interaction within *distributed* MDEs, where the displays are physically distributed across a large physical space. CCAB was designed for clinical work in a hospital, and the hospital is con-

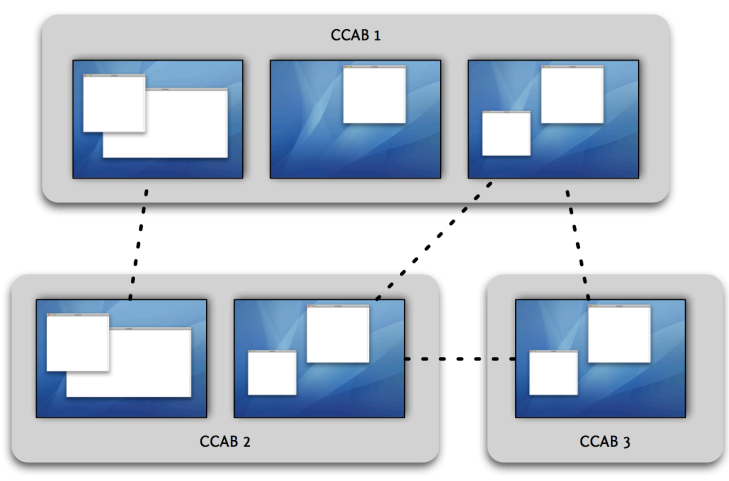


Figure 3.18 An example three CCAB instances. Each instance can execute multiple activities concurrently.

sidered as one single distributed MDE. The technology behind the CCAB system can be split into two parts: the interaction through the user-interface which is the topic covered in this section, and the underlying infrastructure as described in chapter 4.

The primary type of context which was used to adapt the displays was *location*; both of the devices and of persons – personnel and patients – within the hospital. For location-tracking, RFID readers and tags were attached to devices and persons and provided a rough, room-level estimate of their location. The adaptation itself was both in which activities to display as well as how to do it.

CCAB was designed for use with three types of displays each exhibiting a different degree of mobility and privacy as shown in Figure 3.19. First, the fixed LCD provides no mobility. It is configured to display activities relevant for a fixed location, the users present and the level of privacy associated with the location. The LCD can be placed in both locations, which can be considered private, i.e. an office or a closed meeting room, but also in more public locations such as a hallway. The second LCD was mounted on a mobile platform and could be moved between locations. Since the display requires an external power-supply the mobility is hindered somewhat, and the system must be shut down before the screen can be moved. However, the screen was configured to automatically detect its location and configure itself to match the privacy characteristics and the personnel present automatically when moved. The last display was a smaller handheld tablet device which easily could move between locations. The device is considered as a public device and could therefore be transferred from one user to the next causing it to automatically adapt itself to the new user, displaying his/her activities as well as those relevant for the location. A trait shared on all devices through the generic CCAB software is the ability to turn any of the devices into a "private" device. This is done by inserting a USB stick into the device. This clears all activities from the display replacing them with a set of personal activities. The stick may either con-

Figure 3.19 The interaction is distributed on three types of diplays. (A) is a mobile LCD, (B) is a fixed LCD, and (C) is a mobile personal device

tain a "key" which is used to fetch the private activities from the infrastructure or it may contain the serialized activities themselves. As long as the display is in this private mode it is not affected by the context.

Figure 3.20 The user-interface of the CCAB system, here three perspectives on different activities are displayed simultaneously (B),(C) and (D)

Figure 3.20 shows the user interface of the CCAB, which consists of four main parts: (A) the context bar showing this public display's current context; (B) "carousel" views of the activity list showing an overview of relevant activities within this context; (C) two activities that are resumed, each providing an overview of its associated resources; and (D) a X-ray viewer launched as part of one of the resumed activities.

The context bar in Figure 3.21 shows the current context of this public display. This includes displaying the current location of the CCAB instance using a blue

'home' icon, and nearby RFID tagged persons using a green label and a picture. If the public display is fixed to e.g. a wall, the current location stays fixed as well. However, if this is a mobile public display, such as either the mounted LCD or the hand-held tablet device, the current location is continuously updated.

Figure 3.21 The *context bar* in a public mode of use

Figure 3.22 The activity-list showing activities with varying relevance

The activity list of Figure 3.22 is continuously adjusted to show and highlight which activities are most relevant in this context. The current CCAB implementation operates with three levels of relevance shown as a large, small, and transparent red box. The icons attached to each activity reveals the type of relevancy: the icon with the two users illustrates that several persons co-located with the public display participate in this activity; the home icon illustrates that this activity is relevant for the current location of this public display. The simple algorithm for calculating the relevance of an activity is thus based on whether one or more of an activity's participants are present, or whether the activity is tagged as associated with the current location.

An activity can be resumed by clicking the "play" icon. Resuming an activity causes the user-interface to query the infrastructure for the activity. When the serialized activity has been fetched the "carrousel" is shown. It contains a list of previews of the resources associated with the activity as seen in Figure 3.20 (C). The user can now choose to either restore each resource individually by clicking the "enlarge" icon on each preview, or she/he can restore the whole activity, i.e. all of the associated resources by clicking on the "play" icon on the titlebar of the "carrousel". Restoring a resource means that this resource is fetched and displayed using a local application on the device[1].

The defining feature which CCAB shares with the call-center user-interface of 3.2 is that it has the ability to resume multiple activities simultaneously. In practice, this would typically require a rather large display equipped with some form of multi-touch technology, and we expect that this will be used primarily for co-located collaboration in front of large wall-based public displays.

[1]The current prototype only simulates this step.

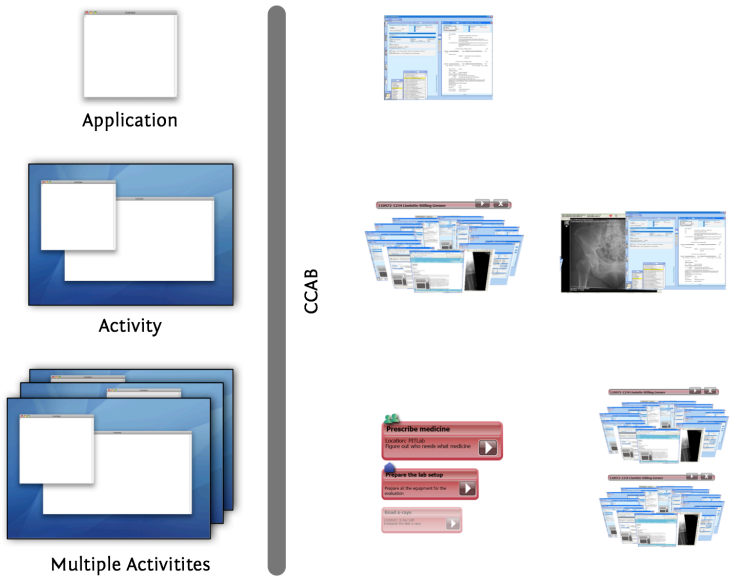


Figure 3.23 The single CCAB user-interface offers three perspectives on activities.

The CCAB system thus spans three of the perspectives in Table 2.1 (application, activity, and multiple activities) as seen in Figure 3.23 by virtue of the multiple views on activities. Single applications can be shown by opening an activity and then selecting a single window. There are two views which show a single activity. The first is the resumed view of an activity, where all the applications contained in the activity are shown. Here the user can interact with the individual activity. The second is the "caroussel" preview in which users can scroll through the applications, but not interact with them. Using this mode multiple activities can be previewed simultaneously. The last view simply lists the activity-name and description, and adapts the size and appearance based on the contextual relevance for the activity. This view allows a greater number of activities to be presented to users and provides an overview of the available activities, but does not allow the same kind of awareness as e.g. the call-center user-interface in the wearable system.

3.3 Evaluations

The desktop-based activity-bar user-interface and the CCAB system was both evaluated with real users, desktop-workers (computer science students and researchers) for the former, and hospital staff (doctors and nurses) for the latter. The two evaluations were scenario-based, i.e. the participants were asked to play through a scenario which involved using the software. The overall goals of both experiments was to investigate the applicability and usefulness of the interaction. The results presented here focus on *perceived usefulness* and are both based on questionnaires given to the participants after they had played through a number of scenarios.

Activity-Bar in Desktop Work

The scenario the participants were asked to complete, was based on a single user working at a desktop switching between two or three activities. Interruptions were then simulated, and participants were asked to handle them using the activity-bar. The primary objective was to establish whether the concepts of activity-based computing and interaction through activities could be applied to desktop-work, i.e. how *useful* the system was. Secondly, to evaluate how well the user-interface worked with experienced Windows users, i.e. the practical *ease-of-use*.

Here, only a few remarks will be attached to the results of the questionnaire, the full analysis is available in Paper I. On average the participants found it quite likely that the activity-bar interface would be easy to use. The ease-of-use questions included questions on learnability, understandability, and flexibility. Furthermore, the participants found that on average it would be slightly likely that the system would be useful to them. Looking more specifically into the underlying factors, the mechanisms for activity aggregation and activity roaming were perceived quite likely to be useful.

The questionnaires were followed up by interviews. The support for activity roaming through the suspend-resume operations, was here again perceived as useful:

> *The best thing is the ability to move your state from one computer to another. The whole idea of making it persistent [...] It's extremely nice to be able to close your computer and then it comes back up in the same state.*

The interviews also contained questions regarding areas for improvement to the current system and its user-interface. Users had some more general comments on the user-interface and suggestions for improvement. They found that the zoom could be greatly improved. Currently, the zoom functionality is simply too slow to be useful. Users raised, however, also more fundamental issues. First of all, the apparent lack of support for having the same service (window) in more than one activity. As argued by Bannon et al. [12] you need "multiple perspectives on the work environment" and need to support "multiple windows" as done in the Rooms system [10].

Overall the system was perceived as useful but still too premature to be used on a daily basis. The surprising result is how well the idea of activities and the interaction with them was received – one participant remarks how obvious the extension to the desktop seems.

Hospital Deployment

The evaluation of the CCAB system took place at a hospital. It involved 8 staff members (doctors and nurses) who used the system for a full day, going through a scenario which included treatment of an appendicitis[2] patient, ward rounds, a

[2] simulated, that is

medical conference, a surgery and ad-hoc discussions in the hallways of the hospital. The system was used to present relevant information contained in activities as participants moved from location to location and from activity to activity. Again, questionnaires were filled out and follow-up interviews conducted from which a few points will be made here. The full treatment of the evaluation is available in Paper III.

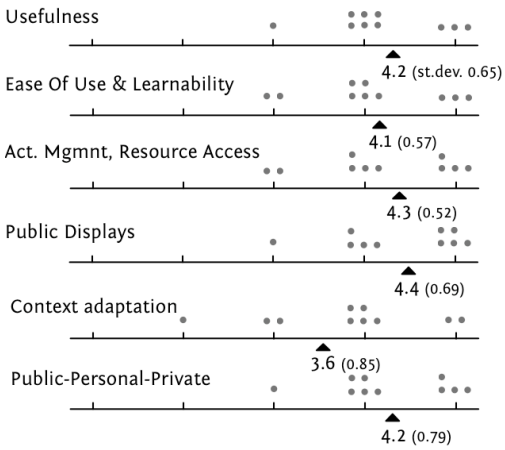


Figure 3.24 Results from the questionnaire in the CCAB evaluation. Likert scale 1-5, N = 8.

Support for activity management scores high, the clinicians either agreed or strongly agreed that it would be useful to gather all relevant patient information in an activity, that an activity would help recover relevant patient data quickly, and that it would be easy to get access to patient data while moving around inside the hospital. Furthermore, the use of public displays in a clinical environment was considered very useful. The clinicians agreed that fixed and mobile displays would be a good fit to the working conditions of a hospital; that it would be important to have the same CCAB interface on both the wall-based and the mobile displays; and that being able to just walk up to a public display and start using it would be a crucial feature. As expressed by a nurse:

> *[...] another good thing about this system is, that in a busy hospital [...] you do not need to spend time accessing data. You just go to the screen and get the data. [...] One can imagine that there are 10 of these mobile displays for each [patient] -ward and you just take one from a re-charger station -when you enter the -ward.*

The feature of context-based adaptation is not considered especially useful. In particular, the clinicians did not agree that CCAB was displaying the most relevant activities for a specific situation, and that using location tracking of personnel would help find the most relevant activities. Rather than co-location of people in front of the display, it seemed that the relevance of activities – or more specifically the patient cases – depended more on clinical issues like urgency of the case, or the order of the operations to be done in the operating room.

Overall the concept of having access to multiple activities on each device was well received, however some additional work is required to improve the selection and the priority of the activities which are shown to users.

3.4 Related Work

Systems designed to manipulate and present human activities have long been a research area within the HCI community. *Rooms* [41, 85] was an early, if not the original, system designed to support human tasks through the use of multiple virtual workspaces (or Rooms). Each Room represents a single tasks, and Doors are used to move between these. When entering a Room the workspace is reconfigured to the particular purpose of that Room, e.g. reading mail. The system also provides an overview displaying miniature versions of all available Rooms. The system is not distributed, providing local interaction only. Similarly, Microsoft's GroupBar [162] manages the local tasks of users by adding a user-interface for grouping application into higher-level tasks, and enabling users to switch between groups, as they would switch between applications using the standard taskbar. Another project from Microsoft Research; the Research Desktop [119] takes a more holistic approach and provides not only task-management, but also tools for taking notes, researching topics using a library tool and a tool to automatically create summaries from papers and books. The *Giornata* system by Voida et al. [179] includes both the grouping mechanism, via virtual desktops, found in e.g. GroupBar, but also comes with a tag-based organizing principle for tasks (activities) as well as a sharing tool for files and a contact palette which is used to communicate and share items with participants of the activity. This means that when a user switches between activities, not only the window configuration, but also included files and contacts are brought to the attention of the user once more. An extra dimension is added to the desktop in the *Task Gallery* [144]. Here, windows are distributed in a three-dimensional space. It uses a redirection mechanisms to route input and output between the normal windows and their three-dimensional representation. Tasks are represented on the surfaces of the environment with the current task on the "back-wall". Biehl et al. [36] use in- and output redirection in *IMPROMPTU* as well, not in virtual 3d environment but rather to distribute windows of an application between collaborating individuals. The basic mechanism is here the ability to share windows, essentially duplicating them on a remote machine as with VNC [137]. In addition, the system has a collaborator bar showing which applications are shared, and with whom.

The automatic detection of human activities is another related area of research. Kaptelinin [98] present a system which does automated monitoring of user-activity called User-Monitoring Environment for Activities (*UMEA*). The system uses interaction histories derived from internal application events, e.g. opening a webpage, which are then funneled into project contexts. These contexts and the associated can then be viewed through an overview tool which shows a time-based log of each. An analogous approach is taken by Dragunov et al. [61] in the *TaskTracer* desktop environment which uses the collected events to automatically analyze and detect the task of the current user. Tasks and resources can then be

accessed through a user-interface that is integrated into the Windows XP operating system, either as a replacement for the task-bar or directly in file-explorer windows.

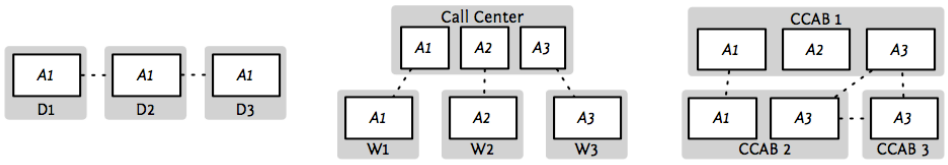
Temporal and spatial approaches to managing user's activities have been around since the introduction of the desktop-metaphor. In the LifeStreams system by Freeman and Gelernter [74] documents are chronologically stored in one-dimensional lists visualized similarly to Apple's Time Machine [9]; a backup and document recovery system. Rekimoto [138] extends this idea to a spatial two-dimensional desktop surface re-arranging files and other information as users navigate in time. They also add the ability for physical contexts to trigger time-travel, as well as linking the internal "current-time" of applications such that when a user e.g. reads an older email, other applications return to the state they were in when that email was received. Tashman [172] describe the *WindowScape* window manager which similar to Rekimoto [138] uses a two-dimensional surface to arrange information – in this case the windows of applications in the Windows operating system. The system includes novel features for navigating between windows and zooming in and out, but also a timeline enabling users to snapshot window configurations and later return to these. In *Taskposé* [33], the placement of windows changes dynamically over time. It uses an algorithm to predict the association between windows. The stronger the association is, the closer together windows are placed, automatically creating groups of windows. Work by Hsieh and Shipman [89] supports reviewing and playing back activities by the use of *activity-links* within hypertext documents. Activity-links are references to a point in the edit-history of a hypertext-document, and accessing them causes the playback of a "flashback"; a short interval of changes made around the edit being linked to. The links thus support time-based navigation within a hypertext corpus. This functionality is also found in the *Coral* [121] tool-set where multiple tools, including the Tivoli whiteboard application and the WhereWereWe multimedia framework, are interlinked through indexed points-in-time annotations allowing the simultaneous playback of multiple documents (media and text) in multiple applications.

The use of public displays to visualize activities or other context-dependent information increasingly receives more attention as more and more public screens emerge. Congleton et al. [50] presents the Proactive Display framework (*ProD*) which use a pipeline model starting out with presence detection, and going through a number of content dissemination steps before determining which content to display. It extracts information relevant to the context from a number of external sites (flickr, twitter, etc.) which are displayed on large public screens. Other pro-active displays include the *AutoSpeakerID* and *Ticket2Talk* for use at academic conferences[117]. In *Kimura* [110] the focus is on using wall-sized background displays to maintain a user's peripheral awareness of her/his tasks. As the user works on a personal computer the wall-sized display shows background activities which are sorted according to their current relevants based on contextual clues. Mccarthy et al. [116] distinguishes between three types of displays: Unicast-displays are confined to individual offices and displays content according to personal preferences, outcast-displays are positioned outside an office giving information about the individual occupying the office, and groupcast-displays are used in public workspaces

presenting information useful for people in its presence. The work by Izadi et al. [91] on the *Dynamo* project only supports a fully public display, but instead enforces user priviledges on the data-level. Users interact with the system by plugging in media-providing devices, e.g. USB-sticks, cameras, or mobile-phones, and the display is then used to present and share a wide range of media. Ambient public displays present information related, not to the presence of individuals, but more generic information related to the place or the surroundings of the display. Streitz et al. [167] define zones in the vicinity of public displays; the interactive zone is close to the display and where users physically interact with screens, the notification zone is where the user may be alerted by the screen displaying some form of personalized content, and the outer zone, the ambient zone, is where the screen displays ambient non-personal information. Vogel and Balakrishnan [175] expand the notification zone into "subtle"- and "implicit" interaction areas. The latter is aimed at catching the users attention, while the former is entered into when the user displays an interest in the screen and approaches it.

3.5 Conclusion

This chapter presented user-interfaces for distributed interaction with activities. The user-interfaces showed how interaction on the desktop could be enhanced with activities and how interaction could be distributed in multi-display environments. The chapter address *research question I* by showing how activities may be presented and manipulated by individuals, and how interaction and collaboration between multiple participants is mediated through the user-interfaces for manipulating activities.



A) Desktop-based (activity-bar). Views are syncrhonized. Single activity perspective.

B) Wearable (Iaso). Call-center executes multiple concurrent activities.

C) CCAB. All instances can run multiple activities.

Figure 3.25 Summary of how interaction is distributed on the three user-interfaces

The three main systems; the desktop-based activity-bar, the wearable- and the CCAB system each enables users to navigate within different perspectives both in a single activity, but also in multiple activities. The distribution of activities across user-interfaces facilitated interaction between users engaged in collaboration, and each system had a unique approach to how the interaction was distributed. This is illustrated in Figure 3.25, which is collected from Figure 3.10, Figure 3.13 and Figure 3.18. The three system instances in A is synchronized, all participants have an equal share of the interaction. In B the call-center user is afforded a higher-level

perspective enabling him/her to participate in multiple activities concurrently. The last system, illustrated in C, enables all system instances to execute multiple activities, and thus has a more fluid approach to interaction delegation since each display can be used to e.g. both provide overviews and to display individual applications.

Contributions

The contributions presented in this chapter revolve around user-interface technologies which allow distributed interaction within activity-based systems to take place. The implementation of these interfaces and their evaluation are the main contributions. The desktop-based system shows how activities can be made visible in the user-interface and provides handles for managing activities locally. It furthermore shows how asynchronous and synchronous collaboration can take place within activities, and how the user-interface supports this. These points are made in Paper I and IV which presents the interface and its evaluation. The wearable interface is presented in detail in Paper II and represents a contribution which applies the fundamentals of activity-based computing in the field of wearable computing systems. Another contribution is the interface elements which provide support for uneven distribution of interaction when a collaboration between a call-center and multiple wearable clients are initiated. The implementation shows how multiple perspectives on activities can be combined within one distributed system. Finally, Paper III demonstrates how multiple perspectives on activities can be combined within one user-interface and how this can be applied to the work of clinicians at a hospital. The paper furthermore introduces the concept of a distributed MDE where multiple heterogenous devices and displays form a configurable multi-display environment based on context information.

Chapter 4

Infrastructures for Distributed Interaction

The pillar on which distributed interaction is supported is the software *infrastructure*. The infrastructure is the "glue" that binds otherwise isolated devices together and enables distributed interaction which spans multiple devices to take place. The 'infrastructure' term can be applied to both the underlying hardware, the servers and other machines available, the network topology etc., but also the software which provides basic middleware services to applications. Here, the term 'hardware infrastructure' is used to denote the former, while 'software infrastructure' or simply 'infrastructure' when referring to the latter. Furthermore, the term "server" is occasionally used to denote the central process in a centralized software infrastructure. The focus in this chapter is on the design and subsequent construction of infrastructures for realizing distributed interaction both within activity-based computing but also on a more general level. The focus on design takes its outset in a number of architectural patterns, while the prototypes described here are the results of applying these patterns to specific problems or to specific environments. The infrastructures all represent a medium in which activities are managed, and provide the core functionality for distribution of data and events for collaborative sessions. The four key functionalities prioritized in the design and implementation of the prototypes presented here, are the following:

Activity Management Functionality for storage, distribution, and run-time management of activities is the basic "controller logic" for the specific domain of activity-based computing. It provides the backbone for moving activities between devices and for suspending and resuming them. In order to realize this functionality the infrastructure also needs to provide a computational construct representing the human activity. Activity management is derived from the principles of activity-based computing as described in Section 2.1, and especially aims at supporting *roaming* and *sharing* of activities. This functionality is associated with the application area of activity-based computing, but the key principle – that of domain specific controller logic – is a more generic and more widely applicable piece of functionality.

Event notifications To enable synchronization of distributed activities the infrastructure should provide mechanisms for notifying listeners of changes to data, e.g. activities, as well as mechanisms for listeners to declare which notifications they are interested in. This functionality is part of what is often referred to as publish-subscribe functionality.

Concurrency control It is necessary to provide functionality for managing and synchronizing the access to shared data, when the interaction with them is distributed between multiple machines and multiple users. This functionality is part of ensuring the overall consistency of the system.

Programming model Some form of interface for application developers is necessary for integrating external applications into a distributed activity-based system. This functionality is provided by protocols for communicating with the infrastructure and libraries which eases the integration. This will only briefly be mentioned here, as it is the focus of chapter 5.

These functionalities are shared by the prototype designs of this chapter and are part of the basic requirements for distributed systems [171]. The architecture of an infrastructure is a determining factor, both for which kinds of functionality can be provided, but also for its non-functional properties such as performance, scalability and stability. The rest of this chapter is organized to reflect this. First, a few basic architectural patterns are described. The following sections then each present an instantiation of the patterns in the form of infrastructure prototypes. The succession starts out with a *hybrid* architecture, which seeks to utilize the advantages of a client-server and peer-to-peer architecture, while minimizing the drawbacks. This is followed by a presentation of an infrastructure which is a *fusion* of the centralized and de-centralized architectural patterns. Lastly, a generic software infrastructure component is presented; a "meta"-infrastructure which focus on flexibility and whose architecture can be customized to its runtime environment.

Architectural Patterns

An important issue in the design of any distributed and collaborative system is its architecture [58, 49]. Traditionally, an archetypical distinction in distributed collaborative systems is made between *centralized* and *de-centralized* architectures. In centralized architectures, all users interact directly with a single shared central process, so that what one user sees, all users see. In a de-centralized (or replicated) architecture, on the other hand, each user interacts with a local application "replica" which accepts changes locally and then forwards them to the rest of the distributed system, often via a central server. The advantage of centralization is its simplicity, and that the functionality found in the central server removes the need for very complex clients. Replicated systems are often more responsive and offers better scalability. However, such systems are considerable harder to implement and maintain, since clients are now burdened with a great deal of code implementing non-business logic e.g. communication and concurrency functionality.

One approach towards a replicated architecture, but which still has a central process at its core, is that found in the client-server architectural pattern. This pattern uses a central server process as the mediator between replicated processes enabling them to communicate and synchronize their state. This partial centralization has many inherent benefits including simplified concurrency control, client and session management, since state and collaboration data only reside in one place. However, the scheme also has its drawbacks. The system relies heavily on a single server, making the system vulnerable to failures, scalability and performance. The client-server architecture also relies on a stable infrastructure with a well-known server process, which hinders its usage in transient and pervasive environments.

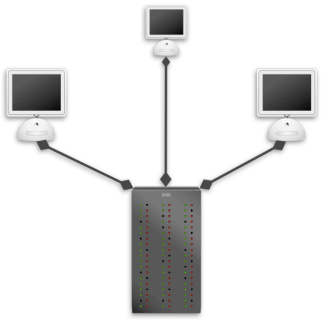


Figure 4.1 An illustration of the client-server architectural pattern.

Figure 4.2 The peer-to-peer architectural pattern.

The peer-to-peer architectural pattern is another architecture based on replication which address some of these issues. In peer-to-peer architectures, state information is replicated on all participating peers, and distributed synchronization algorithms are employed to ensure ordered communication and to synchronize states. This provides such architectures with the advantage of robustness to network failures, and the ability to function in many and diverse environments, enabling collaboration anywhere [64]. However, the advantages come at a high price. Complex synchronization algorithms and peer discovery methods are often required [181, 44]. This is a factor which significantly raises the complexity of implementing and maintaining peer-to-peer systems.

The client-server and the peer-to-peer patterns are the two archetypical patterns, which the infrastructures presented here are based on. The combination of the two patterns is used to determine the distribution of the functional components of the infrastructures, i.e. which functionality to centralize and which to distribute. In Figure 4.3 five architectures are depicted, the last three, marked C) - E), are based on a combination of centralized and de-centralized components, and are the focus of this chapter. The three layers are a simplification of the architecture of an activity-based distributed system. The lowest layer represents the data (activity-store), the middle is the controller logic (activity management), while the top is an application layer. The first example shown, A), is a fully centralized approach – basically a monolithic system. Here, all components run on a single machine and users interact with this single shared process. The architecture depicted

in B) is the client-server pattern, where distributed applications communicate with a central server. The prototypes presented in this chapter are represented in the next three architectures: C) is a hybrid between a centralized and decentralized architecture where part of the controller layer has been distributed to increase scalability and performance in collaborative sessions. This is the topic of Section 4.1 and Paper V. The next architecture, D), is the ad-hoc fusion architecture described in Section 4.2 and Paper VI where both collaboration controllers as well as data is distributed in order to enable the overall system to function in an ad-hoc manner. The ad-hoc functionality relies on a client-side cache which can be kept synchronized both with and without a central infrastructure. The last example, E), is a generic and customizable infrastructure component. It is not specifically designed for activity-based computing (as C) and D) are), but it can be customized to be useful in this domain as well. In this architecture both data and functionality can be distributed across multiple machines, and each deployment of the infrastructure may be tailored to fit a specific purpose. This infrastructure is the focus of Section 4.3 and Paper VII.

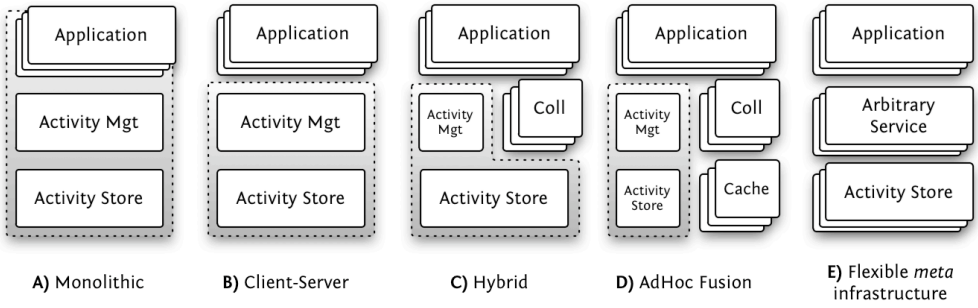


Figure 4.3 Five infrastructural architectures. The components inside the dotted line represents the centralized component.

4.1 A Semi-distributed Hybrid Architecture

The design and implementation of the hybrid infrastructure is part of Paper V. The purpose of exploring the concept of a semi-distributed [58] or hybrid [181] architecture, a combination of the client-server and a peer-to-peer patterns, is to take advantage of the strengths of both approaches while minimizing the drawbacks. It makes sense, for example, to keep concurrency control in a central location, while using a peer-to-peer scheme to route traffic which is not subject to synchronization. This also enables the system to operate in transient environments for shorter periods and with limited functionality. Environments where maintaining a persistent connection to a central server process might not be possible.

One of the main drawbacks with a pure client-server based infrastructure is that it does not immediately scale well. The server quickly becomes a bottleneck as the number of clients increase. In the case of the activity-based computing the

problem is greatest when one or more collaborative sessions are started. This is caused by heavy traffic imposed on the system by the collaboration widgets as seen in Table 4.1 which shows the distribution of traffic in a typical collaboration session. Telepointer and voice are by far the most eager producers of data. This observation is used to improve scalability and the responsiveness of individual clients, by doing a basic differentiation between what we have termed *accountable* and *ephemeral* events.

	Amount (Mb)	Percentage
Activity state changes	0.068	0.05
Telepointer	5.900	3.85
Voice	146.500	96.10

Table 4.1 Data traffic measured in a typical activity sharing session between two users working for 30 minutes.

Event Routing

The differentiation between *accountable* and *ephemeral* events is used to determine how to route events and which concurrency and transmission properties that needs to be applied. This decision whether an event should be labeled accountable or ephemeral also affects which parts of the hybrid architecture to activate; the central client-server infrastructure or the distributed peer-to-peer architecture. Accountable events should pass through a central server; the ephemeral events can be distributed by a peer-to-peer scheme. A simplified illustration of the architecture of this infrastructure is C) in Figure 4.3.

We place the label *accountable* on those events which can benefit from central processing, because the infrastructure must be held accountable for their delivery in order to ensure overall consistency between clients. Examples of accountable events from the literature are the classical text insert, move, and delete commands in collaborative editors [185] or the state changes in general purpose frameworks like Corona [157] or GroupKit [150, 79]. Generic collaboration state which is subject to concurrency control, session and client management, and events containing information which needs to be persisted are included in the accountable category.

There are, however, a range of other event-types which need not be subjected to the same level of accountability. Such events are typically absolute values, independent of previous and subsequent events, and where the loss of a number of events does not affect the overall stability of the system. We call these events *ephemeral* because they are short-lived and transient. Examples of such events are telepointer events, voice events, and other collaborative awareness events like the ones found in the MAUI Toolkit [87].

The peer-to-peer architecture scales much better for large amounts of data traffic since there is no single process which all data must pass through, however it is complicated to use it for transmitting accountable events. In an activity-based

system, a concrete example of accountable event data is changes in the state of an activity, e.g. when a user moves an application window. This event – a state change – is routed through the server (the size and position of an application are part of the activity state). Voice and telepointer data on the other hand are, as previously mentioned, examples of ephemeral event data which. As Table 4.1 suggests, this removes a significant burden from the server, and allows for an increased responsiveness of the interaction at local clients as well. This observation is confirmed by earlier experiences with Rendezvous [133]; where Patterson et al. argue that "many synchronous multi-user applications (e.g., games, whiteboards, etc.) have only modest throughput requirements. [Therefore], it seems reasonable to centralize shared state" [134, p. 125].

This performance gain is our main motivation for using the hybrid architecture. A secondary goal is to enable the clients a limited level of functionality when no server is available. This is achieved by maintaining a local cache which abstracts away the connection to an infrastructure and which continue to provide access to cached data when no infrastructure can be reached.

Design of the Hybrid Architecture

The hybrid architecture consists of two tiers as illustrated in Figure 4.4. The lower tier is the central process, the server. The middle tier is client layer is the distributed tier which executes on all connected clients.

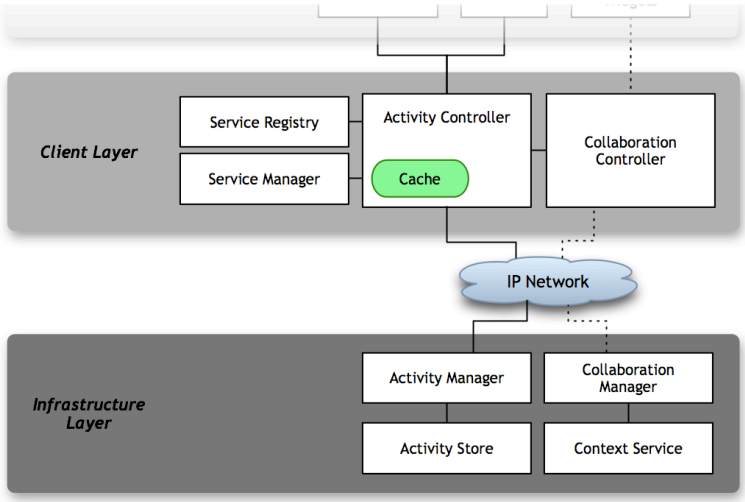


Figure 4.4 The two tiers in the hybrid architecture. The upper-most (hidden) layer is a user-interface as presented in chapter 3.

Server Tier

The core of the infrastructure is responsible for both the storage, the runtime management of activities, initiation of collaborative sessions and synchronization during those sessions, as well as handling contextual information. Persistent storage is done in the activity-store, which maintains updated serialized versions of all activities in the system. Activities are modelled hierarchically, and can thus easily be serialized into XML documents, which are then served to clients.

The activity-store also provides functionality for querying for certain activities based on the meta-attributes attached to activities such as the participants of an activity. Runtime management is the responsibility of the activity-manager which keeps track of which activities are resumed on which devices and by whom. The collaboration-manager is the central component which synchronizes access to an activity and provides notifications when the state of an activity is changed during a collaboration session. The concurrency mechanism is quite simple and uses an optimistic scheme. This is possible because all changes have to go through this central component. The protocol used for communication between the central server and connected clients is a stateless `HTTP`-based protocol. It uses the standard methods of `GET` and `POST` to fetch and commit changes to activities. Activities are referenced using a simple URL scheme. Clients are automatically notified of changes in activities they have resumed, which forms the basis of the synchronization during collaboration sessions.

Client Tier

The client tier is responsible for managing the local execution of activities at each client. The service-registry and -manager keeps track of local resources and applications and is able to map the descriptions of activities, the serialized XML, onto these local resources when activities are instantiated. Collaboration is handled in the collaboration controller which is responsible for the routing of collaboration events and for the peer-to-peer distribution of ephemeral events. The activity-controller is the component for managing the runtime state of local activities. It uses the cache to retrieve and post activities and changes to activities to the infrastructure. When no server is available the cache allows the client to continue to function, although in an isolated mode where no external events are received or sent. This is one factor in reducing the reliance of the central server. The cache still gets updated with local changes, when the server does not respond. This ensures that all local changes are saved and persisted on the local machines while the server is down. When the cache discovers that the server is responding once more, it simply plays back the changes from last successful request allowing the server to merge these changes. The other part in reducing the dependency on the central server is the stateless protocol, which ensures that as a server comes back up from a failure the clients can simply continue communication with the server where they left of. Temporary server crashes are therefore easy to recover from as long as the server state has been persisted. Conflict resolution is done at the server simply by ordering incoming requests and applying them them to the data in the

order they arrive. If an unresolvable conflict is detected during a merge phase, the conflicting state is duplicated such that both versions are represented within the activity. This has the effect on the clients of producing two instances of the application, which represents the state and allows the users themselves to solve the conflict. As this merge of conflicting states takes place the clients are not notified of changes, but when it has concluded an updated activity-state is sent, forcing clients to one common state once more.

Evaluation of Performance and Scalability

The claim that a hybrid architecture, where the central server is used only for accountable events in combination with a stateless protocol, performs and scales to a realistic number of users, is the focus of this section. The experiment presented here evaluates the central infrastructure, and its behaviour when distributing accountable events. The non-functional properties under evaluation are the *performance*, *reliability*, and *scalability* properties. In the experiment an activity is created and a number of users join and resume it. The test-harness then performs a number of actions within the activity, simulating a real world collaborative situation. By having only one activity for large numbers of users, we stress both the server and client implementation because any change must be propagated to all users (and thus all clients) and all changes automatically conflict making it harder for both server and client to agree on a consistent state. A detailed description of the experiments and further results are given in Paper V.

We measure the *performance* of the server as the time it takes to distribute a single event to a number of clients. The *scalability* is measured in terms of how increasing the number of clients affects the performance of the server. *Reliability* is measured as the number of times an event is lost (or extremely late). All clients run on a single machine in order for us to time the arrival of events consistently on all clients. To simulate adverse network conditions we included simulated delays in one run of the experiment. We use the same delays as [49]: 72 ms simulating a user in Germany, 162 simulating a modem user in Germany and 370 ms simulating a user in India (assuming that we are located in the U.S.).

The results of the performance experiment is shown in Figure 4.5 where mean-time is shown as a function of the number of clients. The response time of the client-server round trip remains under 1 second throughout the experiment. The graph is approximately linear[1], which indicates that the architecture scales well. Figure 4.5 also reveals that the simulated delays only affect the results by a constant factor which is as expected.

As can be seen from the results, the central server is capable of handling at least 100 users within a single activity and this is clearly sufficient for the purpose. Furthermore, the experienced delays for all users in a 100-user session is less than one second (including simulated delays). It is difficult to say if this is an acceptable delay since it depends very much on the applications involved in the activity. Also, esti-

[1] $R^2 = 0.97$

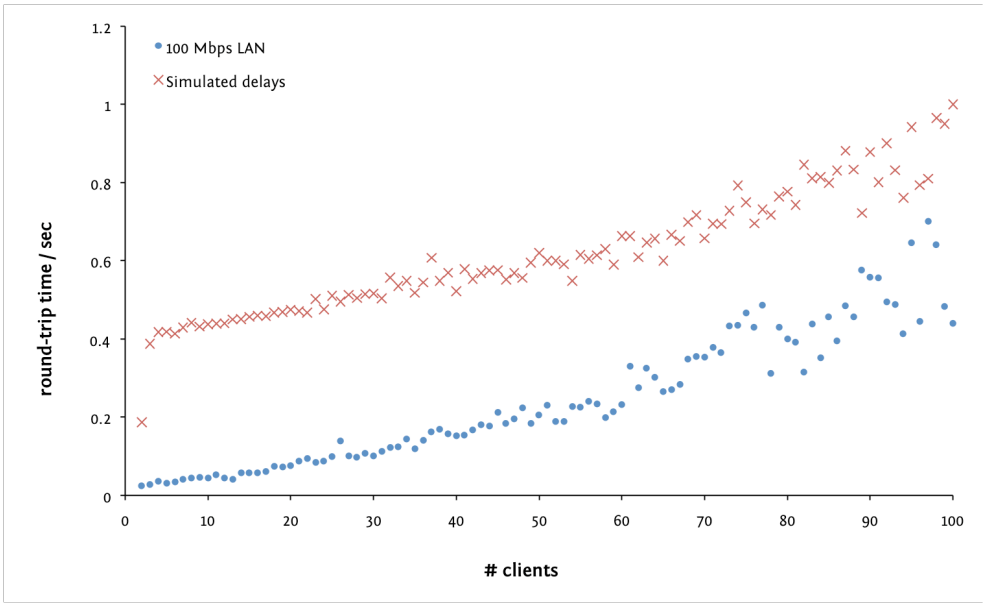


Figure 4.5 Results from experiment the evaluation of the central server's ability to handle and distribute accountable events.

mating the maximum number of users is troublesome – but the nature of real-time collaborative work arguable imposes a limit which is well below the 100 limit used in our experiments. The central server strongly simplified handling concurrency control, session management, event notification, integration, and late-comers to a session. For these reasons, a centralized concurrency control and decentralized mass transit still seems to be a valid architectural choice from a performance point of view.

Two points are thus worth repeating on the use of a hybrid architecture for an (activity-based) distributed system. First, by distinguishing between ephemeral and accountable events, and separating the mechanisms to propagate them, we are able to off-load much data management from the server and in this way improve performance and scalability. Second, by having a stateless protocol, and by incorporating client side caches to enable the clients to work without server connectivity, we are able to use the hybrid architecture in transient environments also. The caveat here is that as long as the clients operate without the server they are in essence isolated from the rest of the system and cannot participate in collaborative sessions.

4.2 Distributed Ad-hoc Architecture

The main caveat from the hybrid architecture described in Section 4.1 is that it still relies on a centralized infrastructure component to realize synchronized distributed interaction between collaborating individuals. This limitation is the main motivation behind the ad-hoc fusion architecture. Here, the focus is on extending the reach of distributed interaction by allowing ad-hoc collaboration to take

place when no central server process is available. The core architecture remains unchanged from the hybrid architecture described in Section 4.1 and is thus still an infrastructure built exclusively for activity-based computing. The main difference is found in the distributed clients, where the local cache is instrumented to allow the cached activities to synchronize themselves across multiple clients in a peer-to-peer fashion. This corresponds to D) in Figure 4.3 where each distributed client is essentially equipped with a slice of all three layers enabling them to operate independently of a central server. The main contribution presented here is therefore the ability of every client in the system to continue synchronous collaboration, even in situations where there is no access to a central server. The clients can thus operate with complete functionality in both ad-hoc and server-based execution environments, increasing the robustness and scalability of the overall communication platform. The implementation of the architecture is described in detail in Paper VI. The following represents an overview of the architecture and its use.

One setting which illustrates the need for infrastructure-less operation is the changing environments of emergency workers or paramedics as illustrated in Figure 4.6. At a hospital, the hardware infrastructure is fully evolved, including high-speed, reliable network connections. This permits the use of centralized architectures, possibly utilizing direct communication between the clients, thereby leveraging the server load. As the paramedics and other emergency workers move away from the hospital, the environment changes. Network connections decrease in both bandwidth and stability. Furthermore, a central server server might not be available for all clients and they may thus rely on ad-hoc peer-to-peer connections.

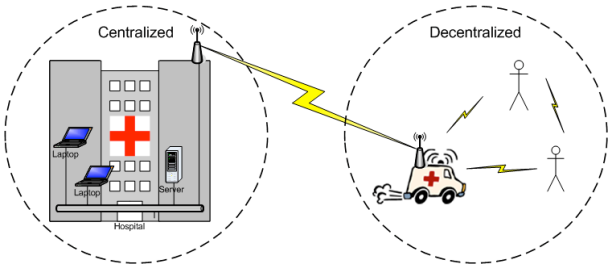


Figure 4.6 The fusion archiecture is capable of adapting to the environment – each client remains fully functional with access in both centralized and de-centralized environments.

The prototype enables both ephemeral and accountable events to be distributed in a peer-to-peer manner where the hybrid architecture was limited to distributing ephemeral events in this manner. Even if the different clients in the ad-hoc network should loose connectivity for a period of time, the individual clients will offer the ability to continue work while disconnected, and upon reconnection the server-based synchronization mechanisms will resynchronize the network. If one of the clients in the ad-hoc network should come within range of the infrastructure, it will connect to the server, but still remain in the ad-hoc network. This enables the client to act as a bridge as well, relaying state changes in the decentralized ad-hoc network to the centralized infrastructure and vice versa as depicted in Figure 4.6. This means that it is possible for a client in the ad-hoc network to engage in a collaborative session with a client connected to the infrastructure, cre-

ating a network architecture which is a *fusion* of a centralized and a decentralized architecture.

To enable this infrastructure-independent operation, an additional pure peer-to-peer scheme was added to the clients. This additional distribution scheme is one based on *peer-to-peer distributed shared objects* (PDSOs) [122]. The cache on each client already stores activity-objects representing individual activities, and these are kept synchronized through the central server infrastructure when it is available. The cached activities are thus the most recent state that a client has seen when it goes offline. Once a connection to the server is lost or cannot be established the cached activities are made active as distributed objects and any two clients having a cached copy of the same activity, and who are within reach of each other, will get their activities synchronized. The cache now essentially mimics the functionality of the server allowing the upper layers of the client to continue functioning despite its disconnected state. Using the concept of peer-to-peer distributed shared objects, we are thus able to create a fully decentralized communication architecture, capable of supporting the same types of collaboration as the centralized hybrid architecture. Delivery guarantees are still handled in the same manner as the centralized hybrid architecture, i.e. by distinguishing between *accountable* and *ephemeral* communication, but now by using a (more complicated) distributed mechanism for concurrency control and conflict resolution [VI].

The combined result is thus a fusion of the hybrid presented in Section 4.1 and a pure peer-to-peer scheme using distributed objects, making the infrastructure capable of adapting to the environment. The adaptation mechanism ensure that work can continue in both environments and that synchronous collaborative features remain available. The architecture furthermore demonstrates a novel way of *fusing* disparate execution environments.

4.3 A Flexible Distributed Architecture

The previous two infrastructures were designed to solely support activity-based computing. This is a limitation in the sense that they incorporate a fixed model of activities, and that the functionality the infrastructure provides is static, and cannot easily be adapted to suit a specific purpose or the requirements of a specific environment. It is hard to use these types of infrastructures to experiment with new environments and new concepts for activity-based computing because all new functionality has to be incorporated and exist alongside all existing functionality. This led to the architectural pattern shown in E) on Figure 4.3, and the implementation of a generic, flexible "meta"-infrastructure which provides the basic building blocks for constructing customized as well as flexible infrastructures. The pattern is basically meant to allow both functionality and data to be distributed on multiple machines, and the implementation accomplishes this with a "micro-kernel" [105] approach which is inherently modular, and by using a shared data-structure. One of the applications of the infrastructure is still deployments of activity-based computing as described later in this section. The infrastructure prototype is called *aexo*

and is described in detail in Paper VII. The name stems from one of its original purposes which was to provide a medium in which applications could export or externalise their internal state, i.e. serve as an "exo-skeleton" for applications. The externalization or sharing of state is also one of the main concepts in activity-based computing and is how applications are made part of activities. The principles behind the design are:

Customizable distribution The environment in which the system executes is highly distributed, therefore the infrastructure should allow for easy partitioning of functionality and data on the available hardware resources. This is in part to ensure that the infrastructure scales, but also to ensure that it can easily be made useful for a larger variety of situations and environments.

Extensible & Reusable The infrastructure must allow for easily adding new functionality, it should be customizable what exact functionality is available in different environments. This customization can be done after the infrastructure has been deployed, i.e. at runtime, to enable the infrastructure to adapt dynamically to its environment. Reuse is accomplished by enabling independent software components to take advantage of the functionalities of one and another [103].

Transparency The distribution of data and events should not be hidden from developers, but rather be exposed such that informed decisions can be made in the applications using the infrastructure.

These three principles guided the implementation of the prototype. The principles all have their foundation in the desire to build a flexible meta-infrastructure – one that is easily made usable in many and diverse environments and which may be customized to a great extent. The notion of flexibility as defined by Parnas [128] is realized by moving away from monolithic implementations of an infrastructure and towards seeing the infrastructure as a set of interrelated applications. The main features in the implementation of the architecture, are a shared and distributed lightweight storage and event system, and a mechanism to extend the core system with additional functionality by loading and running external components.

Design of the Aexo Architecture

The architecture consists of two tiers; the core infrastructure and an optional client tier which implements the functions needed to communicate with the infrastructure. The client tier is the subject of chapter 5. The core infrastructure is kept as simple and minimal as possible. The functionality here is formed first by a hierarchical map (hmap) – a tree-like, path-indexable data-structure containing all data and which represents the structure in which data (e.g. an activity) is modeled. Second, by a component-loader which allows adding of functional components at runtime. The components then make out the actual business logic of the deployed infrastructure. They run in their own sand-boxed environment, but have access to

the same data, that stored in the shared hmap. An illustration giving an overview of the core system architecture is given in Figure 4.7 with example components installed. The following is a brief description of some of the key elements in the architecture.

Hierarchical Map

In aexo, single data items are stored as nodes in a hierarchical map. Each node is referenced via a URL scheme inspired by the REST architectural style [70]. The structure makes it simple to model data which is hierarchical in nature. The structuring scheme is also used for event subscriptions and event propagation. Subscriptions are made on a submap basis. For example, a subscription for `/Parent/Child` will receive notification of any changes to this node, and to all nodes in the submap of `Child`. External applications can thus subscribe to changes from any node or any submap and will get notified when modifications are made to either.

The API for accessing the map is intentionally kept very simple. The operations for reading from the map are `read` and `access`. The former simply returns the data element stored at the given path, while the latter performs some extra logic to, if possible, evaluate the data element. The result of the `access` operation depends on what type of data element is stored in the node. If the element is a reference to a file, for instance, the `access` operation might take a range as argument and only return the data in that range from the file. Modifications to the map are done with the `write` or the `delete` operation. Subscriptions are set up with the `subscribe` method, where callback is either a URI or a subscriber-object for local listeners, i.e. components loaded in the address-space of the aexo instance.

The data distribution mechanism is based on the mounting of remote submaps which can then be queried almost exactly as if they existed locally. These remote mounts are established by writing an endpoint address into a node. Subsequent requests for this node or its descendants are not automatically sent to this remote address though. Rather the requester has to explicitly specify that a request may traverse across mount-points – if this is not done then an error is returned. This is done to ensure distribution transparency, and to enable clients to discover how data is physically distributed.

Subscription Manager

The *Subscription Manager* as depicted in Figure 4.7 handles subscriptions for nodes in the map. Subscribers will be notified about changes to the node or events occuring within some depth of the submap below. Notifications can also travel over mount-points, letting subscribers subscribe to events occurring in remotely mounted maps. A notification is piggy-backed with a *context object* which is a description of which changes in the map caused the notification. Initially the entity writing to the map need to supply a descriptive text for the change, which is then bundled with the actual change (the path, new and old value) into a context-object attached to the notification of the change. Subscription owners which, upon re-

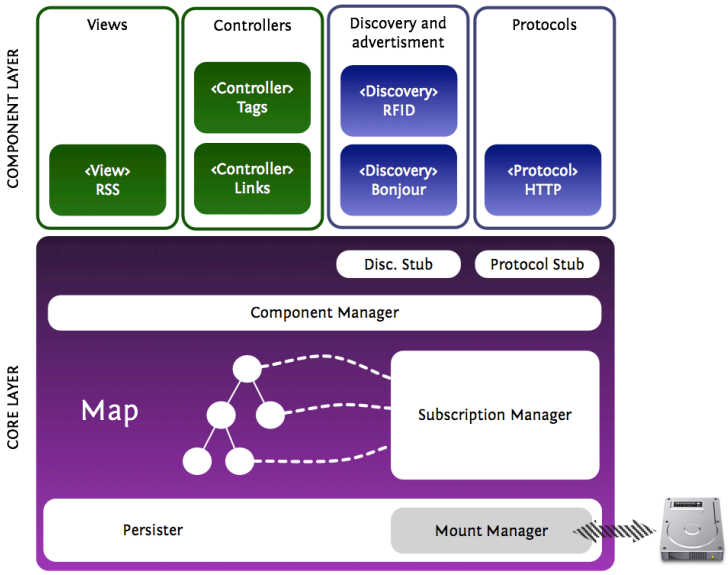


Figure 4.7 An overview of the aexo core system. The core layer is responsible for managing access to data and loading of components. The component layer contains all other functionality. Each individual component is loaded at run-time.

ceiving a notification, again write to the map, are required to add another entry to the context. The context then ends up containing a sequence of modifications and accompanying descriptions allowing the receiver of a notification to browse the complete causal chain leading to this exact event notification. This results in increased transparency for event chains which may become arbitrarily complex and enables the infrastructure to supply human-readable explanations as to *why* any given change happened. Since events always originate from modifications to data, and data is not replicated but rather referenced via mounts, concurrency control can be handled locally by the machine which actually holds the data. This is the same scheme which is used in the client-server patterns, where state changes and the resulting notifications are ordered centrally, or in this case, centrally according to the data distribution. Centralizing concurrency control again significantly reduces implementation complexity, while distributing data improves the scalability issues with a single central server.

Component Manager

The *Component Manager* handles loading and unloading of components and manages the resources which are allocated for each component. Components are loaded at runtime from `jar` files in a given directory. If a new component is copied to the directory it will automatically be loaded and instantiated by the component manager. Each component runs within a contained environment, but is given a

reference to the map in which it can then read, write and install listeners.

The system has specific support for two kinds of components: discovery and protocol. The *discovery component* is used by the map to advertise its presence and its shared submaps, and to look for and mount other shared submaps. Currently, aexo implements a Bonjour [8] discovery component, and one which uses an external RFID reader. *Protocol components* allow the infrastructure to dynamically load communication protocols. A resource constrained device, for instance, might not be able to host an HTTP protocol, and would rather implement a lighter protocol. All other components implement either view or controller logic specific to any one environment. View components are components which export a specific view of the data contained in the map. The RSS component, for example, depends on the HTTP component to export an RSS feed for a chronological list of modifications made to a specific submap. A common task of a controller-component is to maintain a specific perspective on one submap in another submap. For example, to look for location information on nodes in one submap and then use this information to maintain another location-centric submap, i.e. one where nodes represent locations and their children represent entities contained in these locations. External applications are then able to take advantage of this controller logic by subscribing to changes in strategic locations in the controller maintained submaps e.g. the location centric submap. Thus, it is possible to encode controller logic directly in the infrastructure, allowing transient applications to take advantage of environment specific logic which is kept external to the application itself.

The component manager enforces a simple component dependency scheme in which a component may indicate that it depends on another component to be available. An example of such a relationship is found in the RSS component which, as previously mentioned, relies on the HTTP protocol.

Aexo Example

The main aspects that sets this infrastructure apart from the hybrid and ad-hoc systems of previous sections is, that it is flexible in the sense that the set of components making up the functionality of the infrastructure is dynamically loaded at runtime, and furthermore that the data the infrastructure contains can be partitioned on multiple devices. This enables the tailoring of the infrastructure such that it can be used in different environments and situations.The data distribution is accomplished by mounting remote submaps. This means that no or very little data replication takes place. Each aexo instance holds a unique piece of the data in the whole system, and by mounting these remote maps in strategic locations the access to the data (and not the data itself) can be centralized. The infrastructure can therefore both be centralized by having a single aexo instance, but may also be decentralized by using multiple. Each instance can similarly load different components, thus distributing the overall functionality of the system on the available devices.

An illustration of an infrastructure consisting of three simple aexo instances is given in Figure 4.8. Here data has been partitioned on three machines marked 1,

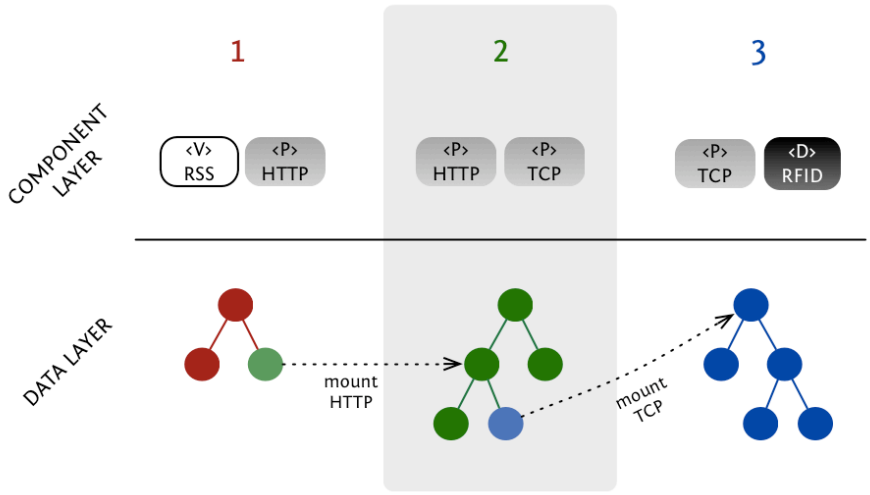


Figure 4.8 An example customized infrastructure. The components are typed with ‹V› for view, ‹P› for protocol, and ‹D› for discovery.

2 and 3. Each machine is responsible for one portion of the combined data in the system but most data can be accessed centrally from machine 1, since this machine has mounted a subtree on 2 which again mounts the root of 3. The data on 2, which is in the right subtree, is private and only accessible via this machine. The example also demonstrates how a bridge between two machines, 1 and 3, which do not share protocols can be established through an aexo instance that has both of these protocols loaded. This lets machine 1 access the RFID discovery component of 3. Functional components, other than the protocol components, are also distributed between the three instances. Machine 1 has a view component for exporting RSS feeds loaded, and 3 has the discovery component for detecting RFID tags. Event-notifications also travel across mount-points. For example, assume that the RFID component on machine 3 in Figure 4.8 continuously updates a submap with information about which RFID tags are available. A subscriber can then subscribe to these updates from machine 1 by installing a listener in either the local mount-point or any of the ancestors on either machine 2 or 3. This is the case with the RSS component which listens for all changes and maintains a feed listing these changes and their context-information. Thus, by browsing this feed a user would be able to see a history of which RFID tags were detected by 3. This is a very simple example of how distribution of both data and functionality is tailored to a specific setting.

To summarize; specific components can be dynamically loaded in order to extend the basic functionality at each aexo instance creating a software infrastructure where functionality is determined dynamically, and moreover where functionality is distributed on multiple machines. Data can be partitioned on multiple devices; one unit of data spanning a submap, e.g. an activity, can be split among several machines. Transparency is achieved in terms of data distribution by requiring requests which traverse mount-points to explicitly state that this is the intention with the request. The modifications leading to event-notifications are made transparent by

having each event carry a context object containing the list of modifications which caused the event. This allows subscribers the option to reason about why an event occurred. To further illustrate the customizability and extensibility of the aexo architecture, two concrete applications are presented next.

Applications of Aexo

This section presents two customized applications of the aexo infrastructure. These examples are both non-trivial and represents real-world applications tested with actual users. Furthermore, they are very different technically and are designed for quite different domains as well. As such they demonstrate the utility of aexo for building customized infrastructures for diverse environments.

Conference Room

The inSpace conference room infrastructure [178] was designed to support interaction around a conference table with multiple users interacting with a shared repository of data. The conference table itself was equipped with RFID sensors at each seating position, used for detecting the RFID-tagged laptops. The room was equipped with a projector and a large LCD display. The participants seated at the table would have an application installed on their laptops which enabled them to interact with the room infrastructure, and, through the infrastructure, with the available hardware resources, i.e. the laptops and the machines running the two display devices. The combined software therefore consisted of a single aexo instance, a broker system for event coordination, the transient applications running on participants laptops, and the fixed views running in separate applications for presenting data on the shared displays. The aexo infrastructure was used for coordination of events as well storage, distribution and organization of the shared data.

The typical data shared during meetings were documents, text-, pdf- or MS Word files and presentations. All files were stored in a submap specific to each meeting, but this was never accessed directly by any of the participating applications. Instead, a controller-component would organize another submap based on the "tags" applied to the shared data, and furthermore, would add automatic tags to all shared items. These automatic tags were based on context information, for instance one tag applied was the title of the current meeting which the controller extracted from a calendaring system, while others represented the people present in the room. Other tags were manually applied using the application running on each participants laptop. The resulting tag-centric submap maintained by the controller was then used by the display devices. Items tagged with e.g. "projector" was automatically retrieved by the machine attached to the projector and shown in a grid of available media-items on the shared display. Likewise, the laptop applications could set up subscriptions for a given tag, e.g. "meeting x", to have a local copy of the submap extracted from aexo and stored on their local device.

The single aexo instance was thus customized to the conference environment

and specifically the use-situation. Adding or removing functionality to the system following its deployment is simply a matter of loading or unloading components.

Clinical Context-aware Activity Browser

The Clinical Context-aware Activity Browser (CCAB) system [III] is an interactive system designed to support activities in hospitals. The interaction techniques and user-interfaces implemented in the CCAB system has previously been presented in 3.2. To recapitulate, the user-interface is distributed across a number of public displays; some fixed larger screens and some tablet-sized mobile devices. These interfaces provide context-adapted *views* of the activities modeled in the infrastructure, and controls for manipulating and browsing content of the activities. A typical activity could for instance be the treatment of a patient. The applications and data contained in it would then be e.g. the electronic patient record, lab test results, and radiology images.

The CCAB system was built on top of the aexo infrastructure, which provides support for storing and distributing activities. Activities can easily be modelled in the infrastructure by mapping the hierarchical structure of an activity onto the map. The infrastructure keeps track of contextual information such as the location of personnel and mobile devices, and organize and merge this information into the activity model. Location tracking of personnel and devices is done using active RFID. In aexo, a set of controller components map tracking information from the RFID readers to the corresponding nodes in the hmap. CCAB clients listen to changes in this location sub-map and are continuously updated to display information from the submap which match their current location. This has the effect that the mobile devices are continuously showing the relevant activities for their current location, and, in the case of a mobile device, the personnel carrying the device. The CCAB system including the aexo platform was deployed and tested at a large hospital by a team of doctors and nurses in a scenario-based evaluation as reported in Paper III.

4.4 Related Work

The related work listed here are examples of infrastructures which support distributed interaction and ranges from groupware, collaborative systems to publish-subscribe architectures. The *Aura* [75] project is a middleware for pervasive computing which focuses on resource adaption and on providing support for resource constrained environments and devices. Their premise is that "human attention" is a limited resource, and that using self-tuning and pro-activity enables ubiquitous applications to lower the demands imposed on users, freeing them to do actual work. Aura extends a Linux-based OS with custom filesystems, resource monitoring processes, and support for remote execution of applications. It also includes *PRISM* – a layer intended to capture user-intent, explicitly representing user tasks, and monitoring context. *Gaia* [147, 46] is similar to Aura in the sense that it is es-

sentially a meta-operating system designed for portable applications executing in so-called "active spaces". An active space is in Gaia defined as a "...physical space coordinated by a responsive context-based software infrastructure..." [148]. Sessions is the concept used to associate users with application and data. Sessions can be moved between active spaces, enabling users to activate and suspend them as necessary. Gaia includes run-time support for distributed applications based on a modified model-view-controller scheme, and includes a context file system which uses the context of e.g. a user to present a personalised and context-based view on files and data in general.

At Stanford University the *Interactive Workspaces Project* [73] is a multi-device and multi-user ubiquitous environment which enables distributed interaction using augmented legacy applications. The project includes a physical construction called the *iRoom* in which touch-screens, desktop machines, and mobile hand-held devices interact through the *iROS* [94] infrastructure. iROS is (similarly to Gaia) described as a meta-operating systems, and has at its core an event-heap. The heap is a tuple-space of name-type-value fields in which applications may subscribe for and post events. User-interfaces for e.g. room controls are automatically generated by iROS producing for instance a web-based PDA application or standard desktop application. The *i-LAND* environment by Streitz et al. [168] also integrates with physical artifacts. It is a collection of software components – "roomware" components – which provide user-interfaces and infrastructure support for interacting with technological artifacts embedded in the environment and in the actual architecture and furniture. This moves i-LAND well beyond the desktop and requires a distributed infrastructure. This infrastructure is called *BEACH*, and handles both distribution of information within shared information spaces and enables distributed collaboration using 'virtual environments' corresponding to physical environments. The lowest level of BEACH handling distributing, replicating and synchronizing data is called *COAST* [156]. It is based on replicated objects, or documents, which are manipulated by views and controllers, again following the model-view-controller paradigm. COAST does automatic event propagation and automatically updates custom views to match changes to the underlying document, freeing the developer from making such relationships with an application explicitly. All modifications to documents are made within transactions, ensuring overall consistency and making the framework suited for synchronized collaboration.

The *Presto* [59] and later the *Placeless Documents* [60] (PD) is similarly a document-centric system, and presents the idea of "active infrastructures"; infrastructures into which application level functionality can be pushed to enable the infrastructure to specialize itself to the needs of different applications. The central data-structure is a "document" which is a contained piece of data. Active properties and delegates are then executable meta-content which can be attached to documents hosted by the infrastructure, expanding its functionality beyond storage and distribution. The system is highly interactive, enabling properties to be attached to documents at runtime by end-users. Two examples of the extra functionality which properties may add, are in delivery of documents and versioning. The delivery service allows documents to be modified before being served to users, for instance by converting from one format to another. Versioning functionality maintains a his-

tory of documents allowing users access to all revisions. The programming model in the aexo infrastructure is application-oriented, linking the internal state of an application with an external representation, while the PD system is data-oriented. However, once the application state has been exported the treatment of data is very similar in both infrastructures. For instance, in the PD system there is a concept of an exploded application, where functionality is "spread" throughout the infrastructure via the active properties. This is *very* similar to the role that the distributed components plays in aexo. Here the functionality is seen as external to data though and as such not as document oriented. The programming model of PD is java-based and needs to be compiled into the system, while aexo provides both dynamically loaded components and the ASPI programming model for external application based on annotations.

The *HP CoolTown* project [99, 23] rely on web-technologies to enable nomadic users carrying e.g. PDAs and mobile-phones access to fixed electronic resources such as printers, kiosks and teller-machines. These resources each run an embedded web-server giving them a local or global web-presence. URL resolution mechanisms are then used to resolve e.g. bar-codes or infra-red beacons into URL addresses for accessing the resources. Once a resource URL has been resolved it is then usable by the device to read from or write information to.

Event-based infrastructures are designed to support the management of subscriptions and event notification and routing. One of the prevalent paradigms for message-based communications is the publish-subscribe [132, 67, 45, 32] mechanism. The *Siena* wide-area notification service [43] is an efficient and scalable implementation of decentralized event-system. It provides a flexible scheme for defining subscriptions including filters to constrain which events are valid for a given subscription, and supports a wide range of network topologies and server architectures. The *JEDI* [52] event-based infrastructure uses a pattern scheme for event subscription where active-objects are the producers and consumers of events. As with Siena, they do not assume a coupling between data and event, but rather rely on the active-objects to define their own internal models. The *Yet ANother Extensible Event Service* [141, 142] (YANCEES) framework is a versatile *meta* publish-subscribe architecture. The framework can be customized with plug-in components to define new subscription and notification models, filters for selecting or restricting event streams for specific subscribers, for adding new protocols and adapters for translating between publish-subscribe models. YANCEES is highly specialized for event-based systems, but is within this domain extremely flexible. In contrast to these systems, event management in aexo is integrated into the data structure and requires no extra effort from the application developers. This is a much simpler (but also more limited in scope) approach than those of generic publish-subscribe systems, and creates a tight coupling between events and data which is not otherwise present.

4.5 Conclusion

This chapter presented an overview infrastructures with different architectures to support distributed interaction and collaboration through activities. The prototypes were based on the three patterns C)-E) in Figure 4.3; a hybrid peer-to-peer/client-server architecture, a fusion of the hybrid architecture and a decentralized ad-hoc architecture based on distributed objects, and lastly a customizable infrastructure in which both functionality and data can be distributed. The work presented in this chapter address *research question II* by showing how support for distributed interaction is implemented in the infrastructure, and how the infrastructure prototypes implement mechanisms for effective distribution of events in both stable and transient environments. Furthermore, it presents a generic mechanism in the aexo system for enabling distributed interaction. An infrastructure, in which interprocess communication is realized through shared models, and the events which originate from modifications to these models. The chapter has thus illustrated how activity storage and distribution, event notifications and concurrency control can be accomplished, optimized and made flexible using different architectural patterns.

Contributions

The contributions are mainly the implementation of the three software infrastructures which provide the underlying functionality for distributed interaction through activities. Their defining characteristic is that they simultaneously embody multiple architectural patterns in order to operate in both stable and transient environments. Paper V holds more detail on the implementation and evaluation of the hybrid scheme as well as the stateless protocol employed. The main contributions here are, firstly, the introduction of concepts of ephemeral and accountable events and a discussion on their design implications. Secondly, the presentation, implementation and evaluation of the hybrid architecture supporting the scheme showing that it does indeed scale to an acceptable number of participants. Paper VI shows how to support hardware infrastructure-less operation using the client-cache and a distributed object scheme. The contributions of this paper are a novel fusion of a hybrid or client-server architecture and a pure decentralized scheme. The scheme has a hybrid architecture in stable environments, using both a central-server and a peer-to-peer mode of communication, but does not rely on a central server when used in e.g. transient environments. Paper VII presents the aexo infrastructure. The contributions are the implementation of aexo as well as an accompanying programming model which is presented in chapter 5. The paper furthermore gives two proof-of-concept deployments which demonstrate the utility of the infrastructure. One of these is presented in detail in Paper III. The novelty of aexo lies mainly in its flexibility and its approach to the distribution of functionality on multiple hosts in the infrastructure.

Chapter 5

Composition & Distribution of Stateful Applications

The concept of *application state* is central to activity-based computing. The state represents a snapshot in time of an application's internal structure. Extracting this state, storing and managing it in the infrastructure is the central infrastructural mechanism in activity-based computing. We denote state stored in the infrastructure as *external*. The externally stored state and the internal state of the application are then continuously kept synchronized essentially making the stored state behave as an external model of the application. The state is used when resuming a previously suspended activity, returning the state of the computer to when the user last worked with the activity. The externalization of state also exposes the application to outside manipulation, allowing processes to inspect and manipulate the application through its exported state. State-sharing is thus a mechanism to enable interprocess communication and awareness as described in Section 2.2. Externalization of state relies on two basic assumptions. First, that it is in fact possible to *represent* the state of applications in a manageable format, and second, that it is actually possible to extract (and *manage*) that state from the running application. The first assumption is meta-physical in the sense that one has to approach it through a discussion on whether it is even sensible to talk about application state, as something which can be represented outside the application. The second assumption is more technical in nature and calls for generic mechanisms to externalize state and subsequently manage it, synchronizing the internal and external state of applications. This chapter is concerned with both these issues, and describes the mechanisms presented in Paper VII to define and externalize the state of applications.

Representations of Application State

The concrete structure of the state of any given application is in some sense unique to that specific application. A word-processor would have a different definition of its state than a web-browser for instance. Despite differences in state, however, it

is still possible to extrapolate the state of a wide range of applications and have that state or model represented externally. Actually, most applications already have a very clear definition of their state – represented by the data in the files read and written by the application. Saving a file can be seen as externalizing a portion of the application's state, while opening the file again can be seen as a mechanism, that subsequently returns the application to the state it was in, when the file was last saved. I will therefore argue that a great deal of the hard work is already done, most applications already have the ability to externalize a large part of their state into files. From this perspective data and files are pure state, and the remaining state information needed to restore an application to a state, that is indistinguishable from when it was last used, mostly involves navigating within files, i.e. scrolling to a specific location in a webpage or a text-document, or has to do with the visual appearance of the application. These are state elements which are tightly coupled with the capabilities of the actual application, and as such do not belong in the files or data accessed by application. However, one could argue that they should be saved alongside the actual data forming a complete state serialization but in separate files. This state data is then something which is personal to the user accessing the content and simultaneously application-specific.

The approach taken in this dissertation is to store the application state in the infrastructure, leaving the files and data accessed by the application in place, while others have looked at ways to include data in activities [111, 124]. This state is represented as a hierarchical data-structure in the infrastructure serializable as XML. This structure represent the hierarchy of Figure 2.2 on page 16, which illustrates that the complete activity-state represented by an activity is composed a number of application or "service" states which again consist of "component" states. An example is given in Listing 5.1, where the two main components of the activity is seen as the `meta` (ln 2) and the `state` (ln 9) section. The meta-section contains information (as shown here) about participants, but also resources attached to the activity (screenshots, files, etc). The state-section contains the actual state of the activity. This state is composed of `services` which, when instantiated on client machines, are translated to application instances, as well as `components` which are the independent pieces of information which combined form the complete state of the application.

```
1  <activity id="1" name="MyActivity" status="000" type="111" creator="jbp">
2    <meta>
3      ...
4      <participants>
5        <person id="23"/>
6      </participants>
7      ...
8    </meta>
9    <state activity="88">
10     <service id="123" type="browser">
11       <component id="url">
12         <location url="http://dr.dk/" bookmark="#editor" />
13       </component>
14       ...
15     </service>
```

```
16      ...
17    </state>
18  </activity>
```

Listing 5.1 The state of an activity serialized as XML.

The activity is from the data perspective thus a container of state, essentially a wrapper around the state representation of multiple applications. The methods for extracting the runtime state from applications is very much dependent on the specific application, and how open to introspection the application is.

Extracting Application State

The applications of activity-based computing presented in chapter 3 and chapter 4 rely on the ability to access state information for running applications. We label applications, which in this manner have their internal state externalized, as *stateful* [17], and differentiate between three types of stateful applications:

Legacy This type includes applications with no externally accessible interface and no way of getting the internal state of the application. Only the information that we can extract directly through the operating system, is included in the overall activity-state. For Windows XP and Vista, for instance, this is the application's name, window size and position, and the executable file. This means that when such an application is moved from one device to another, the application window is restored, but not the content or internal state.

Scriptable Here, access to the internal state of an application is given in the form of an API or some other form of externally accessible programming interface. In order to integrate the application and its state, special-purpose "application wrappers" are created. An example is the Microsoft Internet Explorer application which provides a COM [118] interface, which is used to get access to the currently displayed URL and the scroll location on the page. Using this interface, the state of the application window, as well as its content, can be restored when the activity is resumed.

Customizable This includes all applications where it is possible to access the source code. Here, the applications themselves can be customized to make their state available for activities. It is the support for open sourced .NET applications that the tools presented in this chapter are concerned with. This support is in the form of annotations, which enable developers to specify exactly which parts of their application represents the state to be made public. The libraries which include the annotations also provide the run-time wrapping of applications required to make them interface with the infrastructure, and enable users to include the application in their interaction with activities.

Legacy applications are difficult to integrate with for a number of reasons. First of all, getting access to any useful internal state is painstakingly hard. An example

is the Windows XP Notepad application. Here it is actually possible to get the text shown in the application through a series of win32 system calls, but the method is very fragile. The second reason to leave legacy applications alone is that the procedure, by which internal state might be extracted, is very likely to break with any application update. Thus, defaulting to the lowest common denominator provided by the operating system, e.g. the window state, is the most reasonable course of action.

For scriptable applications, the burden of integration is still high. The system, by which the applications are queried for their state, need to know how to do this for each individual application, and custom wrappers must be made for all applications. Minor application updates are not a problem, as long as external APIs remain the same, however the risk of updates breaking the wrapper-functionality is still there.

Customizable applications on the other hand can be more richly integrated as long as a suitable scheme to do so is made available for the application developers. In the following the focus will be on *customizable* applications and on how to enable developers to *annotate* properties in .NET based applications for specific use with the aexo infrastructure of Section 4.3. The combined value of all annotated properties in an application then make out the state of the application, which is extracted to be stored and manipulated in the infrastructure.

The following sections introduce the annotation scheme – the programming model for making applications stateful – and the concept of a *distributed application space* in which applications, by exporting state and relying on functionality already present in the infrastructure, can be included in activities and take part in distributed interaction within shared activities.

5.1 Stateful Applications

A stateful application is, as previously mentioned, an application that is capable of interacting with its software environment by exporting its *internal state*. The environment, or the software infrastructure in which it runs, may query for certain components of its internal state, or post changes to modify it. The application itself may query or change the state of the environment in return. These are the core mechanisms of synchronizing the state of an application. The capability of exporting state is used in activity-based computing to *bind* the internal state of an application into an activity which is then stored and manipulated through the software infrastructure and the user-interface of the activity-based system. As an activity is suspended the state of the application is persisted in the activity and when resuming the activity the state is posted back to the application once more restoring the state of the application.

The support for these operations conceptually "wraps" the application as illustrated in Figure 5.1. It consists of three software components. The first is the scheme which takes the local application state-representation, which is here assumed to be in the form of objects and their properties or public field variables,

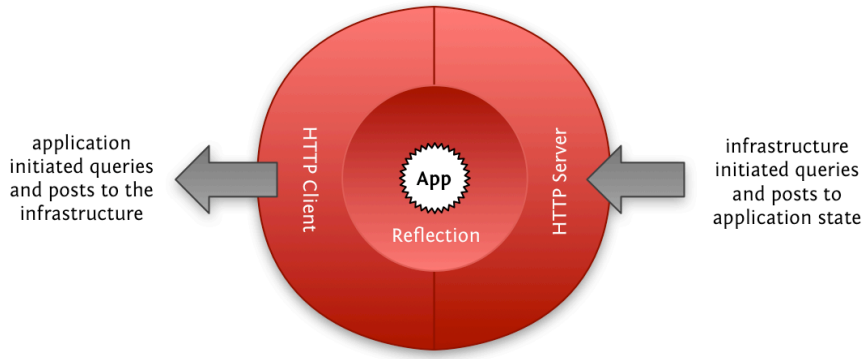


Figure 5.1 The encapsulating functionality required to make an application "stateful". In this figure HTTP is the protocol with which the application synchronizes its state. This figure is a magnification of an application in Figure 2.3.

and maps it to an external structure represented in the software infrastructure. This functionality relies on *reflection* to discover and bind properties. The next two components are the protocols for keeping the local structure synchronized with the external representation. Here, the HTTP protocol is used, but any protocol supported in the infrastructure may be used. An HTTP client and server is then embedded in the application. This also has the side-effect, that the state of a stateful application can be accessed from any other application, e.g. a browser, as long as the port which the application's HTTP server has been bound to is known by that application. The action of mapping and exporting the internal structure of the application is done on the level of properties using an annotation scheme.

Annotations

The scheme for exporting application state using annotations is called ASPI – short for *application space programming interface*. The hierarchical nature of the core data-structure of aexo, the hmap of 4.3 on page 55, lends itself easily to this scheme, since live run-time objects and their properties form graphs which can be represented directly in the map. An example of how a binding may be constructed is given in Figure 5.2 and Listing 5.2. The binding here is of a submap holding collections of photos to a `List` of `PhotoCollection` objects.

The `Reflective` annotation shows how the binding is set up from the point-of-view of the application developer. This annotation can also be seen as the contract or the interface which others can use to interact with the application with, since changes in the external state can affect the local state. Here the internal and external states are synchronized by using the `TwoWay` synchronization definition.

The annotation scheme lets developers specify a number of properties on the binding, the example above has values for all options, but only the path, the first argument, is required. This simple annotation allows the application developer a great deal of control with how the externalization of state should take place,

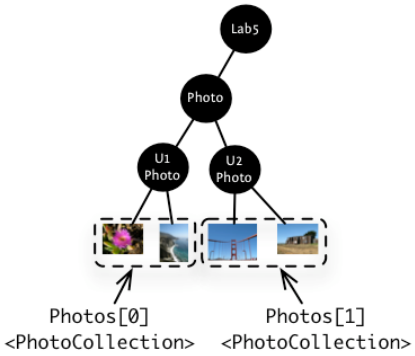


Figure 5.2 The external representation of collections of photos.

```
1  // DisplayPhotoModel is the ASPI for the public display model
2  public class DisplayPhotoModel {
3     [Reflective("local://Lab5/Photo/U{list._counter}Photo",
4                 Detection=Automatic,
5                 Binding=OnEnable,
6                 Recursion=AllAttributed,
7                 Synchronization=TwoWay})]
8     public List<PhotoCollection> Photos
9     {
10        get; set;
11    }
12 }
```

Listing 5.2 An example of property annotations for photo-display application.

while simultaneously not forcing the developer to explicitly provide values for all options. Paths can contain certain dynamic components, e.g. the path `{bonjour._http._aexo_}/Lab5` will be resolved to the `/Lab5` path in the map which advertises its presence using Bonjour under the `_http._aexo` service type. Paths are resolved using one of the built-in path-resolvers. Among these are also resolvers which can traverse runtime objects. This is useful in expanding paths, used e.g. for recursive data-types or list-based data-types. For instance, the `{list._counter}` expression (in ln 3) is used to map multiple nodes, matching the path to a local list. When the `Photos` property is bound to a map, the nodes in the map which match the path are bound to elements in a local list, e.g. `Lab5/Photo/U1Photo` is bound to the second element in the list, and as new nodes appear in the map, the list is expanded with new elements and vice versa. The optional *detection*-property (ln 4) decides when local changes are written to the infrastructure. By setting the detection to `Manual` for instance, the application may delay writing local changes to the infrastructure to a later time. This can be used to e.g. bundle updates to minimize connection time. Determining when the binding to the infrastructure is to take place is done via the *binding*-property (ln 5). The options are to let the developer manually bind the object or to continuously try to bind it. The latter

will try to keep the object bound to the infrastructure, and could be used to e.g. host objects on a mobile device. When the device moves between networks the object will be bound to the different aexo instances that are available, thus broadcasting the presence of the (application on the) device. The *recursion* property (ln 6) denotes if and how the local object-graph (starting with the current object and following all reference type objects) should be traversed and bound onto the infrastructure. Currently, there are three options. `AllAttributed` indicates that all public properties which are annotated with the `Reflective` attribute are automatically bound. The `AllPublic` options includes all public properties. `None` only maps the current object, i.e. it does not traverse the object-graph beyond `this` object. There are three modes of *synchronization* (ln 7). First, `TwoWay` keeps the map and local object in total synchronization. It is the state in the infrastructure which is considered the master and conflicts are resolved by using this state. By using `AppToMap` the infrastructure state is considered read-only and any changes made in it will not affect the local model. The reverse is the case with the `MapToApp` option.

Using the annotation scheme, application developers can define how the internal state of the application should be represented externally, essentially defining how the externally accessible "API" of the application should appear. This can be used within activity-based computing to incorporate application state into activities, but is also more widely applicable as a sort of *remote reflection* for distributed applications. The following section describes how to utilize the aexo framework and the annotation scheme to build distributed applications using concepts from the model-view-controller pattern, and introduces the concept of a *distributed application space*.

5.2 Distributed Application Spaces

We define a *distributed application space* (DAS) in Paper VII as a set of software entities or applications distributed across several networked devices, which together form one logical end-user computer system. A DAS typically includes end-user *views* as graphical user interfaces, *controllers* that implements the application logic, and *models* holding application data. The combined functionality of a DAS is thus the sum of the components running in the total set of available devices, and the functionality may hence vary dynamically according to the number of devices and components participating at a given point of time. This section demonstrates how such distribution of model, view and controller-functionality is supported by the aexo software infrastructure.

The programming model for building a DAS is based on the model-view-controller pattern. The following describes which roles models, controllers and views have in application spaces.

Distributing Models, Controllers and Views

The aexo infrastructure has a very flexible approach to distributed application composition and how it allows distribution and manipulation of models, views and controllers respectively.

In the context of aexo a *model* is the external data as it is represented in the hierarchical map. The distribution of the map, the model, can be done on a submap basis. This means that when looking at the complete data available in a distributed application space, then there is support for any arbitrary submap to be distributed to any given device running an instance of the infrastructure. Exporting an otherwise internal application model to infrastructure in this manner can be seen as a way of "reversing" the structure of the application, exposing its core object model for other applications to see and change. This also enforces a strict decoupling of applications: *Any* entity can provide the information which becomes the internal state of the application. As soon as a data-structure has been bound in the map, its components, the nodes in the map can thus be made to form relationships with other nodes, representing components in other data-structures (which are possibly bound to applications running on other machines). This cross-cutting of models enable complex relationships to be made in the models, that are shared in a given space.

The functionality of a typical *controller* is provided in aexo by a combination of the controller-components (see Figure 4.7) loaded by the infrastructure and the external applications themselves. Controllers essentially implement the business logic of an application space. This task is accomplished by controller-components whose task it is to maintain and organize certain parts of the map based on data and events from certain other parts. That is, to define the dynamic relationships between the data stored in the map and to provide coherent interfaces to data from otherwise disparate data-sources. An example of this is given in 4.3. Since components can be dynamically loaded, the logic represented by the controller-components will then be logic that is specific for a given infrastructure configuration, tailored to the physical context of the system. This means that as applications enter an environment backed by an aexo infrastructure instance, they are automatically augmented with the controller-logic loaded by it, essentially *reconfiguring* them for the space in which they now execute. Controller-logic may also be implemented in the external applications themselves – this logic however, migrates with the application and should thus constitute functionality unique to the specific application. The annotation scheme of Section 5.1 is part of the local application controller logic, deciding when and how information is synchronized between the local application and the infrastructure. The total controller-logic available in a DAS therefore depends on which controller-components are loaded, but also which partitions are currently present.

Views are the interfaces which end-users interact with. Mostly, they are represented in the external stand-alone applications and linked to data and controller-logic in the infrastructure using the annotation scheme, but they may also be implemented in components which are loaded as part of the infrastructure. The former

allows transient views while the latter may be utilized to build mainly HTML-based views which are specific to a given environment and infrastructure setup. An example is an RSS component (see Figure 4.8 on page 58) which provides a chronological view on modifications on a given submap.

This notion of composing applications of distributed components, and especially the concept of exporting application state, is, as previously mentioned, used within activity-based computing to integrate applications into activities. Furthermore, it represents an initial step towards supporting "fragmented" activities where interaction within an activity is distributed on and spans multiple devices. An example illustrating the distribution of models, views and controllers can be found in Paper VII.

5.3 Related Work

The related work for this chapter includes programming models and other application composition mechanisms of distributed infrastructures as well as the architectures supporting distributed interaction. Taylor et al. [173] presents a message-based architectural style for graphical user-interface components. Components are software entities which may be written in different programming languages, running concurrently and distributed on multiple machines. The composition mechanism enables developers to construct a hierarchical network of concurrent components hooked together by message routing devices. The hierarchical organization is among other things used to enforce awareness rules; a component is only aware of its descendants but may receive requests from its ancestors. This is similar to the structuring mechanisms of ambient calculus [42] and the agent composition mechanisms of Satoh [153]. The organization of components is also, as is the case with aexo, a determining factor for which notifications a component may emit and which requests it may receive. The ideas behind the architectural style presented by Taylor et al. [173] are quite similar to the ones implemented in the aexo infrastructure, however aexo cannot strictly be modelled using the style, because it relies on relationships within the data stored in hmap and as such does not enforce any compositional rules within external components or external applications.

The active spaces of Gaia [149] clearly resemble our definition of a distributed application space. Active spaces are virtual locations that host the execution of applications, which themselves may be composed of distributed components. The components are then managed as distributed objects, but are not automatically connected to each other. Our approach is more lightweight, enabling distributed applications or views to "export" model and controller functionality to the space, and does not require the middleware to actually execute on the hosts running the views. A distributed application space has data sharing as its default and does not require applications to explicitly create bindings to communicate. The main differences between aexo and Gaia though are in the approach to application composition. Here, Gaia requires services to conform to specific application-types (context-services, presence-services etc) while the informal data structure of aexo

encourages a more dynamic approach. The architecture of aexo is also more lightweight, in the sense that protocols, discovery and all but the very basic functionality is dynamically added to the core system and not integrated by default, a structure which carries a greater level of flexibility.

Satoh [153, 126] uses agents in place of components, and describe the *MobileSpaces* system for building adaptive distributed applications. Agents, from which applications are constructed, can be dynamically combined to form new functionality. The composition of agents is hierarchical and takes place by having one software-agent virtually enter another. All functionality is implemented in agents analogously to the components in aexo, but in MobileSpaces agents are self-contained software entities which can migrate between machines distributing functionality more dynamically than it is possible with aexo. The hierarchical composition of functional units – the agents – is similar to how data is structured in aexo, but there is no similar functionality to compose applications, instead the data and events within it, becomes the core medium for composition of complex functionality. A similar system for application composition is the *PCOM* [29] framework, which exports a toolchain to build dynamic and customizable distributed applications from singular components. Both PCOM and MobileSpaces enables great flexibility in application composition, but does not deal with data distribution, cross-application events or truly distributed applications. Another framework which is based on a hierarchical compositing mechanism is the *one.world* framework [81]. The aim of this system is to support adaptable applications by "exposing contextual change and encouraging ad-hoc composition of services and to facilitate sharing between devices and applications". Entities called "environments" are the central structuring mechanism for services here. Each device has one root environment in which users and services are nested. Nesting is the primary action for application composition, as it is in the MobileSpaces project, and an application gains new functionality by having a sub-service nested within it. The application domain of the one.world framework is a biology-lab in which both application state and data advantageously are migrated between machines.

Bardram et al. [17, 21] describes a framework for activity-based computing in which stateful applications can be built from custom Swing components. For instance the state of a scrollbar would be its position while the state of a textfield its contents. Using these components developers can build applications which report the combined state of the user-interface widgets they are composed of, plus any additional internal application state. The concept of activity-state found in this dissertation stems from this work, but are generalized to non-java applications and has a more elaborate scheme for defining state in the annotation scheme, which includes properties to define values for e.g. synchronicity and binding-time.

Edwards et al. [63, 64] address the need for interoperability between heterogenous devices within ubiquitous computing. Their approach towards this form of serendipitous interoperability is called "recombinant computing" and aims at supporting the arbitrary combination of devices and applications in line with the original thoughts on ubiquitous computing by Weiser [184]. Their architecture is called *Speakeasy* and relies on three premises: a small set of fixed generic interfaces under-

stood by all processes, mobile code to allow processes to acquire new functionality or new protocols with which to communicate, and keeping users in the loop enabling them to control the interaction and the interoperability. Speakeasy services, called components, implement a basic set of protocols, but has inflection points in which mobile code can be inserted extending the functionality of the component, e.g. a streaming end-point can be transferred to a component such that it can listen in on a video-stream. User-interfaces can also be added dynamically to components as a mechanism for keeping users in the loop. Applications can thus be created or modified at run-time, and applications require no (or very limited) a priori knowledge of each other. Examples of applications built on Speakeasy is a ticker-tape and a screen-saver application [39] displaying media from a public "soup" of items.

5.4 Conclusion

This chapter described an interface between applications and the aexo software infrastructure. An annotation scheme supports the creation of customized applications where the basic components are a reflection annotation-discovery component and a communication end-point, an HTTP-client and server. Using this scheme on the aexo software infrastructure allows applications to export their internal state and to utilize the controller functionality loaded in the infrastructure in the form of controller-components. This enables applications to take part in distributed application spaces whose models, views, and controllers are distributed on multiple machines, and is a novel and expressive way of defining the interface between an infrastructure and the applications supported by it. The work address *research question III* by defining a method by which application developers can choose which parts (properties) of their application defines the state of the application, and how and when the state must be synchronized with an infrastructure component. It shows how to *bind* the internal application state to activities, but also represents a more generic approach to *remote reflection*. In essence, the aexo system is an enabler or meta-infrastructure for building and experimenting with concrete implementations of infrastructures – the swiss army knife of infrastructures for distributed interaction.

Contributions

The annotation scheme and the concept of a *distributed application space* is presented in Paper VII. The contribution is the use of annotations as an expressive mechanism used by application developers to bind internal state to the distributed data-structure represented in the software infrastructure. This enables the infrastructure to inspect and modify this external representation and to use the data in other contexts, perhaps allowing a part of the state for one application to be shared with another application. Or to use the state of one application as input to a second.

Chapter 6

Conclusion

This chapter concludes Part I of this dissertation. Throughout the dissertation the focus has been on the software elements required to support distributed interaction for activity-based computing. The research areas involved have been within user-interface and -interaction technologies, event-based infrastructures, and technologies for application composition. The contributions presented in chapter 3, 4, and 5 together illustrate the approach of building multiple vertical prototypes which interoperate to form software "environments" in which interaction can be distributed between multiple machines and where the concepts of activity-based computing is operationalized to manage the tasks and information of users. With a birds-eye perspective, this software enables distributed interaction by linking applications within activities and across machines, and provides interaction with activities and distributed applications through a range of user-interfaces, essentially realizing the structures and connections of Figure 2.3.

6.1 Addressing the Research Questions

The three research questions of this dissertation are linked by the concepts of distributed interaction and activity-based computing, and only by considering all three above mentioned areas a sensible approach to addressing the overall research question of how to realize distributed interaction in activity-based computing is found.

The topic of research question **I** was the presentation and manipulation of distributed activities by end-users. The question has been addressed by constructing user-interfaces displaying multiple perspectives on activities. One perspective lets users focus on a single application, while others show complete activities, either sequentially; by letting users switch between activities, or concurrently; by giving glanceable overviews of multiple activities. The systems supports the notion of peripheral awareness by laying out the applications of an activity on an infinite 2d surface such that participants in the same activity may appropriate areas where

they can work in "their own" applications, but still get an idea of what others are working on. The work presented in chapter 3 more generally demonstrates the application of activity-based computing as an abstraction on files and applications. It shows how a usable interface for manipulating activities can be applied to both desktop-based work as well as in multi-display environments such as those found in wearable computing or in hospitals.

Research question **II** revolved around the underlying infrastructural elements required to distribute and manage end-user interaction with computational activities. The approach was two-fold: first, looking at the architectures to ensure performance, scalability and to enable users to move between heterogenous environments without severe loss of functionality, and second, to build a flexible *meta*-infrastructure capable of being deployed in and adapted to many diverse environments. "Flexibility" was the keyword here, and it was accomplished using a micro-kernel approach as well as relying on many interrelated components to define the functionality of a given infrastructure deployment. Chapter 4 demonstrates that it is feasible to build a relatively simple infrastructure for activity-based computing using a hybrid architecture, which supports the principles of *suspend and resume*, *roaming*, and *sharing* of activities. This allows both asynchronous and synchronous collaboration to take place using activities. The flexible *meta*-infrastructure is demonstrated as a vessel to explore more distributed and loosely connected deployments of activity-based computing.

The interaction and state-sharing between applications either mitigated through infrastructure-bound activities or directly through other models in the infrastructure, was the focus of research question **III**. The use of the mechanism to externalize application state, the annotation scheme, is not confined within activity-based computing and, as is the case with the *meta*-infrastructure, it can be used as a more generic remote reflection method. Used together with a distributed infrastructure, however, it becomes less arduous to implement many of the concepts of activity-based computing, and furthermore opens applications up to external introspection and modification. The notion of a distributed application space conceptualizes this idea by using a shared infrastructure to represent the combined model of a space, and dynamically loaded components to implement its logic. The work in chapter 5 abstracts away communication protocols and lets developers concentrate on the semantics of the synchronization of state. It does not replace interprocess communication mechanisms, but is within the realm of activity-based computing a simple method to integrate applications.

The overall research question of how to provide computational support for distributed interaction in activity-based computing, has been addressed by a three-way approach as described above. This division separates the concerns of each area, but they are ultimately related by overall functional and non-functional goals. The idea of a *meta*-infrastructure makes sense in this context by virtue of it being easy to adapt to new and changing requirements, but also to enable the concepts and prototypes in each of the three research areas to evolve independently. The interplay between the interaction mechanisms of the user-interface, a distributed infrastructure, and application integration has enabled us to explore areas of activity-based

computing which would have been hard to reach with a narrower focus. The current state of the technologies in this dissertation enables us to begin looking at more advanced (and possibly more distributed) user-interfaces and to begin experimenting more with modelling, runtime management, and distribution of activities on the infrastructure level.

6.2 Future Work

In a true iterative fashion of Section 1.4, the most pressing future work for the meta-infrastructure is to create an activity-specific deployment, allowing us to begin to model and manage more complex distributed activities and their relationships explicitly. Among the things we would like to be able to do is to support fragmented activities and to put more structure on activities moving them in a workflow-direction.

Fragmentation

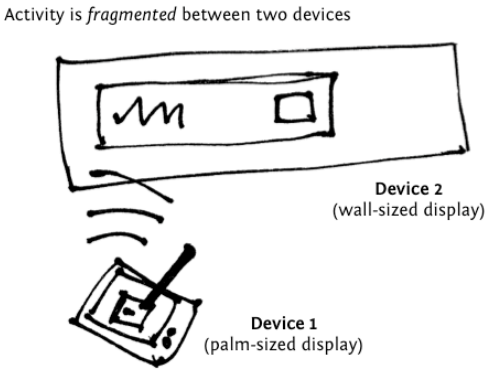


Figure 6.1 An illustration of a simple fragmented activity consisting of a remote control application and a presentation display.

The idea of fragmented activities such that applications within one activity may span multiple devices as briefly mentioned in 2.2 seems logical as a next step. The idea is basically letting a single activity span multiple devices and displays, such that the applications which are part of the activity executes on different machines. This would allow activities to be used as a concept for managing interaction with multiple displays and devices contained in a single room, e.g. an operating-room. Inducing a state change through an application running on one display then is a change to the combined state of the activity running in the room. The novelty of this idea rests on the use of activities to manage the computational resources in the confined locations. Here, infrastructure support would include the ability to distribute individual applications. Something, which the aexo infrastructure would be suited to implement. However, more work is required to allow for the

management of activities which are fragmented across devices in this manner. An initial foray into this topic is presented in Section 5.2, but the focus there is on the programming model for the distributed applications.

Lightweight Workflows

Activities are treated as data and data-containers in this dissertation, however we also talk of activities in terms of "executing an activity" or "suspending the activity". This suggests a dynamic nature of activities which are not being realized by the current infrastructures (or user-interfaces). A future direction is looking at the modelling of activities and how this modelling could affect or be affected by instrumenting activities with workflow-like capabilities, i.e. allowing activities to dynamically evolve. This runtime behaviour of activities could be anything from simple transforms of the structure to actually compiling descriptions of activities into some *runnable* form.

External Applications

The future work with the annotation scheme and the construction of distributed application spaces involves some form of evaluation of their applicability and utility. Such an evaluation would involve getting external developers to build distributed systems using the annotations. An option here within the activity-based computing project is to let the commercial partners, whose applications we aim to integrate, use the annotation scheme. The effort required in doing so would then be an evaluation measure. Another approach recently surfaced is the use of an output redirection mechanism as e.g. *citrix*, where the idea is to have a main-frame to host applications whose views are then transported to wherever an activity is resumed. This would require minimal effort from external application developers. We are currently looking at integrating this approach with the aexo infrastructure in an activity-specific deployment.

Yet another area of future work concerns privacy and access control – issues which have not been discussed in this dissertation. The annotation scheme essentially exposes the application to outside manipulation, and it is therefore crucial to ensure some form of access control. One possible option would be to encode access control properties into the annotations. This would constitute a fine-grained mechanism (on property level) for securing access to internal application state. The infrastructure would have to enforce these rules as well, and the user-interface for interacting with activities would need to provide some mechanism for individuals to authenticate themselves.

Part II

Papers

Paper I

Support for Activity-Based Computing in a Personal Computing Operating System

Jakob E. Bardram Jonathan Bunde-Pedersen Mads Søgaard

Abstract

Research has shown that computers are notoriously bad at supporting the management of parallel activities and interruptions, and that mobility increases the severity and scope of these problems. This paper presents *activity-based computing* (ABC) which supplements the prevalent data- and application-oriented computing paradigm with technologies for handling multiple, parallel and mobile work activities. We present the design and implementation of ABC support embedded in the Windows XP operating system. This includes replacing the Windows Taskbar with an Activity Bar, support for handling Windows applications, a zoomable user interface, and support for moving activities across different computers. We report an evaluation of this Windows XP ABC system which is based on a multi-method approach, where perceived ease-of-use and usefulness was evaluated together with rich interview material. This evaluation showed that users found the ABC XP extension easy to use and likely to be useful in their own work.

1 Introduction

In 1983 Bannon et al. argued that "current human-computer interfaces provide little support for the kinds of problems users encounter when attempting to accomplish several different tasks in a single session" [12, p. 54]. Today, 20 years later, these

Published as: Jakob E. Bardram, Jonathan Bunde-Pedersen, and Mads Soegaard. Support for activity-based computing in a personal computing operating system. In *CHI'06: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 2006. ACMPress.

words seem valid still. Contemporary studies have shown that there is a significant mental and manual overhead associated with the handling of parallel work and interruptions [53, 114, 145, 77, 3]. Studies of non-office work, like clinical work in hospitals, similarly show how desktop computers do a poor job in supporting mobile users' parallel work activities, distributed in time and space [17, 20].

Generally speaking, contemporary computer technology is designed according to an application- and document-centered model. This model enables users to work with specific, targeted applications that support the manipulation of particular kinds of information and performing specific tasks, like writing a letter or making a budget. This model is deeply embedded in the hardware, operating systems and user interface software, as well as the development frameworks available today. It has proven well-suited for office work at a desktop, but the personal and task-oriented approach provides little support for the aggregation of resources and tools required in carrying out higher-level activities. It is left to the user to aggregate such resources and tools in meaningful bundles according to the activity at hand, and manual reconfiguration of this aggregation is often required when multi-tasking between parallel activities.

In our research, we have seen how these problems are highly exacerbated when moving out of the office and into a mobile and collaborative working environment like a hospital. Mobile and nomadic work amplify the reconfiguration overhead when users move from one work context to another, potentially using different computers and different types of devices. Thus, mobility introduces yet another obstacle for suspending and resuming activities, since a user's activities are 'tied' to his or her personal computer.

To meet these challenges, we are pursuing the concept of activity-based computing (ABC). In activity-based computing, the basic computational unit is no longer the file (e.g. a document) or the application (e.g. MS Word) but the *activity of a user*. The end-user is directly supported by computational activities which can be initiated, suspended, stored, and resumed on any computing device in the infrastructure at any point in time, handed over to other persons, or shared among several persons. Furthermore, the execution of activities is adapted to the usage context of the users, i.e. making activities context-aware. This paper reports our approach to activity-based computing with special focus on the user experience. It presents the design, implementation, and evaluation of a user interface software technology for activity-based computing. The implementation is done as an extension of Windows XP, and illustrates how the principles of activity-based computing can be incorporated into an existing operating system (OS).

2 Related Work

The original Rooms system [10] was the first *virtual desktop management system* which allowed users to organize application windows in different 'rooms' associated with different tasks. A more recent version of this principle has been presented in the GroupBar system by Microsoft [163]. In addition to virtual desktop management

systems, a number of research projects have proposed solutions for task management. These include support for moving groups of windows to the desktop periphery in the Scalable Fabric system [145], extending the user's desktop with additional screen space [24] or peripheral displays [109], employing 3D in the TaskGallery window management [143], using time as the main organizing principle in task management [139], or to allow for hierarchical window organization and elastic stretch and resize of windows in the Elastic Windows system [96].

Compared to this work on new user interface metaphors for task management, our work is distinct in at least three ways: Firstly, activities in ABC are *persistent* and *stateful* which means that state is preserved across service restart or computer shutdown. This is a fundamental difference, because it illustrates that activity-based computing is not designed to support grouping of application windows in convenient ways, but to support long lived human activities which unfold and evolve over time. Secondly, activities are *distributed* across networked computers and thereby support users to move their work activities with them while roaming between devices. The Gaia architecture for Smart Spaces [86] also supports applications to move across different Windows XP computers, but there is no support for task or activity management. Finally, we have explored design for activity-centered window management that do not replace the entire PC desktop with a new metaphor, but rather adhere to the same conceptual and physical window management as the underlying OS. In line with the conceptual idea of activity-based computing, the ABC extension to Windows XP 'merely' adds another level of user interface mechanisms on the activity level in terms of the activity bar, the Ctrl+Tab switching, and the activity sharing support. Hence, activity-based computing, and its supporting technology, seeks to extend – and not replace – the prevalent application and document view.

Other systems have been designed to provide more direct support for managing of multiple concurrent activities associated with large amounts of digital material. In *task-centered communication systems*, recording the history of the task, and the virtual workspace associated with it, is done by creating contexts out of message threads. Email communication threads are stored and reused in TaskMaster [31] and activity threads comprised of all accessed data objects are stored and maintained in the Activity Explorer prototype [177, 125]. These systems accurately reflect history of a project (or task) and the various data objects used in it; contacts, email, documents, etc. A common feature of such systems is however, that they are tied to the application making the history and do not support arbitrary use of the computer. The Activity Explorer, for example, supports 6 predefined types of objects: Message, chat, file, folder, screen shots, and to-do items. Hence, users need to "live"[125, p. 381] in the application and important information (state) and work tasks that do not go through the application are not supported. Our approach differs while we seek to enable activity-based computing support on the operating system level and not on the application level.

In *inference-based activity systems*, the user's interaction with the computer is monitored and recorded in order to make inference about the activity of the user. The UMEA system [97] records information about users' activities when interact-

ing with the computer via monitoring the computer file system, input devices, and running applications. This enables semi-automatic maintenance of the content of project-related pools of documents, URLs, etc. A similar approach is used by the TaskTracer [62] which tries to infer so-called 'task profiles'. The goal of these systems is to help users access records of past activities and quickly restore the historical task context. Our approach to activity-based computing does not support activity inference by monitoring user-interaction. For our purpose the benefit of having semi-automatic maintenance of activity content does not merit the cost of monitoring and inferring. We find that the small overhead of creating, naming an activity, and attaching services to it, does not warrant the use of inference techniques. Instead we are using a context-aware infrastructure to help recognize relevant activities based on the current usage context of the user [47].

3 Activity-Based Computing

Our approach to activity-based computing is rooted in a year-long engagement in the study of, and design for, hospital work with focus on the challenges of handling parallel activities, interruptions, mobility, sharing, collaboration, coordination, and easy access to large amount of physical and digital information [48, 17, 20]. In the analysis of the use of contemporary computer technology in hospital work we have seen a range of critical short-comings which pose fundamental challenges to the design of future computer technology.

Firstly, computers are *application- and data-centered* and provide little support for aggregating sets of related applications or services as well as negligible support for *interruptions* in work. Often users need to manually reconfigure their applications to match new or recurrent tasks. Secondly, computers are designed for *stationary* use at a desktop. This also includes so-called 'mobile devices', like laptops, which are difficult to use without sitting down at a desk. Thirdly, applications run *isolated on homogeneous devices.* Hence, it is very difficult to move a set of applications or services from one computer to another, and even more difficult to move it between different kinds of devices. Fourthly, the 'Personal Computer' with its operating system is made for *single-user tasks.* However, a core aspect of everyday activities – especially in a hospital – is their collaborative nature. A fundamental challenge is therefore to investigate how support for collaboration can be made part of the computing infrastructure. Finally, computers are inherently *insensitive* to the working context of its users. Hence, there is no way in which a computer can take contextual information into consideration in the interaction between human and computer.

However, many studies have shown that these challenges also exist in other kinds of work, including so-called 'information work' taking place in an office environment. Hence, even though our initial research was rooted in a hospital environment, we are currently working on a more general level and is proposing activity-based computing (ABC) as an approach to computing, which focuses on computational support for mobile, collaborative, and distributed activities which

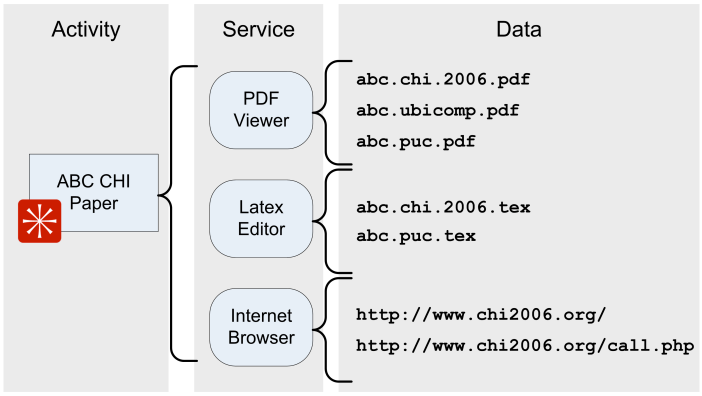


Figure 1 The 'ABC CHI Paper' activity used as an example throughout the paper.

are adapted to their usage context. We are arguing that support for whole activities, rather than individual tasks, is important in pervasive environments. Figure 1 is a conceptual illustration of an *activity* which is a work-related aggregation of *services* and *data*. We have defined activity-based computing around the following essential principles:

Activity-Centered – A 'Computational Activity' collects in a coherent set a range of services and data needed to support a user carrying out some kind of (work) activity. This principle addresses the challenge of application-centered computing.

Activity Suspend and Resume – A user participates in several activities and he or she can alternate between these by suspending one activity and resuming another. Resuming an activity will bring forth all the services and data which are part of the user's activity. This principle addresses the lack of support for interruptions.

Activity Roaming – An activity is stored in an infrastructure (e.g. a server) and can be distributed across a network. Hence, an activity can be suspended on one work-station and resumed on another in a different place. This principle addresses the challenge of mobility.

Activity Adaptation – An activity adapts to the resources available on the device (i.e. computer) on which it is resumed. Such resources are e.g. the network bandwidth, CPU, or display on a given devices. This principle addresses the challenge of isolated and homogeneous devices.

Activity Sharing – An activity is shared among collaborating users. It has a list of participants who can access and manipulate the activity. Consequently, all participants of an activity can resume it and continue the work of another user. Furthermore, if two or more users resume the same activity at the same time on different devices, they will be notified and if their devices support it, they will engage in an on-line, real-time desktop conference. This principle addresses the challenge of collaboration.

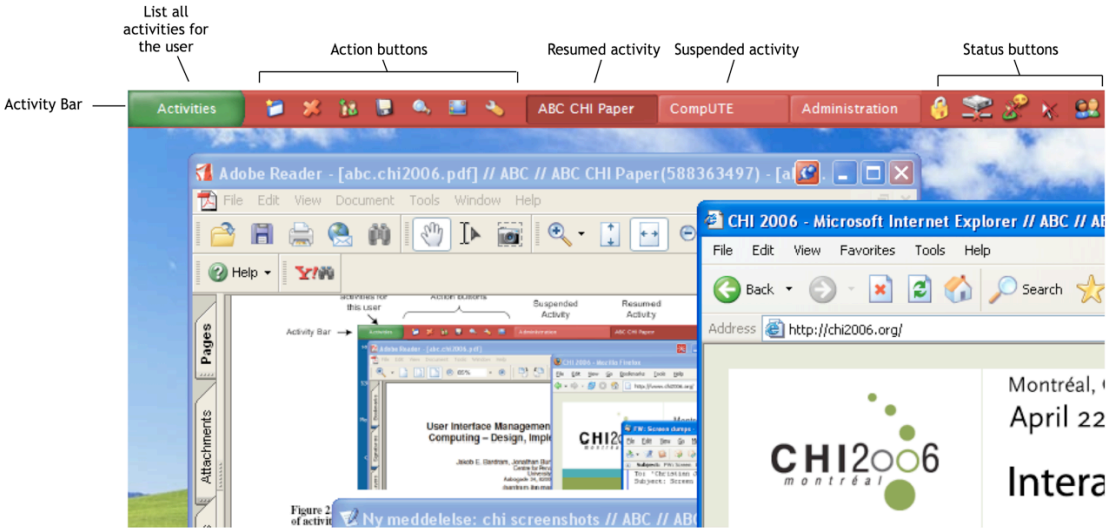


Figure 2 An overall view of the ABC user interface for Windows XP, including the Activity Bar, the current user's list of activities, and different application windows that are part of the resumed activity labeled 'ABC CHI Paper'.

Context-awareness – An activity is context-aware, i.e. it is able to adapt and adjust itself according to its usage context. Context-awareness can be used for adapting the user interface according to the user's current work situation – or it can be used in a more technical sense, where the execution of an activity, and its discovery of services, is adjusted to the resources available in its proximity. This principle addresses the challenge of context insensitivity.

The focus of this paper is to show how activity-based computing has been incorporated in the Windows XP operating system. Hence, the paper will focus specifically on the single user aspects, i.e. support for handling activities, activity suspend/resume, activity roaming, and activity adaptation. Collaborative activity sharing and context awareness has been discussed elsewhere [17, 47].

4 ABC for Windows XP

This section presents the user interface and implementation of the activity-based computing extension to the Windows XP operating system. This ABC user interface is part of the client layer in an overall ABC architecture, where the underlying infrastructure layer is responsible for activity distribution and concurrency control in activity-based collaboration. This paper, however, exclusively focuses on the client layer and its implementation as part of Windows XP.

The ABC user interface for Windows XP is shown is Figure 2. In Windows XP, each service is mapped to an application window. Thus, an activity can be made up of a range of windows, including child windows to an application, where the main window is not part of the activity. In Figure 2 the activity labeled 'ABC CHI Paper'

is resumed and contains windows from different applications like Adobe Reader, Firefox, and an open mail in a child window from Thunderbird. Let us consider the different parts of the ABC user interface for XP in more details, including some of the implementation details.

Activity Bar

The main user interface component is the Activity Bar illustrated in figure 2. This bar replaces the Windows XP Taskbar since activities – and not applications – are the main focus in ABC. In order to facilitate an intuitive understanding of how the bar works, the activity bar is deliberately designed to resemble the Windows Taskbar. The 'Activities' button is used to list the current user's activities as shown in figure 4. The action buttons are used to: (i) Create a new activity; (ii) suspend the current activity; (iii) invite other participants; (iv) save the activity locally; (v) zoom out the activity; (vi) show the ABC control panel; and (vii) to show the radar view. Frequently used activities are shown in the middle part of the bar, and the status icons on the left reveal the collaborative status for the current user: (i) Other online participants; (ii) tele-pointers on/off; (iii) voice-link on/off; (iv) and server online status.

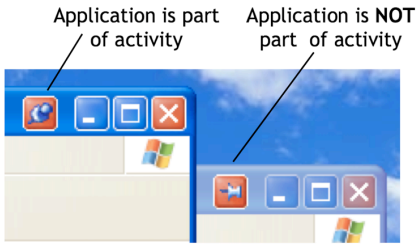


Figure 3 The 'activity icon' for pinning and unpinning an application to an activity.

Application windows are added and removed as services in an activity by using the 'activity icon' in the windows title bar as illustrated in figure 3.

Activity Suspend and Resume

In the ABC user interface, activities can be suspended and resumed in several ways. The typical way is to suspend the current activity and resume another by selecting a new activity in the activity bar or in the activity list illustrated in figure 4. The red cross action button in the activity bar suspends the current activity without resuming another one. This is useful in getting rid of an activity when creating a new one. For easy activity alternation, we have extended Windows XP with a 'Ctrl+Tab' switcher analogue to the Windows 'Alt+Tab' switcher. By pressing 'Ctrl+Tab' a user can quickly switch between his activities. The Windows 'Alt+Tab' switcher is still working and is used to switch between application windows within an activity.

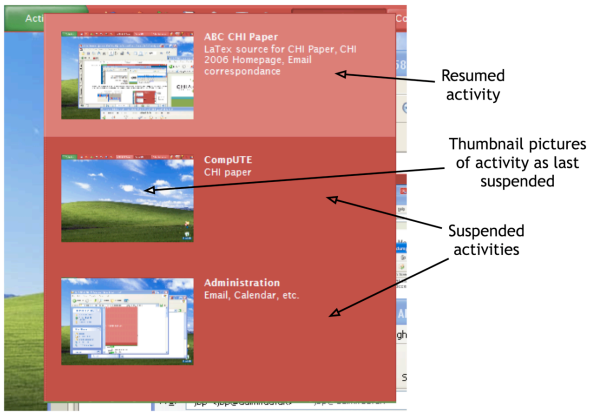


Figure 4 The list of activities for a user displayed when pressing the 'Activities' button.

Technically, all applications can be kept running when the activity is suspended. This is done for two reasons: (i) to enable *concurrent activities* by having services in a suspended activity to run in the background while working on another activity; and (ii) to allow for much faster switching between activities on a local machine, saving the time it takes applications to start up every time an activity is resumed locally. However, applications which belong to suspended activities are removed from the Windows 'Alt+Tab' switcher. Hence, the user is not confused by applications not belonging to the currently resumed activity. When activities are resumed and suspended, activity state information is synchronized with the underlying ABC infrastructure which is responsible for activity roaming and sharing.

Activity UI Adaptation

The ABC user interface for Windows XP is also a zoomable window manager which enables the user to zoom in and out on activities as illustrated in Figure 5. This zoom functionality is used to adapt the user interface to different display sizes and thus supports the principle of activity roaming in the underlying ABC architecture. When an activity is resumed on various Windows XP devices with different screen resolutions, the ABC window manager for XP can zoom the activity to fit the current screen. In Figure 5 the different application windows used in the 'ABC CHI Paper' activity is distributed in space and the activity is zoomed out. The shortcut 'Ctrl+1' is used to toggle between zoom in/out and enables the user to quickly get an overview of an activity and the zoom in on details by clicking on the window. Furthermore, in order to navigate within an activity, there is a radar view as illustrated in Figure 6. This radar resides transparently on top of the application windows and a red square indicates the current viewport. By dragging the red square in the radar view, the current viewport on the Windows XP desktop is adjusted.

The zoom functionality is primarily designed to enable the ABC user interface to support activity adaptation to different devices. Zooming however, also proved

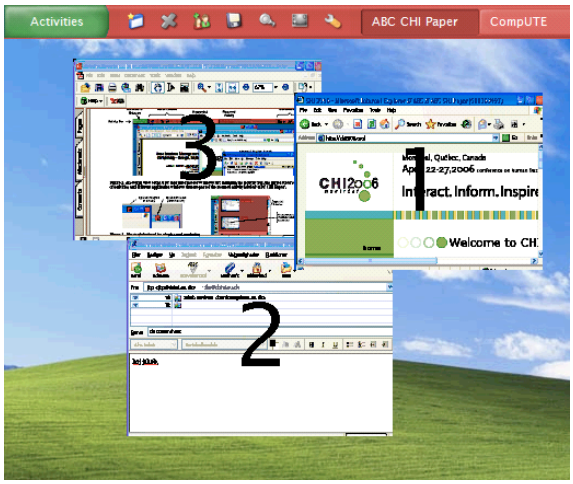


Figure 5 The activity 'ABC CHI Paper' zoomed out. The numbers rendered on top of applications are used to by a voice interface, allowing the user to zoom in on e.g. service no. 1.

useful in single-device activity handling as it helps users manage activities with many windows in a spatial 2D metaphor that conserve the arrangement of windows . In contrast to other zoomable user interfaces, we maintain the window size and position, and do not rearrange windows (like ExposÈ in Mac OS) or introduce new layout metaphors using e.g. 3D [143].

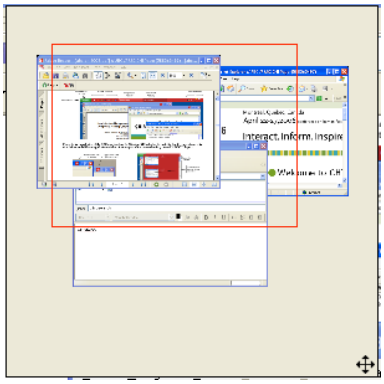


Figure 6 The radar view of an activity.

Activity Roaming

Activity roaming is done via the underlying ABC infrastructure which distributes, manages, and stores activities and their state information. A local activity cache is used if the client is not online (the online status is revealed by the small icon on the right-hand side of the activity bar in Figure 2). When an activity is resumed, the description and the state of the activity is fetched from this distributed infrastructure. On each device, the ABC XP extension runs a small registry which maps services to local applications. For example, the service named `internet_browser`

may be mapped to Mozilla Firefox on one device and to MS Internet Explorer on another. When an activity is resumed on a device, the applications corresponding to the services in the activity are started and their state is restored, including references to relevant data. Similarly, when the user suspends an activity, state information from each application is collected and stored in the activity description, which is then handed over to the ABC infrastructure.

Clearly, activity roaming relies on corresponding distribution mechanisms for the data involved in an activity. It is beyond the scope of this paper to discuss this in great detail, but solutions to data distribution exist. Network file systems, mobile file systems, peer-to-peer file systems, or WebDAV may be used to distribute files, and in a hospital environment, most systems rely on a client-server architecture which handles distribution of the data layer. Furthermore, the ABC infrastructure contains mechanisms for attaching data to an activity, which may be exploited by an application.

Implementation

The software architecture of the Windows XP extension for activity-based computing is illustrated in Figure 7. The 'Activity Controller' is the main component. It provides handles for resuming and suspending an activity, manages connections to the activity infrastructure and to other clients, and has two direct hooks into the OS. Basically it is the point of contact between the infrastructure and the client, as well as between the client and any user-interfaces which may be built on top of it. Hooks are handles used by the client layer to interface with binary components of the OS. The 'desktop hook' interfaces with the windowing system and grabs the thumbnail pictures of an activity just before suspension (see Figure 4). The 'keyhook' listens for the ABC key-combinations, which are 'Ctrl+Tab' for activity switching and 'Ctrl+1' for zooming. Other external components can listen for arbitrary key combinations using this hook.

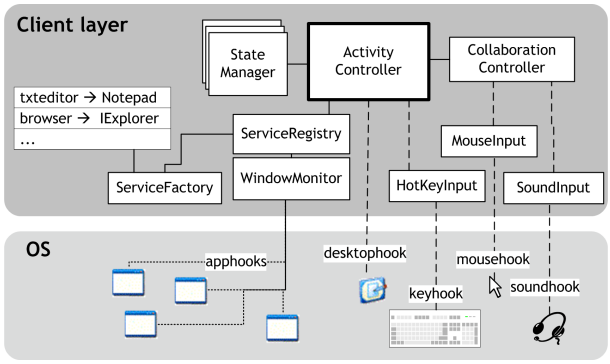


Figure 7 The software architecture of the Windows XP extension for activity-based computing.

The 'Service Registry' has access to the tables which map installed applications to service descriptions in the activity. This allows for adaptation of an activity to locally available applications and for users to define which applications they prefer

for which services. The 'Window Monitor' is responsible for managing running applications in the OS. It publishes events whenever an application window is opened or closed. This is done by polling the OS at intervals for all visible windows. Events are then sent to the registry, which registers changes to the window. The monitor also installs the 'Activity Icon' in the title bar of the windows (see Figure 3). This icon is used for pinning (adding) and unpinning (removing) applications to activities. Whenever an application is added to an activity (by pressing the activity icon), the registry asks the 'Service Factory' for a service which is capable of wrapping the specific type of window. This service is then added to the current activity through the activity controller. Once a service has been added to an activity the state of it is continuously monitored. This happens in the 'State Manager', which tracks changes in the state of all local services. If the activity is shared then a change in its state is immediately sent to all participants via the 'Collaboration Controller'. In this manner, when sharing an activity, the state of all services in the activity is synchronized on all participating devices – giving each participant in the activity an identical view of all applications. The 'soundhook' and 'mousehook' allows the collaboration controller to establish voice links and tele-pointers to collaborating peers.

State Management

Activity suspending/resuming and activity roaming rely on the ability to access state information for running applications. We call such applications *stateful* [17]. In the present Windows XP implementation, we differentiate between three types of stateful applications: (i) legacy applications with no programming API, (ii) legacy application with an API, and (iii) applications with full access to the source code.

In the first type of applications, only the application's name, window size and position, and executable is stored as state information – this is the information that we can extract from the OS. This means that when such an application is moved from one device to another, the application window is restored but not the content. For the second type of applications, special-purpose ABC application wrappers are created. MS Internet Explorer, for example, provides a COM interface which is used to get access to the currently displayed URL and the scroll location on the page. Hence, application window and content can be restored when an activity roam. We have created application wrappers for MS Internet Explorer, MS Powerpoint, MS Word, the Eclipse IDE, and Notepad. These wrappers are quite simple and require little amount of code.

The `StatefulApplication` interface in the ABC programming model is used to develop stateful applications. To help implement state management in the application, fields that represent 'state' information can be annotated as stateful. Figure 8 shows an example of state annotations for an internet browser which ensures that the URL and the scroll position is saved as state information for this service. Stateful applications and their annotated fields are managed automatically by the programming model and adds little extra work to application programming. Our argument is that requiring the programmer to denote stateful fields in his or her pro-

```
[StatefulField(current_url)]
public URL CurrentURL;

[StatefulField(current_position)]
public int CurrentPosition;
```

Figure 8 State annotations in a browser application. Just by annotating the URL and the page position, the browser becomes stateful and can participate in activity roaming.

gram is a small overhead and is negligible compared to other requirements which an operating systems imposes on applications and application-programmers.[1]

5 Evaluation

Earlier versions of the activity-based computing platform have been evaluated extensively by clinical personnel from different hospitals [18], which has provided good evidence that the principles and suggested technologies are useful in a hospital setting. The goal of the work presented in this paper is, however, to investigate whether support for activity-based computing can be part of a contemporary OS, like XP. This raise the following usability questions:

Usefulness Our primary concern was to establish whether the whole idea of activity-based computing as embedded in the Windows XP operating system (OS) is useful, i.e. can ABC be part of a contemporary OS? Can activities exist in an OS or does it clash with the desktop metaphor? Is ABC useful in other, non-clinical, domains?

Ease-of-Use Our secondary concern was to establish if the ABC user interface for XP was easy to use. Our goal was that experienced Windows XP users would be able to use the ABC extended version with no prior training.

To answer these questions, we wanted experienced XP users to use the platform because they can give much more accurate test results that precisely address the stated research questions. Our previous evaluations have shown that clinicians are far from experienced computer users and many of them had serious difficulties understanding and using basic functionality in XP. Furthermore, to assess if activity-based computing is useful outside a hospital, we need users and scenarios from a non-hospital setting.

Usefulness was more specifically evaluated by investigating how users were able to use the activity concept to aggregate services and data when handling different tasks, large amounts of digital material, parallel work, interruptions, migrating work across different devices when moving around, and adapting the user interface of an activity to the available screen real-estate.

[1] The programming model is based on the .net platform and the example in Figure 8 is C#.

Methods

We have refrained from measuring task completion time. It is trivial to see that users of an ABC platform like the one presented in this paper would out-perform standard Windows XP users if asked to perform tasks that require handling parallel tasks, accessing lots of digital material, coping with frequent interruptions, and moving across heterogeneous devices.

The goal was to provide objective measurements on the usefulness and usability of our design while, at the same time, investigating the underlying detailed user reaction to the ABC user interface in a more qualitative fashion. For this purpose we have devised and used a *multi-method evaluation setup* where we (i) ask users to perform a range of tasks while thinking aloud, (ii) do analysis by scenario [51], (iii) investigate perceived usefulness and usability based on a questionnaire [54], and (iv) make a semi-structured interview.

Experimental Setup

To recruit experienced Windows XP users, we asked 16 graduate computer science students to participate in a semi-controlled evaluation. They were tested separately, each test lasting just over 1 hour. All participants were skilled programmers. Mean age was 27 years.

The test was set up to simulate well-known tasks for the participants, in this case a programming assignment for a company. The experiment was run in 3 rooms by 2 experimenters and a 'scenario judge', responsible for the analysis by scenario. Room 1 was the participant's 'office', room 2 was the office of his colleague (played by experimenter no. 2), and room 3 was the 'board room' of the fictional company at which the participant worked. Experimenter no. 1 and the scenario judge followed the participant around as he changed locations. The participant had a desktop PC, a laptop, and a telephone in his office. The board room had a computer with a wall-sized display. The 3 computers all had varying screen resolutions and displays, the ABC XP extension, the Eclipse IDE, MS Word, MS PowerPoint, Internet Explorer, plus five custom-built medical applications.

Tasks

Six overall scenarios were the backbone of the test: (i) Code; prepare a presentation; perform private activities; be ready for support calls. (ii) Take care of personal matters. (iii) Answer a support call from a client wanting support for medical applications. (iv) Consult a colleague in a nearby location to get help for the code. (v) Take care of personal matters. (vi) Present your work to the Board Members.

Experimenter no. 1 conducted the test according to a script which both guided the participants through the overall scenarios and their subtasks, and at the same time introduced interruptions and asked the participants to start working on parallel tasks. Examples of interruptions are different kinds of phone calls introducing

new tasks and people entering the office. Examples of parallel tasks are asking the user to do some brainstorming in MS Word.

Procedure

After signing the informed consent form, the participant was given a quick introduction to the ABC user interface and the think-aloud method. He was then given a description of the tasks he was to perform. While performing these tasks, the scenario judge observed the participant and decided, based on a breakdown factor and on a 5-point scale, how much of the ABC Framework's potential the participant used for each scenario. This yielded an 'analysis by scenario' score [51].

A questionnaire was administered to the participants after the experiment. It consisted of 39 questions covering perceived ease of use and perceived usefulness. The latter were further sub-divided into the factors listed above (see also table 1). Each factor was covered by a number of redundant questions, the order of which was randomized, and framed in both positive and negative ways. The questionnaire is an extension of the 'perceived usefulness/ease-of-use' questionnaire [54]. Participants were asked to make self-predictions about their likely future use of the ABC Framework. This, we believe, will increase the ecological validity since the user to a lesser degree is evaluating the ABC Framework with regard to specific laboratory tasks but evaluating its potential role in their own work. As argued by Davis and others, these kinds of self-predictions "[...] are among the most accurate predictors available for an individual's future behavior" [54, p. 331]. Participants were asked to make self-predictions based on questions like: "Using ABC in my daily work would enable me to handle more information than today." Answers were given on a seven-point scale with 'likely' and 'unlikely' as end-point adjectives.

After the questionnaire, a semi-structured qualitative interview was conducted to get evaluation feedback in the participants' own words. It featured questions such as 'In your own words, what is the best and the worst thing about ABC?' and was designed to address everything from the participant's sentiments about the test to any suggestions for improvement they might have. The whole test was video-taped, including the follow-up interviews, and subsequently analyzed and partly transcribed.

Results

The results from the questionnaire are shown in Table 1. Low scores indicate 'likeliness', e.g. a score on 1 is 'extremely likely', a score on 2 is 'quite likely', a score on 3 is 'slightly likely', and a score on 4 is 'neither likely or unlikely'. Scores above 4 indicate 'unlikely'. Table 1 shows that on average the participants found it quite likely that ABC for Windows XP would be easy to use (2.13, std. dev. of 1.05). The ease-of-use questions included questions on learnability, understandability, and flexibility. Furthermore, the participants found that on average it would be slightly likely that ABC would be useful to them (3.06, std. dev. of 0.92). Looking more specifically into the underlying factors, the mechanisms for activity aggregation and activity

roaming were perceived quite likely to be useful. In general, it is interesting to note that on average none of the evaluated factors in Table 1 were perceived unlikely to be useful (i.e. all scores are 4 or below). The standard deviation for all factors is around 1, which indicates that the answers were rather consistent.

	Avg.	*Std. dev.*
Usability	2.09	0.89
Usefulness	3.07	0.96
Activity aggregation	2.67	1.07
Large amount of data	3.36	1.37
Parallel work	2.97	1.15
Interruptions	3.16	1.27
Roaming	2.31	1.07
Het. Devices	3.11	1.73
Zoom feature	4.07	1.70

Table 1 Results from the questionnaire.

The results from the analysis by scenario by the scenario judge is shown in Table 2. In this measurement, high figures (5) indicate that the experimenter judged the participant to make extensive use of the features of ABC. Table 2 shows that on average participants had a high score in the use of ABC and its features. These figures indicate that participants quickly learned how to use and utilize the features of ABC, supported by the results from the ease-of-use questionnaire. Note, however, that the figures in Table 2 also reveal that users learn as they use ABC – the score for the same scenario ('Personal tasks') is higher the second time.

#	*Scenario*	*Avg.*	*Std. dev.*
1	Code	4.33	1.35
2	Personal tasks	4.33	1.29
3	Support call	4.27	1.49
4	Colleague	4.87	0.52
5	Personal tasks	4.87	0.35
6	Presentation	4.53	0.52

Table 2 Results from the analysis by scenario.

6 Discussion

Going back to the original work by Bannon et al. [12] we feel that the current ABC extension to Windows XP has come a long way in creating computational support for activity management. As our evaluation shows, users found it likely that ABC would help manage parallel tasks and interruptions,"reduce mental load when switching tasks" and help users "suspend and resume activities" [12, p. 54].

Our approach has been to embed activity support into the operating system because we intent to extend, rather than replace, the way users use files and appli-

cations today. Based on their studies of task switching and interruptions in Windows XP, Czerwinski et al. [53] reaches a similar conclusion: "It is clear that more can be done within the operating system and software applications to help users multitask and recover from task interruptions, hence potentially increasing productivity." [53, p. 181]. The need for activity support on the operating system level was also pointed out by one of the users during the follow-up interview:

> *I think this is smart – to associate things to activities. I've often thought about having 'association tags' on files, thereby being able to associate them to different things instead of them just being a file. [...] It's appealing to think of this as a greater framework for using your computer where files just have disappeared. [...] I think all of this is nicely done, and I'm certain that if you guys don't do it, then others will do it, because it is completely obvious to think this way...*

This user finds the idea of having activities as an organizing concepts around files and applications 'smart' and this way of organizing the operating system for 'completely obvious'. Another user similarly argued that the simplicity of the approach was a benefit, i.e. that the Windows XP ABC mechanisms introduce a minimum of additional 'things' to the operating system.

> *It is extremely simple – that's also a good thing. It doesn't try to be advanced in any way. It can do a few things and it does it very nicely – that's the best thing about it.*

Going back to Table 1 the support for 'activity roaming' was perceived useful, which were also backed up by the interviews:

> *The best thing is the ability to move your 'state' from one computer to another. The whole idea of making it persistent... It's extremely nice to be able to close your computer and then it comes back up in the same state. That's the reason why I newer turn off my Linux machine – it just runs for weeks as does all of the applications on it.*

The original motivation behind activity roaming was to loosen the tight one-to-one binding between a user and a (personal) computer. Hence, by supporting activity roaming users would be free to use different computers and to move around more easily. This hypothesis was also confirmed in our experiment, as illustrated by the following quote:

> *[I]t is a sensible way to be independent of precisely this particular computer. That you have the opportunity to, well ... close the book and then open it again another place and you get the same back.*

The interviews also contained questions regarding areas for improvement to the current system and its user-interface. Users had some more general comments

on the user-interface and suggestions for improvement. They found that the zoom function was necessary in the adaptation on different display sizes, but they found that it could be greatly improved and work as smooth as the ExposÈ function in Mac OS X. Currently, the ABC zoom functionality is simply too slow to be useful. One user also had suggestions for creating automatic adaptation to the size of the screen. Furthermore, there were comments on providing better feedback to the user on how a service and a piece of data (e.g. a file) were related to an activity. We regard these issues as important, but they are more related to further development of the user-interface for Windows XP, rather than more fundamental research issues.

Users raised, however, also more fundamental issues. First of all, the apparent lack of support for having the same service (window) in more than one activity. As argued by Bannon et al. [12] you need "multiple perspectives on the work environment" and need to support 'multiple windows' as done in the Rooms system [10]. The current implementation of the ABC extension to Windows XP actually supports this technically – the same window can be part of more than one activities and its state may be different in the different activities, as also pointed out in the Rooms paper to be important. The users just cannot do it because there is no user-interface handles for doing it; we simply have not been able to come up with a good design for doing this in a simple manner.

The other more basic issue raised by some users were concerned with the life cycle of an activity, i.e. when does an activity emerge, when do you create one in the user-interface, how does it end, and how is it related to other activities. As argued by one of the users:

> *The ¬worst thing? Well [...] if you have to put everything into activities, then you need to constantly consider '¬where does this one belong'. In many situations something just appears quickly and then you start up some application and do some things in it. And if you don't get it categorized into some activity, then it may disappear? Or you may forget ¬which activity it ¬went into. Then you may need to search for it.*

This user addresses the basic notion of how an activity emerges, or more specifically how it often may overlap or intertwine with another activity. In a previous version of the ABC user interface, application windows which were opened were automatically attached to the currently resumed activity. In our own use of the system we however discovered that this was inconvenient – often your windows and files are opened 'inside' an activity, but they do not have anything to do with this activity. They may, for example, result from an interruption. As the user interface works now, the user is able to launch new applications and files and then suspend the currently resumed activity. This will remove application windows related to the activity, and the user is left with only those new windows, which have been opened due to the interruption. These windows may then form the basis for a new activity, or may just be used without any activity support.

7 Conclusion

In this paper we have presented an approach for embedding support for activity-based computing in the Windows XP operating system, with special emphasis on the single user experience. The core principles for activity-based computing was presented, which pivots around the support for aggregating services and data in coherent sets called 'activities', support for activity suspend and resume, support for activity roaming between different computers, and the support for activity adaptation to different display sizes.

We presented the design and implementation of the ABC extension to XP. The core user-interface components are the Activity Bar, which replaces the Windows Taskbar, the Activity Icon which integrates activity-support to native Windows application windows, the Activity List showing a user's activities, and the Activity Zoom which support spatial 2D layout of the windows within a service.

Finally, the ABC extension to XP was evaluated using a multi-method evaluation setup. This evaluation revealed that users found the user interface easy to use and that activity-based computing support would be useful for them in their work. Due to the multi-method approach, we were also able to establish more precisely what part of the user interface that needed to be improved. These results are being used in our current enhancement of the system, which also targets the support for roaming files together with activities. Hence, files belonging to an activity would move between computers as the user roams.

Acknowledgments

The ABC project is funded by the Danish Research Council under the NABIIT program. Henrik B. Christensen has been much involved in the early work on ABC and Martin Mogensen has helped with the implementation.

Paper II

IASO – An Activity-Based Computing Platform for Wearable Computing

Jakob E. Bardram Jonathan Bunde-Pedersen

Abstract

Displaying and navigating complex information on wearable computers is very different from accessing the same information while sitting at your desk. Limitations in hardware, screen-size, input- and output devices are not met with changes in the software running the wearable system. We propose using the ABC-framework as a middleware layer providing collaboration, adaptation, activity sharing and context-awareness for wearable applications. Furthermore, enhancements in the user-interface and interaction techniques overcome some of the hardware limitations. The proposed system makes wearable computers usable by e.g. emergency workers, policemen and firefighters.

1 Introduction

Wearable computing is an emerging technology which enables the use of computers and access to digital mate-rial in working situation that are physically and conceptually far away from the office desk. Despite an intensifying research endavour, wearable computing still faces a number of challenges before wearable computing

Published as: Jakob E. Bardram and Jonathan Bunde-Pedersen. Iaso – an activity-based com- puting platform for wearable computing. In *ICDCSW'05: Proceedings of the Fifth International Workshop on Smart Appliances and Wearable Computing (IWSAWC) (ICDCSW'05)*, pages 484–490,Washington, DC, USA, 2005. IEEE Computer Society.

hardware and software melt naturally into everyday activities like police work, military warfare, maintenance work, and paramedic work. Some of these challenges involve basic hardware issues [71, 160]; other challenges involve more basic software architectures and operating systems for wearable computing [57]; and yet other challenges involve how users can use such computers for information retrieval, overview, data input and collaboration [26, 37, 159, 101]. In this paper we want to address the challenges in helping users navigate, overview, and interact with complex information on a small wearable screen; how to make a bridge between the user's current work activity and the working context; and how collaboration in wearable computing setup could be accomplished.

Activity-Based Computing (ABC)[1] have been proposed by researchers as a new computing paradigm for pervasive computing [48, 19, 165]. This paradigm helps users organize the computational support in logical bundles specific to a human activity and helps users to share such activities in a collaborative effort. Activity-Based Computing can be viewed as a pervasive computing platform – or operating system – with a underlying distribution middleware, a resource and context aware adaptation architecture, and a new user-interface metaphor. Furthermore, it contains a collaborative environment for synchronous and asynchronous collaboration.

In this paper we demonstrate how Activity-Based Computing can constitute a computational platform for wearable computing. This work is motivated partly by our current empirical work designing wearable computing support for emergency workers, like paramedics and ambulance personnel, and partly by a review of the research done within the field of wearable computing. Section 2 presents a future scenario for supporting an accident using wearable computers, and Section 3 presents and discusses four design goals and principles for Activity-Based Wearable Computing. The paper then presents 'Iaso', which is an Activity-Based Wearable Computing platform with special emphasis on new types of user-interfaces, sharing of many displays, context-aware activity support, and support for collaboration[2]. Section 5 presents details on the implementation and the deployment of the Iaso platform, and Section 6 concludes the paper by discussing to what extend the Iaso platform meets the design principles stated in Section 3.

2 A Bus Accident Scenario

The 911 call center receives an alarm on a bus accident and immediately dispatches 5 ambulances to the scene of the accident. When the alarm goes off in the ambulance garage, the paramedic Dr. Strong and the other ambulance personnel grab their uniforms ¬with a built-in ¬wearable computer. While driving to the scene, they prepare for the accident by uploading maps of the scene and by loading medical data on the people involved in the accident. All this information is prepared for easy access and display later. When arriving to the scene Dr. Strong starts by ¬walking around to all the injured people to create an overview of the accident. He attaches an RFID tag ¬visible on all patients. This tag ¬works as a temporary

[1] Also known as 'Task-centered computing'

[2] Iaso was a Goddess of Healing and the daughter of Asclepius.

identification and he links this unique id with the patient's set of medical data which have already been prepared. The ambulances at the site create local WLAN coverage and has 3G connectivity back to the call center.

By using the speech interface, Dr. Strong is able to toggle between the different patients and to zoom in on specific information. While he is visiting the patients he (along with the other paramedics) are filling in the 'trauma forms', which indicate how, where, and how bad each patient is hurt. All this information is relayed to the ambulance server and on to the call center.

The work load for Dr. Strong is too heavy due to the number of injured people. Hence, he contacts the call center and ask them to help fill in the trauma form by communicating with one of the ambulance personnel and to find and prepare some vital information on a specific patient who seems badly injured. When the call center has the requested information ready, Dr. Strong touches the patient's RFID tag, thereby resuming all the data views and forms associated with this patient. The call center highlights specific information on allergies to penicillin and show Dr. Strong some specific sections of particular interest in the record.

Dr. Strong decides to move this patient for hospital treatment immediately and moves the patient to an ambulance. When entering the ambulance, Dr. Strong moves his user session on his wearable computer to the touch display in the ambulance. This display is much larger and hence helps him to have a better overview of the patient's condition. The medical equipment in the ambulance is also used now and all the vital signs from the patients are shown on the display and is transmitted to the call center and on to the hospital. Arriving at the hospital, a staff of well-prepared surgeons and nurses take over the patient who is moved directly to the operating room.

3 Key Design Principles

Based on our research into the working conditions of paramedics and emergency personnel, combined with the a litterature study of the current state-of-the-art within wearable computing, we have identified four key design principles for the Iaso wearable computing platform. These are support for 'preparation and recall', 'focus + context', 'shared collaboration', and 'using multiple displays'. In the following we shall discuss these principles in detail and compare them to related work in this area.

Preparation and Recall

Most scenarios for wearable computing involve users in situations where there is little time for navigation and information seeking, like train maintenance [159], aircraft inspection [127], the fire service and police force [84], warfare [186], and the emergency workers described in this paper. The key challenge in most of the applications of wearable computing is the limited time and support for information search, retrieval, selection, filtering, and displaying. Based on these observations researchers have proposed to use the principles of 'context-awareness' to have the computer adapt to the user's current work situation [2, 129]. As a supplement to

context-awareness in wearable computing we are suggesting 'preparation and recall' as a core design principle for wearable computing. This principle enables a user or one of his colleagues to prepare for a situation and then recall relevant data and/or services when this situation occurs. This is illustrated in the scenario above where the paramedics use a standard template for a patient and recall a prepared map of the accident area.

Focus + Context

A core challenge in wearable computing is the small screen and resolution of state-of-the-art head mounted displays (HMD)[3]. Furthermore, the standard user-interface mechanisms for managing large amount of information and/or applications on a small screen are not feasible in wearable computing. Overlapping windows, the Windows or Linux taskbar, and 'Alt-Tab' switching between applications is hard to use on a wearable computer. It has been suggested to use principles from Virtual Reality in wearable computing by creating 'Wearable Information Spaces' [37]. The information spaces allow users to navigate between multiple pages of information by moving their head as if they were standing inside a virtual cylinder of information. The same idea has been explored in [108] where they create a virtual space around the user using 3D-sounds. In the emergency scenario such solutions would not work because field workers are constantly moving their head to overlook the scene of the accident. Instead we are pursuing the user-interface design principle of 'Focus + Context' [69, 25], which enables the emergency worker to have an overview of all his activities, over what is taking place within each activity, and then focus on specific aspects within each activity. This is illustrated in the scenarios above where the paramedics focus on different medical data while maintaining an overview of his on-going activities.

Shared Collaboration

Support for collaboration has been identified as central to wearable computing [101]. However, most of the collaboration support pivots around audio and video links, enabling a remote 'expert' to guide the field worker (see e.g. [101, 26, 102]). In many cases, however, the expert *is* the field worker – or he is at least the expert in the current situation. This is typically the case in the emergency scenario. Little attention has been given to enabling the remote user to take active part in the field worker's local activity. Hence, a design principle for collaborative wearable computing is to enable 'activity sharing' in the sense that a remote user can take active part in an activity in the field. This is illustrated in the scenario above where the call center employee searches, sorts, prepares and presents relevant information for the paramedics and where he starts helping the paramedics in filling in the trauma form. In this case the call center employee is participating actively in the work on the site and is not merely providing oral advice.

[3] A typical HMD today is the MicroOptical SV-6 with a VGA resolution on 640x480 or 800x600 in 18 bit color.

Using Multiple Displays

Most research on wearable computing assumes that the only display used is the head mounted display worn by the user. This is, however, seldom the case. In many situations additional displays like PDA, tablet PCs, laptops, and touch screens mounted in e.g. the ambulance are available. A central design principle for wearable computing must be to enable the user to utilize this computational context in his work. For example, to let the paramedic in the scenario use the tablet PC and the screen in the ambulance. This must be achieved with no or minimal overhead – the user must simply be able to walk up to a display and transfer his session. This enables him to utilize the larger screen, resolution, the keyboard and mouse, as illustrated in the scenario above. This also helps users to overcome the limitations in screen size and resolution, and the limited input devices on state-of-the-art wearable computers.

4 Activity-Based Wearable Computing

We are currently working on applying the principles of Activity-Based Computing as a metaphor and platform for wearable computing. Based on the ABC architecture and development framework [19] we have created the Iaso platform for wearable computers. This architecture has special focus on the design principles listed above.

The principles of Activity-Based Computing can be summarized as:

Activity-Centered A 'Computational Activity' assembles a set of services, needed to support a user carrying out some kind of (work) activity. In the scenario above, there is an activity for each patient which contains services (data view and forms) for (i) the patient medical data, (ii) the trauma form, and (iii) the chart displaying vital signs (see figure 3(b) or Figure 2(a)) which illustrate an activity showing these services). This ABC principle supports collecting services into logical bundles around a typical user activity and support the user to easy toggle between activities, thereby supporting alternating work tasks and interruptions.

Activity Roaming An Activity is stored in an infrastructure and can hence be distributed across a network. An activity can be suspended on one device and resumed on another in another place. This principle supports mobility and enables the user to use available devices nearby – potentially without carrying any device. In the scenario above, activity roaming takes place when Dr. Strong moves his session to the display in the ambulance.

Activity Adaptation An Activity adapts to the resources available on the device on which it is resumed. Such resources are e.g. the network bandwidth, cpu, or display on a given devices. Hence, an activity might look quite different whether it is resumed on a wall-sized display or on the wearable computer.

This principle suggests a path in the middle between thin clients using web services and self-contained applications running on homogeneous devices.

Activity Sharing An Activity is shared among collaborating users. It has a list of participants who all can access and manipulate the activity. Hence, one user can resume another user's activity and continue working on it. Furthermore, if two or more users resume the same activity at the same time on different devices, they will be notified and if their devices support it, they would engage in an on-line, real-time 'activity sharing' session. This is what takes place in the scenario above where the person at the call center helps Dr. Strong during an activity.

Context-Awareness An Activity is context-aware, i.e. it is able to adapt and adjust itself according to its usage context. Context-awareness can be used for adapting the user-interface according to the user's current work situation – e.g. by showing medical data for the patient currently in front of Dr. Stong. Or it can be used in a more technical sense, where the execution of an activity, and its discovery of services, is adjusted to the resources available in its proximity. This happens when Dr. Strong resumes the activity in the ambulance.

The Iaso Platform

The Iaso platform for wearable computing is illustrated in figure 1. This architecture is built on top of the ABC Framework as a special purpose client tailored for wearable computing. However, many of the core functionalities are directly inherited from the ABC framework, features like support for activity management, storage, activity sharing, context-awareness, collaboration awareness, and the support for activity-adaptation to multiple devices and displays.

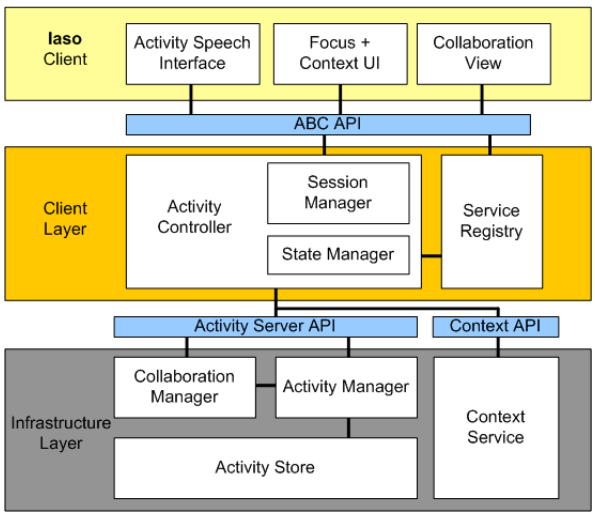


Figure 1 The Iaso system platform as built on top of the ABC architecture.

The ABC Framework consists of an infrastructure layer and a client layer. The infrastructure layer is responsible for storing (the Activity Store) and runtime management of activities (the Activity Manager), for collaborative activity sharing (the Collaboration Manager), and for handling context-information (the Context Service). The infrastructure layer is typically deployed on a set of distributed powerful machines.

The client layer resides on each device participating in the ABC deployment. This layer is a middleware layer between the user-interface and the infrastructure layers. The core component is the Activity Controller, which is responsible for controlling activities on the client and for communicating with the Activity Manager in the infrastructure layer. The State Manager is responsible for handling activity state when an activity is resumed and suspended on various clients. The Session Manager is responsible for handling the collaborative activity sharing session.

The Iaso client resides on top of the ABC framework. It consists of a speech interface client which enables the user to speak to the Activity Controller, thereby enabling him to suspend and resume activities, create new activities, and to navigate within activities. The Iaso client also contains a Focus+Context user-interface component, which helps the user to toggle between an overview of activities and focusing on details within an activity. Finally, the Iaso client contains a list of current participants. The next section described the User-interface of the Iaso client in greater details.

User Interface

The Focus+Context user-interface component of the Iaso client is illustrated in Figure 2 and 3, the former on a wall-sized display and the latter on a wearable computer. Hence, the same activities can be displayed on screen with a substantial difference in screen size and resolution. The first screen on 3(a) is an overview of all active activities for this user, Figure 3(b) and Figure 2(b) are overviews of a specific activity. Lastly, Figure 2(c) and (d) shows details within the activity, focusing on a specific window (or service) in the activity. Each activity consists of a number of applications or services, each running in a separate window. Hence, the trauma form is one application, the map of the accident scene is another.

The wearable user can switch between these two levels of overview/detail using voice commands. The commands are limited to "Show all services" in Figure 2(b), "Focus service `number`" (Figure 2(c) and (d)) and "Resume/Suspend activity `activity name`" (Figure 2(a) and Figure 3(b)). This is a rather limited set of commands but sufficient to navigate in the activity-based user-interface. Support for voice input in the specific applications (like the trauma form) is not part of the Iaso platform. Hence, support for voice input and navigation within applications and services are specific to the application itself.

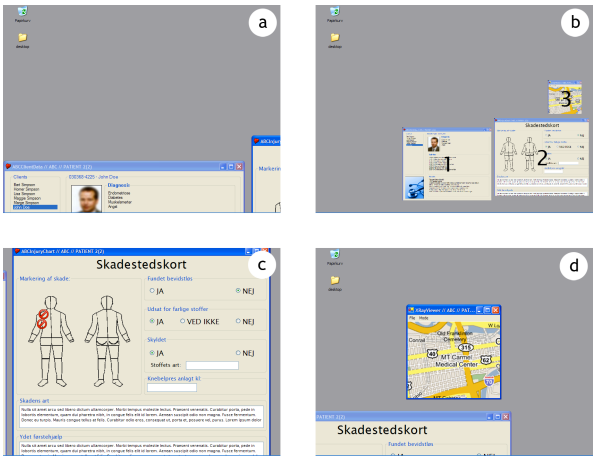


Figure 2 *(a)* The activity just after it has been resumed. *(b)* All services are visible when the user issues a 'Show all services' command. *(c)* and *(d)* Show two different services (trauma-form and map) which have been 'focused' on.

Activity-Based Collaboration

The ABC Framework supports 'Activity Sharing', which means that when two or more users engage in the same activity they have a shared view on the activity and the services or applications running within it. Furthermore, voice links and tele-pointers are created between the participants. In the scenario above, Dr. Strong would e.g. be focusing on the list of injured people on the site (Figure 2(b)) while the colleague at the call center is helping him by filling in the patient's trauma form (Figure 3(b)). The two users would be able to see what each other is doing and hence work directly together or have a mutual awareness of each other. They would be able to talk to each other and the person in the call center would e.g find some information and present it for Dr. Strong. Dr. Strong would see this information pop up as a new miniaturized window, which he can zoom in on using the voice command. The lay-out of the windows are maintained across platforms. Hence, if the person at the call centers uses a monitor with a 1680x1050 resolution while collaborating with Dr. Strong using the wearable monitor with a 640x480 or 800x600 resolution, they will still see the same layout of windows within the activity, but at different zoom levels. Hence, the activity view (Figure 2(a)) shows the whole activity zoomed to the current display.

The degree of collaboration within each application is dependent on the application itself. Hence, the degree of collaboration support in the 'Trauma Form' application depends on its implementation as some kind of client-server system where both Dr. Strong and the person at the call center can access shared data (a web service for example). Hence, the Iaso platform enables collaboration on the activity level and cannot guarantee collaboration within each service[4].

[4]The ABC Framework has an API that helps programmers to develop or extend existing application to make them 'ABC-aware', including participating in activity sharing (see [19]). However, the ABC platform also supports legacy applications without modifications, but in this case we cannot

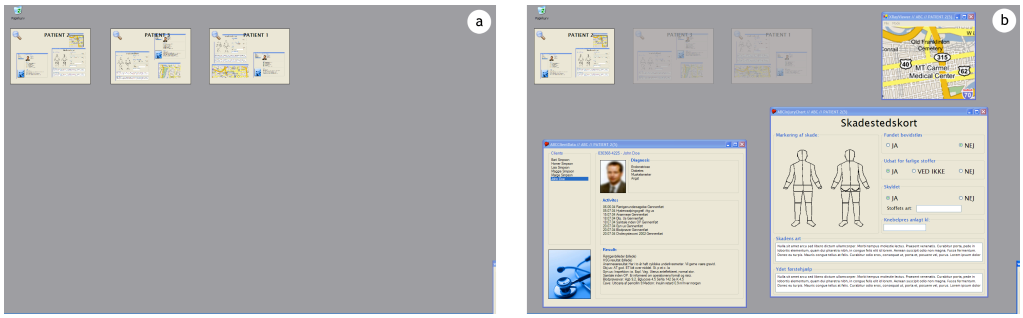


Figure 3 *(a)* An overview of all activities. *(b)* A single activity has been resumed.

Context-Aware Activity Retrieval

Creating a temporary but unique identification of more or less unconscious patients is, according to the emergency workers, a central challenge today. Often, patients are identified by their traits and type of injury, which is very imprecise. The Iaso platform is context-aware and allows for context-based retrieval of an activity. Hence, an activity can be linked to contextual information. This feature is exploited to link simple RFID tags attached to the patient with the patients activity and hence his medical and personal data. By scanning the patient's RFID tag, the Iaso clients present this patient's activity with all its relevant information.

Context-awareness is also used to help Dr. Strong move his session from the wearable to the display in the ambulance. When he and the patient enter the ambulance, Dr. Strong is able to resume his activity on this display.

5 Implementation and Deployment

The platform has been implemented in a version 1 on top of version 4 of the ABC Framework, where the infrastructure layer is implemented in Java and the client layer implemented on the Microsoft .NET platform in C#. The speech interface has been implemented using the Microsoft Speech SDK 5.1. The prototype has been evaluated by members of our research team and has been improved based on their feedback.

The Activity Manager supports both a request-response access to stored activities as well as an event-based subscribe-publish-notify mechamism for distributing events. We have designed an ABC Protocol (ABCP) which supports clients to request and store activities, and to subscribe to changes in activities. An Activity Markup Language (AML) is used to describe activities and events. The ABC Protocol runs over TCP/IP sockets and hence provides and language independent implementation between the infrastructure layer and the client layer.

The .NET client provides the middleware-layer on which the applications run.

guarantee collaboration support.

This layer manages the state of all applications and furthermore monitors the client computer for any changes in the state of the current activity, e.g. a new application being started. This is all done transparently to the user. We have implemented wrapper classes which make it possible to use existing applications such as Internet Explorer, Word, and Notepad. We also provide an API which may be used for creating new Iaso-aware applications. Creating a new application is quite simple; simply extend the class and provide implementations of a state-setting and state-getting method.

Regarding deployment, our current design pivots around a centralized infrastructure layer with one or more coordinating activity servers and a wearable client running Windows XP with the ABC framework and the Iaso software. This is in many ways a 'heavy' platform for wearable computing, which requires substantial hardware and networking resources. However, our initial research into the operating conditions of the accident scenario makes us confident that these deployment requirements can be meet. The wearable unit can be deployed on the Xybernaut MA-V, which runs Windows XP and the infrastructure layer can run on small servers installed in the ambulances and other vehicles arriving at the scene. These vehicles can also provide WLAN coverage for the accident scene and can communicate back to the call center using UMTS. It should be noted that the requirement for network bandwidth from the event-based activity sharing in the ABC Framework is rather limited, even in collaboration mode. Hence, the largest demand for bandwidth would emerge from the use of audio and/or video.

6 Conclusion

The main contribution of this paper has been to introduce four design principles for wearable computing, which supplement the work within this research area. These principles come out of our design of wearable computing for emergency situations, and are:

- Preparation and Recall
- Focus + Context
- Shared collaboration
- Using multiple displays

We have presented the Iaso platform for wearable computing, which seeks to address these principles. This platform was built on top of the ABC Framework and hence supports 'Activity-Based Wearable Computing'. We believe that this platform meets the four design principles above.

Firstly, the design principles of 'Preparation and Recall' is evident in the Iaso platform by enabling the user to prepare various activties and store them for later retrieval. This preparation can take place on other computer devices than the

wearable one, thereby enabling the user to take advantage of keyboard, mouse, and other I/O devices during preparation. By organizing computational services in activities, which are tailored to expected future work activities on the accident site, the emergency workers are able to have resources ready and to easy toggle between them when arriving at the site.

Secondly, the Iaso platform contains a novel, custom designed user-interface which helps the user to get an overview of all his running activities as well as focusing on specific details within these activities. By zooming in on a specific service and zooming out to see a whole activity or the whole set of activities, the user-interface seeks to support the design principle of 'Focus + Context'. For example, the user can by simple voice commands toggle between a large overview of all the activities and detailed information on e.g. the patient or a map of the accident site.

Thirdly, the Iaso platform enables the users (on site as well as persons providing back-up) to collaborate equally and help each other to carry out an activity. This 'Shared Collaboration' is basic to the ABC Framework and is hence a central part of the Iaso platform. The support for activity sharing enables users to join activities and users sitting at e.g. the emergency call center or physicians at the hospital is able to join an activity on the accident site and help the on-site user by finding and displaying information, or to engage directly in the activity by e.g. entering data. The current version of the Iaso platform does not support video communication only audio links. Video is an important part of a final deployable wearable computer support for emergency work and we are currently investigating the best design of video use. One suggestion is to have head mounted cameras (as in [101]). Another option is to have fixed cameras on-site mounted e.g. on ambulances and other vehicles. However, focus of this paper has not been on video cooperation, but the support for sharing data and views with other mobile and non-mobile collegues. This 'Activity Sharing' is a different approach to collaboration than the 'expert-helps-novice-using-video' approach seen so far in wearable computing.

Finally, by using the ABC Framework the Iaso platform enables the user to utilize computational devices in the nearby context. We can probably all agree that wearables are difficult to use (the Twiddler keyboard has a steep learning curve) and the support for moving whatever you are doing to a 'real' computer is not trivial. Hence, support for moving the users session along with all the user's activities to the computer in the ambulance is essential.

In the future we plan to continue this work and we are in the process of setting up both in-house and on site evaluation session with emergency workers. Currently, researchers from our research team is doing on-site ethnographic studies of emergency work which will help us design the type of application to run on the Iaso platform as well as provide some more feedback on the design of the platform itself. At a later point, we plan to evaluate the support for wearable computing during emergency drills where everything but the injured patients are for real.

Acknowledgments

The research presented in this paper has been funded by ISIS Katrinebjerg competence center, Aarhus, Denmark. The design ideas in the bus acident scenario were developed by the students attending the 'Designing Interactive Systems (DIS)'class at the University of Arhus in the Fall 2003 taught by Morten Kyng and Preben Mogensen.

Paper III

Supporting Activity-Based Computing in a Distributed Multiple Display Environment through the Clinical Context-aware Activity Browser

Jakob Eyvind Bardram Jonathan Bunde-Pedersen
Afsaneh Doryah Steffen Sørensen

Abstract

A multi-display environment (MDE) is made up of co-located and networked personal and public devices which form an integrated workspace that enable co-located group work. Traditionally, MDEs have, however, mainly been designed to support a singular 'smart' room, and have had little sense of the tasks and activities that the MDE is being used for. This paper presents the Clinical Context-aware Activity Browser (CCAB), which is a novel approach to support activity-based computing in *distributed* MDEs, where displays are physically distributed across a large building. CCAB was designed for clinical work in a hospital, and enables context-sensitive retrieval and browsing of activities in this setting. We view the hospital as one large distributed MDE where activities can roam between available devices enabling the hospital staff easy access to archived information. We present the design and implementation of CCAB, and report from an evaluation of the system at a large hospital. The main contributions of this work are a novel activity-based computing approach for using wall-based and mobile public displays in a distributed MDE, combined with initial evidence for its usage in the nomadic and collaborative environment of a hospital.

Submitted for: INTERACT 2009. 12th IFIP TC13 Conference in Human-Computer Interaction.

1 Introduction

A multiple display environment (MDE) is comprised of co-located personal and public devices, ranging from small devices (e.g. PDAs, laptops, and tablet PC) to large wall-based displays, which are networked to form an integrated workspace [36]. The goal of such MDEs is to foster co-located group work utilizing available devices.

An on-going research challenge is to design the technologies and user interfaces for such MDEs. Many systems and interaction techniques have been proposed [168, 86, 94, 36, 140, 166], each addressing specific issues of creating an MDE. Existing research has, however, mainly focused on MDEs within a single physical space – often referred to as a *Smart Space*. Little work has been addressing what we call a'Distributed Multi-Display Environment' (dMDE), i.e. an MDE that is distributed across several physical places inside e.g. a large building. Furthermore, most existing research has been focusing on information and device management for sharing basic data on the devices. For example, sharing and exchanging documents, spreadsheets, presentation, images, etc.

This paper proposes CCAB; a set of user-interface technologies and infrastructure elements for supporting a distributed multi-display environment. The design of CCAB builds on the principles of Activity-Based Computing (ABC). Within ABC, the user's activities are modeled and managed by the technology. Modeling means that the technology has a persistent model of the type of activity, the intention, relevant resources, and participants of the activity. Managing means that the technology is able to store, distribute, adapt, and modify activity models according to their use and context. In CCAB the user's activities becomes explicit in the display environment. An activity is then used to browse and manage the complex set of applications, documents, files, and resources which are associated with performing a task. The current implementation of the CCAB system is limited to accessing and browsing pre-defined activities, other aspects of activity-management including creation and distributed collaboration is not within the scope of this paper.

The goal of CCAB is to help users appropriate and use publicly available displays while moving between different physical places. Key features include the support for easy access to an aggregation of all information and data relevant for a specific activity; access to this data from all displays in the dMDE; support for moving activities and their related resources around in the dMDE; support for both fixed (wall-based) and mobile public displays; support for context-aware adaptation of the access to relevant activities; and support for a seamless transition between using the display in a public, personal, or private mode. This work on supporting nomadic use of public displays is motivated by the design of computer support for clinical work inside a hospital – work which is characterized by being highly nomadic, collaborative, and time-critical [22]. Hence, the type of dMDE we are designing for is a setup where displays are scattered around the entire hospital and are available in e.g. conference rooms, patient wards, hallways, and operating rooms.

2 Related Work

Research on MDEs or Smart Spaces dates back to the original work on Ubiquitous Computing at PARC, which supported an integrated user experience ranging from small-size 'pads' to medium-size 'tabs' to large-size wall-based displays [183]. Similarly, the iLand project [168], the Interactive Space project [94] and the Gaia smart space project [86] have been researching the infrastructure and interaction technologies for seamless integration of mobile and fixed displays in MDEs. Technologies like PointRight [95], ARIS [35], and IMPROMPTU [36] enable users to bind devices together, re-direct control and user interfaces, and to share and move information across multiple displays. These projects all hold two things in common. One is that the technology is deployed inside one room only; the other is that besides from simple knowledge about who is in the room, the technology has no knowledge of the activity of the users. Our work deliberately focus on dMDEs as well as creating support for handling devices, information, services, and users in relation to the activities that is taking place in the dMDE.

Explicit computational support for human activity is the main focus for activity-based computing approaches. For example, activity-based computing support for the personal computer [1] illustrates how the computer can model the human activity and use this for information and window management in the Windows XP operating system. Similarly, the Unified Activity Management project [123, 124] uses activities for information management and collaboration inside large organizations. More automatic systems based on sensor inferencing have demonstrated how activities can be inferred from low-level user interaction with the computer. For example, the context-aware activity display (CAAD) [136] is able to detect, cluster, and highlight activities that the user might be engaged in. This is used for semi-automatic information management of documents, files, webpages, and emails associated with an activity.

None of these activity-based computing approaches address large interactive displays which can be used for *public* purposes in a dMDE, as the CCAB does. Furthermore, the CCAB is designed to highlight relevant activities in a specific contextual situation using context sensing of the physical world. This is different from the CAAD system, which is targeted towards activity recognition and highlighting based on low-level user input to a personal computer.

A particular set of semi-public display have been categorized as 'awareness displays', i.e. displays that provide the user with peripheral information on the flow of work. Kimura [110] is an augmented office environment using interactive personal peripheral displays to manage 'working contexts'. The Notification Collage [80] is a system for users to communicate through a network of desk-mounted peripheral displays and public large displays. Because the CCAB constantly lists and highlights relevant activities for its current context, it could work as an awareness display. The aim of the CCAB however, is to support a more active engagement with the displays. Dynamo [91] enables collaboration around a large interactive display, but relies on the users to provide data and organize applications and content at each new session. Thus, Dynamo does not relieve the user of the overhead

that also exists in regular PC usage. The CCAB similarly supports collaboration in a public setting, but in addition it explicitly supports the activity of a user and thereby decrease the work involved in starting applications, finding relevant data, etc.

3 Background

This work takes its outset in observations and design of technology for clinicians working in hospitals. In many ways, the work of hospital clinicians are very different from that of information workers – the latter having been the prototypical target for the design of interactive computer technology. First of all, work in hospitals is nomadic, i.e. clinicians constantly move around inside the hospital visiting different patients, departments, wards, conference rooms, and colleagues [22]. In contrast to mobile work, nomadic work is characterized by the fact that the clinicians do not have any place to sit and work, i.e. most clinicians do not have a desk or an office of any kind. Therefore, it is most often the case that clinicians do not have a personal computer, but use publicly available terminals or PCs scattered around the hospital.

At the same time, clinical work is intensively collaborative; most patient care happens in close collaboration between a team of highly specialized clinical professionals who constantly need to align, coordinate, and articulate their respective part of the overall activity of treating the patient. This collaboration typically takes place co-located, i.e. clinicians meet for planned or ad-hoc conferences and discuss the patient case. Examples of such conference situations range from the formal medical conference every morning, to improvised and ad-hoc conferences between the nurse and the doctor in the hallway of the ward.

When analyzing the work artifacts inside a hospital, one finds that publicly available 'displays' play a core role in the execution and coordination of work. Examples include large whiteboards listing operations, patients, beds, and other critical resources; the medical records; notebooks; and small notes. Some of these are fixed to the wall while others are mobile. However, common to them all is their public nature, that they are used as a shared collaborative resource, and the way they form a loosely coupled web of coordination artifacts [22].

4 Design of the CCAB system

The hardware side of the CCAB system consists of multiple interactive screens linked through a distributed infrastructure. The screens range from portable clinical tablets to large interactive wall-based displays. The software consists of two layers; a context-sensitive presentation layer that displays persons, activities, resources, and applications; and a distributed infrastructure layer handling communication between devices and runtime management of activities. Our notion of activities is similar to that found in [I,14], where activities are computationally

represented as a collection of related resources and applications that can be serialized and thus migrate between machines. A serialized activity consists of the state of its applications, and *resuming* an activity thus restores the state of these applications to when the activity was last *suspended* [1].

Infrastructure

The distribution of data and activities is handled by a flexible distributed storage and event system called *aexo*. The main features of aexo include support for modeling complex hierarchical data, managing relationships in the data, and a publish-subscribe scheme for event-subscriptions and -notifications. The distributed data-structure which forms the foundation for aexo is a hierarchical map.

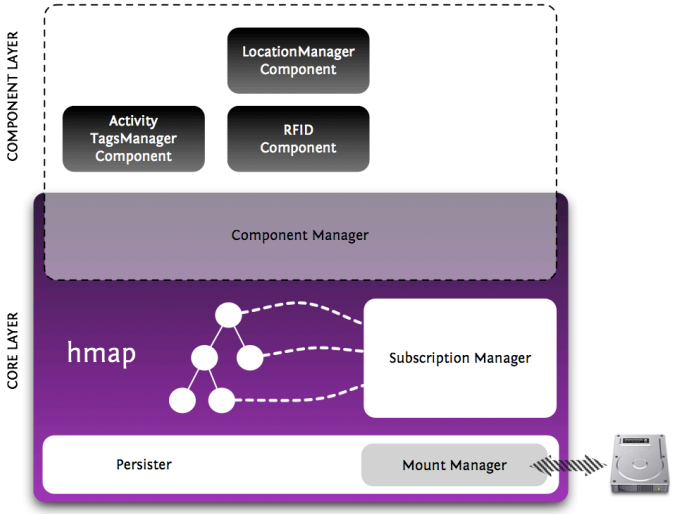


Figure 1 The infrastructure for the CCAB system is a distributed event-based system, where data is stored in a hierarchical map.

The infrastructure illustrated in Figure 1 consists of two layers: A minimal core layer and a layer containing functional components. The core layer is simply a distributed hierarchical map (hmap), in which data can be stored. It also functions as a publish-subscribe mechanism, enabling subscribers to listen for notifications when certain nodes or submaps are modified. The component layer provides the business-logic of the deployment and consists of independent components which interface with the map. The presentation layer exists independently of the infrastructure. It interacts with the data in the hmap by exporting its internal model into it. This is done through a remote reflection layer which binds selected internal field-variable values to nodes in the hmap. Whenever a process modifies the exported model, the application is notified and updated and vice versa when changes happen internally in the application. Much of the business-logic needed by the presentation layer is implemented in components which are dynamically loaded by the infrastructure. These components manage e.g. location tracking through RFID and update the model in the hmap based on this information. The presen-

tation layer simply has to specify which submap of the hierarchical data-structure that it is interested in. This is done using a path to the given submap, for example the path `/Locations/Hallway/Activities` contains all activities which are relevant for the "Hallway" location. Similarly, submaps such as `Users/JohnDoe/Activities` contain activities which "John Doe" participates in. The argument for externalizing business-logic is twofold; first, it implements functionality which is not specific to the presentation layer alone, but may be utilized in other supporting systems, e.g. a personel tracking system. Second, it simplifies the implementation of the presentation layer, such that it can focus on the presentation of activities and resources, and does not have to explicitly made location-aware. The presentation-layer merely states that it is interested in displaying activities from a specific submap, and the infrastructure-based logic handles the updating of this submap according to its internal-logic, e.g. based on location.

Presentation Layer

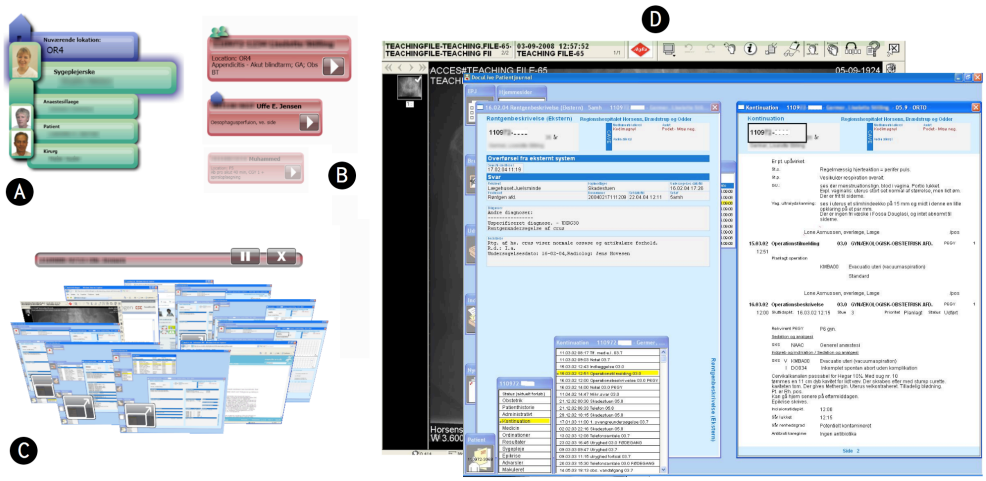


Figure 2 The presentation layer of the CCAB system – **(A)** Context Bar **(B)** List of Activities, **(C)** a resumed activity with associated resources displayed in a carrousel, and **(D)** two applications launched from the carrousel.

The presentation layer affords users with presence information listing the persons co-located with the screen as seen in Figure 2 **(A)** and concurrently provides a listing of relevant activities. The relevancy is determined by two factors; the presence of a person increases the relevancy of the activities, which this person is involved in, and the location of the screen itself is also used to filter activities. This is accomplished by "tagging" certain activities as being relevant to certain locations. The relevancy is visualized in the size and transparency of the activities listed in **(B)**. An activity can be resumed by clicking the "Play" icon on one of the activities listed here. When an activity is resumed the "carrousel" **(C)** appears. This view represents a resource perspective on an activity. Here, the user may "flip" through resources within the activity and, since multiple "carousels" can be opened at any time, is afforded an overview of the contents of one or more activities. Individual

resources in an activity may then be launched by clicking the minimized view of the resource, resulting in **(D)** where applications have been *resumed* to the state contained in the activity, and showing the resource in question. This perspective allows users to continue work within applications. When the activity is *suspended* once again by clicking the "Pause" icon, the state of the activity is saved. This includes information about the placement and state of any resumed applications. Clicking on the "X" dismisses the "caroussel" view. Each screen can display multiple resumed activities simultaneously enabling co-located use.

Design Principles

The four principles the CCAB system design is based on are: *activity management*, use of *fixed and mobile displays*, *context-based adaptation*, and a separation between *public, personal, and private* uses.

The key principle in the CCAB is support for basic *activity management* [I]. The keyword here is activity-centric computing, where resources and participants associated with a real-world human activity are modeled and aggregated in a corresponding computational activity. Moreover, modelling activities as first level entities enable us to support roaming, i.e. moving a computational activity from one device to another. This principle is specifically targeted at supporting easy management of the myriad of heterogeneous data associated with a patient case, and to support the nomadic work inside hospitals. Interruptions are supported through the suspend and resume operations on activities enabling users to quickly switch between activities. Activity-centric systems often include explicit support for collaboration distributed across multiple machines as well, however the current implementation of CCAB has no such explicit support. The creation and maintenance of activities is also not within the scope of the work presented in this paper, but our previous work has explored this issue extensively [14, 22, I].

The infrastructure and presentation software supports a combination of *fixed and mobile displays* that are physically distributed inside a large building. Activities are transferred to the device based on relevancy calculations, which also affects how the activity is presented as described earlier in this section. The fixed displays react to changes in their environment, while the mobile displays move between environments. The mechanism to enable these context switches is essentially the same on both types of devices, and allows information to flow to the screens where it is deemed most relevant.

Context-based adaptation of the user interface highlights and provides easy access to relevant activities. The distribution of data as well as the presence- and location-tracking is handled by the infrastructure, such that a device that is moved from location to location is automatically reconfigured to show the most relevant information for this location. The reconfiguration is possible because the model and the controller logic of the presentation layer is externalized in the infrastructure in a similar manner as e.g. the Presto [59] system. This minimizes the effort and time spent looking for files and information within applications.

The system distinguishes between three levels of privacy; public, personal, and private, and enables *seamless transition* between them. *Public* is the default mode where the relevancy algorithm puts equal weight the presence information of all close-by personnel. The *personal* mode however is used when a personnel picks up a mobile tablet computer. Then the screen is seen as appropriated by this individual and consequently his or her presence is most important. As the individual carries the device around however, the current location is still reflected in the presentation software. *Private* mode is activated when a USB stick containing private activities are inserted into a screen. Then presence- and other context-changes are ignored effectively transforming the device into a private machine.

5 Evaluation

The system was evaluated at a surgical department at a large hospital as illustrated in Figure 3. The locations within this department included the conference room used by the surgeons for the morning conference; the patient ward where all admitted patients were hospitalized; the hallway of the hospital; and inside one operating room. The technical setup included three wall-based public displays using 42" touch screens displaying the CCAB. These wall displays were deployed in the medical conference room, in the hallway leading to the patient ward (Figure 3 **(C)**), and inside an operating room (Figure 3 **(D)**). A Lenovo tablet PC and a Motion Computing C5 Clinical tablet PC were used as mobile public displays. The mobile displays were used mainly at the patient ward (Figure 3 **(B)**). All devices (both mobile and fixed) and all clinical personnel were tracked using active RFID tags (WaveTrend LRX400).

The system was used during a full day (8 hours) by clinical personnel from the orthopedic department, including surgeons, operating nurses, anesthesiologists, and ward nurses. In total 8 clinicians participated. Since patients were not intended users, no real patients were involved; two staff members of the hospital helped play the patient role. The acting patients, were, however treated as any real patient, including being physically examined by the doctors and prepared for surgery. The operations were executed exactly as a real operation, except that the patient was not sedated nor actually cut.

The creation of the 15 patient cases as activities was performed in a "Wizard of Oz" [151] manner by a human operator prior to the evaluation. As the focus of the evaluation was browsing the relevant patient data and not interaction with the applications, we used the screen shots instead of the real applications. All of these activities had associated medical data from the electronic medicine system, the electronic patient record, the radiology system, and various other web-based applications. Figure 2 shows a typical example of the user interface of the CCAB with this data.

Figure 3 The area of the hospital in which we evaluated the clinical prototype. **(A)** is the conference room, **(B)** the patient-ward, **(C)** the hallway, and **(D)** the operating room.

Evaluation Methods

The evaluation was centered around a real patient case in an anonymized form. Hence, the entire patient case was authentic, including all the medical data used and the events happening to him. The main flow of the case was:

1. The patient is admitted to the hospital for acute appendicitis during the night
2. In the morning, surgery is being planned during the medical conference
3. The patient is visited at the bed ward

4. An ad-hoc discussion of the case takes place in the hallway
5. Surgery is prepared in the operation room
6. Surgery is performed
7. The patient is taken back to the bed ward

The goal was providing objective measurements on the usefulness and usability of our design while, at the same time, investigating the underlying detailed user reaction to the system and the user interface in a more qualitative fashion. For this purpose, we used a *multi-method evaluation setup* where we (i) asked the users to perform the different steps in the expriment while thinking aloud, (ii) investigated perceived usefulness and usability based on a questionnaire [54], and (iii) made a semi-structured follow-up interview.

6 Results

The results from the perceived usefulness/ease of use questionnaire are shown in Table 1. Overall, the system is perceived very useful (4.16, std. dev. of 0.65) and – to some degree – easy to use and learn (4.13, std. dev. 0.57). Looking at the specific aspects of the system, Table 1 also shows how the clinicians perceived the potential usefulness of the main features of the CCAB: activity management; using fixed and mobile public displays; context based adaptation; and the support for moving between using CCAB in a public, personal or private mode.

Perceived usefulness	*Avg.*	*Std. dev.*
System usefulness	4.16	0.65
Ease of using and learning	4.13	0.57
Activity Management	4.25	0.52
Public Displays	4.34	0.69
Context-based adaptation	3.56	0.84
Public-Personal-Private	4.22	0.79

Table 1 Perceived usefulness summary.

Activity Management

Support for activity management scores high – 4.25 (std. dev 0.52). Hence, the clinicians either agreed or strongly agreed that it would be useful to gather all relevant patient information in an activity, that an activity would help recover relevant patient data quickly, and that it would be easy to get access to patient data while moving around inside the hospital.

During the interview, the clinicians consistently argued that the support for managing and distributing clinical data was a strong feature of CCAB – as argued by a surgical nurse:

> *Such a system will definitely be a huge advantage. Everything would be together in one place. Today, we spend a lot of time in getting access to the different systems, and to find the right documentation in them.*

Similarly, the simple overview that CCAB provides was highly appreciated by the clinicians as an important feature in a busy work environment – as put by another operating nurse:

> *...What I like about the system is that we all use different aspects of the clinical data, and I can just select the parts that I need. I think the user interface is extremely simple.*

During the discussion, the clinicians also suggested things for improvement. For example, templates for displaying activities or services in fixed locations on the screen might increase overview and you would always know where to look for a certain type of information. As put by a nurse: "*...it would help [you get] an overview ...if you knew where to look for things*".

What we derived from these comments was the need to have some sort of partitioning scheme both with regards to *who* gets to manipulate the data, but also regarding *where* specific types of information appear. For instance, a nurse would be looking for a other kinds of information than a surgeon, and placing e.g. the x-rays and the medical record in two distinct areas of the screen would help avoid conflicts between users. It would also provide more structure to the layout of an activity thus making it easier to see what kind of information it contained.

Public Displays

Using public displays in a clinical environment was considered very useful (4.34, std. dev 0.69). Hence, the clinicians agreed that fixed and mobile displays would be a good fit to the working conditions of a hospital; that it would be important to have the same CCAB interface on both the wall-based and the mobile displays; and that being able to just walk up to a public display and start using it would be a crucial feature. As expressed by a nurse:

> *[...] another good thing about this system is, that in a busy hospital [...] you do not need to spend time accessing data. You just go to the screen and get the data. [...] One can imagine that there are 10 of these mobile displays for each [patient] ward and you just take one from a re-charger station when you enter the ward.*

The participants also discussed where to place wall-based and mobile displays. They concluded that the smaller, mobile displays would suit the work in a patient ward, whereas it would be of little use inside the operating room. In the operating room, the large wall-based displays would be useful since they display more data and can be viewed from a distance. Furthermore, carrying mobile display in and out of operating room also implies a contamination hazard.

Context-based Adaptation

The feature of context-based adaptation scores consistently low (3.56, std. dev 0.84). In particular, the clinicians did not agree that CCAB was displaying the most relevant activities for a specific situation, and that using location tracking of personnel would help find the most relevant activities. Rather than co-location of people in front of the display, it seemed that the relevance of activities – or more specifically the patient cases – depended more on clinical issues like urgency of the case, or the order of the operations to be done in the operating room. As argued by a surgeon:

> *On the operation hallway – if we want to have information about a patient then we can go to the display and activate the name. But in the operation room, it is perhaps better to have the patients' [activities] fixed on the screen so that they will not change or move.*

Another issue with the adaptation is that the content and appearance of the activity list would change based on people entering and leaving the room – as expressed by an operation nurse:

> *When, for example, [the surgeon] enters the operation room he can see his own activities and can see what he is going to do after this process. And it is really good for [him]. But the team in the operating room may be confused if his priority tasks begin to affect on the tasks which are linked to the people in this room.*

Public-Personal-Private

Smooth transition between a public, personal, and private use of the displays also scores high on perceived usefulness (4.16, std. dev 0.65). The clinicians clearly agreed that it would be beneficial to see both shared as well as personal activities, and they also argued that in most cases privacy is less of a problem, since ownership of patient related clinical data needs to be shared among all parties involved in the treating of a patient. As argued by the ward nurse:

> *...it is good that only you have access to the private data on the USB [stick] and not others, but as long as it is patient information and something which everybody has an interest in, it is nice to be able to transfer the screen to another [member of the staff].*

The clinicians, however, pointed out that it was important to consider exactly where to put the public wall-based displays; they should be located in places where people, irrelevant for the patient case (e.g. other patients and relatives), would not accidentally view the data and overhear a conversation. In the evaluation setup, the display in the hallway (Figure 3 **(C)**) was place in a too crowded and public space.

7 Discussion

Our evaluation of the CCAB and the underlying aexo platform for supporting activity-based distributed multi-display environments has provided us with a series of insights, which provides directions for further research. In this section, we will focus on discussing three main topics which we see as having immediate benefit for human-computer interaction research.

Activity-based dMDEs

Based on our initial evaluation, it is quite evident that activity-based computing support for MDEs in a hospital environment is a quite promising prospect. CCAB supports bundling of related services and data which is associated with some real-world human activity. In our evaluation, these activities were patient cases, but there are numerous other candidates for activities as well. The clinicians agreed that this support for activity-centered bundling of resources is very useful. Furthermore, the support for roaming the activities between public displays, as enabled by the aexo platform, was essential for work inside the hospital.

The distributed nature of our interactive space is very unlike most related research on MDEs. The physical disconnectedness of the devices and displays in a dMDE is a crucial point of difference. The structure of hospital work and the architecture of hospitals themselves do not easily lend themselves to the application of MDE as a technology confined in isolated rooms. Instead hospitals have vast corridors, offices, operating rooms and patient wards. The approach we have taken is to consider the entire hospital as an interactive space and deploy multiple displays in such an environment.

This has raised a number of new research challenges, which traditional MDEs do not address. Among these, we found two issues to be of central importance for a dMDE; how to enable the flow of activities and data between the public displays, and how to enable users to access and start using the displays in the dMDE.

In CCAB, the first challenge is addressed by using location information to automatically direct relevant activities onto a wall-based public display. Hence, activities – and their associated resources – would simply follow the users around inside the dMDE. Based on the feedback from the clinicians, this approach seemed to work well.

Furthermore, by using mobile devices, activities could be 'carried' wherever they were needed, and was used in the case of the ward-rounds. In contrast to traditional use of tablet PCs, the CCAB also turned this device into a public display participating in the dMDE. In several cases, the tablet PCs were handed over from one clinician to another (as shown in Figure 3 **(B)**). In these 'hand-over' situations, the display simply switched context and was carried for use elsewhere in the dMDE.

Finally, to support accessing and carrying private activities – with associated data – the USB stick served as a 'pocket-book'; i.e. a private store for activities

which could be inserted in all fixed and mobile screens thereby converting them to personal devices. This kind of local adaptation of specific displays in the dMDE is another important difference from a traditional MDE.

Public Displays and Privacy

There seems to be an inherent conflict between using public displays and maintaining privacy. The latter is especially important in a clinical domain where sensitive patient data is displayed and discussed. As outlined in previous section, this issue was also discussed by the clinicians while using the CCAB in the hallway. It was a deliberate choice of location to put the display in this rather public location in hope to spark a discussion of the feasibility of having such public displays around the hospital. The discussion, however, was concerned with several aspects of privacy. The first aspect was concerning whether it is appropriate that one participant can see and access the activities of another participant present in front of the display. An example is seen in Figure 3 **(C)** where three clinicians are standing in front of a public display showing their combined activities. After some discussion, the clinicians concluded that this was actually not a privacy issue, because all patients hospitalized at the hospital in principle should be accessible by all clinicians, and the activities were only accessible if the user was in physical proximity of the display. Once the user moves away, his or her activities are no longer present in CCAB at that specific screen. Even though the reading range of the active RFID tags seemed to be too long in the deployment setup, the clinicians agreed that this design was suitable for upholding of privacy in hospital-specific activities. By using the USB stick – either as a key or as a holder of data – the users were able to retain private activities for themselves.

Another aspect of privacy was concerned with the physical deployment of such large wall-based displays. The deployment location of the hallway display was not appropriate in the opinion of the users; it was simply put in a place where too many people were passing by, including relatives and other patients. Hence, it would not be feasible to discuss a patient case in this place. The clinicians concluded that such displays would be very useful if put in the hallway of the operation ward or the patient ward. Inside the operation room, there were no privacy issues.

In summary, the inherent conflict between using public displays and the concern for privacy can be mitigated through the proper modeling of public, personal, and private activities in the CCAB and its infrastructure, combined with an appropriate deployment of the displays and tracking technology.

Context-based Relevance

In the current implementation of CCAB, context-based adaptation is linked to the physical co-presence of participants of an activity; the more participants present in the same location as the public display, the more relevant the activity becomes. For the clinicians, this model was too simple – physical co-location could be an indicator for relevance, but a series of other more clinically oriented issues seems

to be more relevant, like the operation schedule of the day, the urgency of the patient case, and whether the case needed to be discussed on the morning conference. The current relevance score based on co-location also had an annoying side-effect, because when an RFID tag became invisible due to other reasons than the person leaving the room (e.g. being covered by something), then the relevance of the activities would immediately change, which would resize and re-organize the list of activities. This caused a lot of confusion during the evaluation. Since the technical implementation of the CCAB allow for easy replacement of the relevance algorithm, a natural next step would be to investigate how to improve this based on e.g. more clinical context information.

8 Conclusion

In this paper, we have presented a novel approach to activity-based support for distributed multi-display environments – dMDEs. We have shown the interaction design and technical infrastructure of the CCAB system. Taking an outset in the nomadic and collaborative work of hospital clinicians, CCAB was designed to enable easy access to medical data across physically distributed public displays. CCAB was evaluated in a large hospital. The evaluation revealed that the clinicians found CCAB very useful and easy to use. The support for activity management, distribution of clinical data, and the ability to quickly access all relevant data on any public display were considered as strong features of CCAB. The evaluation, however, also pointed to areas for further research and improvements, including improving the current implementation of context-based adaptation. Our future work will – amongst other things – concentrate on activity adaptation in distributed multiple display environments.

Paper IV

Towards an Activity-Based World-Wide-Web

Jonathan Bunde-Pedersen Jakob E. Bardram

Abstract

The World-Wide-Web (WWW) provides a simple, but strong, approach to data integration by enabling pages to link to information and data physically distributed on different servers. More sophisticated data integration, however, requires specialized tools and might not even be possible on some proprietary data formats. Activity-Based Computing is a top-down approach which takes advantage of existing applications which are integrated into 'activities'. This paper introduces the ABWWW extension for ABC for which brings support for activities on the WWW, thereby introducing a novel integration technique for heterogeneous sources of information.

1 Introduction

The notion of activity-based computing (ABC) has emerged from studies of work in hospitals, which is characterized by being mobile, disconnected, collaborative, and interrupted [17]. ABC was originally developed in this hospital context but the concepts and principles introduced by ABC have proven to be more widely applicable [1]. This paper therefor deals with ABC in terms of no general domain. In order to provide technical support for activity-based computing we have created the ABC framework, which is a pervasive computing infrastructure and user-interface that focuses on integrating applications on the desktop. The core idea is to move the focus of users away from applications and files to activities, which represents

Published as: Jonathan Bunde-Pedersen and Jakob E. Bardram. Towards an activity-based world-wide-web. In *IIWeb'06: Proceedings of The 3rd IIWeb Interdisciplinary Workshop for Information Integration on the Web*, Edinburgh, Scotland, 2006. ACM Press

a coherent collection of 'resources' which are used in a specific task. Activities are an abstraction on top of applications, services, and data as illustrated in figure 1. Services are abstractions of actual applications, such that a service identifies the "semantic" of an application, e.g. 'text-editor' is the service which represents Notepad, Microsoft Word, Emacs etc.

This paper presents an extension of the ABC framework which integrates activity-based computing with the world wide web. This extension uses the concept of activities to provide mechanisms for organizing heterogeneous sources of information into coherent activities. The activity becomes the shared link which binds such sources together and the functions already present in ABC now becomes available to participants on the web. The core tasks which ABC enables and brings support for to the WWW is summarized below:

Online collaboration The ABC client itself already has support for numerous modes of collaboration as discussed in [V]. Activities are the core concept which collaboration builds upon. An activity can have any number of participants and the contents of the activity will be kept synchronized across the machines of these users if they work simultaneously. Asynchronous collaboration is also possible since the contents of an activity may be synchronized using a server. The ABWWW extension brings these capabilities to the world-wide-web by using an URL scheme for keeping track of the locations of activities, and by synchronizing activities over standard web-protocols.

Integration and aggregation By using activities to bind application instances and their state together, we have enabled users to integrate different sources of information. For instance, a web-page might be associated with a text-document by having a browser and a text-editor in the same activity.

Providing different views on digital material An activity is basically a description of a number of application states. This generic description is then interpreted on the device which resumes the activity, where each device may choose to present the activity differently. This provides each participant with a unique view of the activity, which he/she may alter to fit the current task at hand. An activity is thus a tool to construct multiple views on heterogeneous material.

Structuring worktasks An activity is made to reflect an actual task, such as 'writing a paper', so activities are a structuring tool for work on computers. Switching between activities is switching between different work-tasks.

In this paper we describe how the ABC framework is merged with web technologies and how these two things combined make activity-based computing widely available for users on the web. This means that activities, which bundles a large set of material, can be shared across the web. As such, the combination of ABC and the web can be viewed as a sophisticated meta-browser capable of not only showing one page but a whole set of resources – web pages as well as other application data – in one coherent activity. For example, a physician can click on an ABC web link

in an email and the whole medical activity related to a patient would be shown, including e.g. the medical record shown as webservices in a browser, access to X-ray images, and to a medicine chart.

2 Related work

The field of hypermedia and hypertext has historically focused on integrating linking-principles in existing applications as well as the structuring of information and how to access an application through its elements« interrelationships. Bieber et al. [34] argue for using the world-wide-web as the interface for integration of hypermedia mechanisms in all applications. Tools for structuring information through linking is seen as a requirement for this integration and a further adoption of hypermedia. Similarly Bouvin [40] calls for a more structured approach to information integration on the web. The focus is here on frameworks for augmenting existing information using overlay structures to provide annotations and for relating content through links. They present the framework 'Arakne' which uses 'Navlets' to add custom user-interfaces and domain knowledge to existing content. Navlets are small applets custom-built to handle a specific type of content. Our approach is similar, but we have focused on using existing applications to view and manipulate content, while activities are our basic structuring mechanism.

Another project supporting human activities as first class computational activities is that of Hsieh and Shipman [89]. They explore facilities for representing activities as relationships between a number of documents and the timeframe in which they are accessed. These spans of activity are objects which can be linked from an extension to the Visual Knowledge Builder [158]. Chimera by [7] describes 'hyperwebs' which are conceptually similar to activities. A 'hyperweb' is a collection of "objects, viewers, views, anchors, links, clients, and users". Activities are similar in the sense that they are collections of applications or services which provide views on files and objects. We do not have an internal structuring mechanism to link between services though, structure is solely provided by grouping services into activities.

3 Activity-Based Computing

Activity-Based Computing is a research area devoted to helping users cope with the increased complexity of daily computer use as well as a more suited interaction metaphor (than desktop computing) for pervasive environments. Files and applications are the basic concepts found in operating systems since the dawn of modern computing. Most work-tasks done on computers today still consist in the manipulation of files using applications, and the number of files and applications being used is ever growing. ABC tries to remedy this situation by providing an abstraction on top of applications and files called activities [18, 15, 48]. It is important to notice that ABC does not try to model workflows - we impose no structure besides that of activities on work tasks.

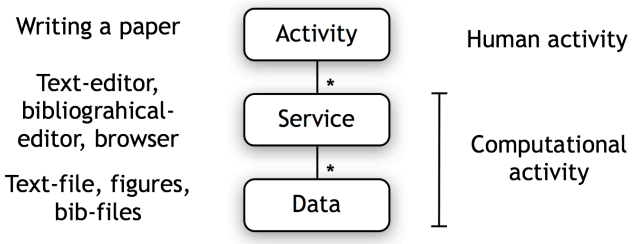


Figure 1 The hierarchy and concepts of ABC.

Computational activities as a concept encompass applications and files allowing the user to more easily switch between different work-tasks (human activities). A *human activity* is for instance "writing a paper", and the corresponding computational activity is the again a description of the applications involved e.g. a text-editor, a bibliographical tool and a browser as well as their state. The state of an application is the information which is deemed relevant to save across instantiations. For example a text-editor may need the filename and position of the cursor as its state. The ABC framework we have built features computational activities which can be created, suspended, and resumed on any computing device in the environment at any point in time. It can be shared asynchronously, or shared among several persons working simultaneuosly. Furthermore, the execution of activities is adapted to the usage context of the users, i.e. making activities context-aware.

ABC as a framework is thus a system for managing human activities by providing OS level support for computational activities. We have so far focused on the perspectives of integrating our framework with the local OS and providing support for applications and files available locally. That is, we have focused on a local integration of the ABC principles. This paper reports on the progress of a more global approach, which is the integration of ABC principles on the WWW. Before discussing the extension in Section 4 we will give an introduction to the ABC framework, its architecture and general implementation.

ABC system architecture

The ABWWW extension to the ABC framework is built into the existing architecture which consists of three layers; the infrastructure, the client and the UI layer (see Figure 1). We will in the following briefly describe the basic structure of the ABC framework.

The infrastructure

The infrastructure layer is responsible for persisting activities through the *Activity Store* component, for managing runtime state and events through the *Activity Manager* and for distributing state-changes and other collaboration-related data

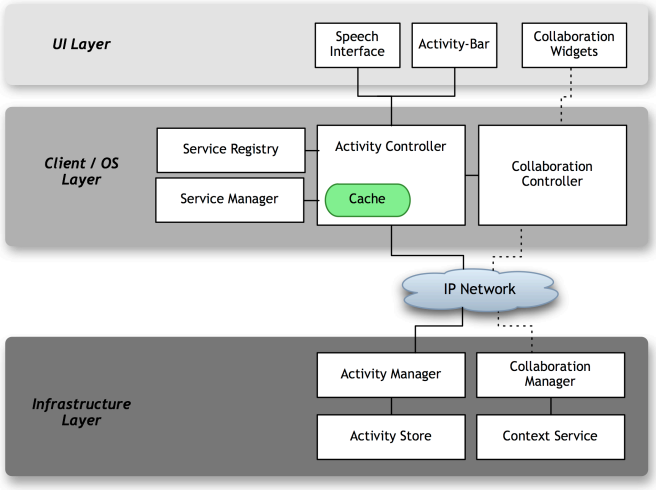


Figure 2 The architecture of the ABC framework.

through the *Collaboration Manager*. The *Context Service* component keeps track of users, machines and their context. The layer is commonly deployed on one or more servers – physically separated from the upper two layers. Information flows from context-sources through the infrastructure to the clients, and from client to client over the infrastructure now acting as an information broker between clients.

The client

Interfacing with the local operating system, applications and data is done in the client layer. The *Activity Controller* provides handles for resuming and suspending an activity, and manages connections to the infrastructure and other client-layers.

The connection to the infrastructure managed using a stateless HTTP-like protocol called ABCP. The client itself also maintains a cache which permits users to keep working in an activity if the connection to the server is lost. Once the server is back online, the cache will synchronize with the server once again. The ABCP protocol is an XML based protocol with storage methods GET and POST as well event-based communication through PUBLISH and SUBSCRIBE methods. The storage methods act as their HTTP counterparts and allows for retrieval and posting of activity-data. Activities are described using an XML format so they are highly portable.

Details of each activity is handled by a *State Manager*, which tracks changes in the state of local services and controls which services are part of the activity. The *Service Controller* component monitors the operating system for windows and wraps standard applications for use by a *State Manager*.

The *Collaboration Controller* allows an activity to be shared. When sharing an activity the state of all services in the activity is synchronized on all participating devices.

The user-interface for the desktop client as seen in Figure 4 is designed to be as unobtrusive as possible, providing a simple way to manage and switch between activities. We have built other user-interfaces for the framework as well; one for a wearable client and one for a wall-sized display. Each handle activities differently but rely on the same client- and infrastructure-layer and are thus compatible.

API for local applications

There are basically three levels of ABC framework integration for applications, i.e. such that the framework can track their state. The minimal level of integration is the situation where the application is almost completely closed of to the outside "world". This is for instance the case with Notepad. In such a situation we can only use what handles the OS provides to gain knowledge of the application's state. Often we can only get the window position and size, the executable of the application and its class – but no internal state. More information is available for applications which provide an externally reachable interface (e.g. COM). The quality of the state-information now depends on the quality of the interface. Full integration can be obtained if the application is made ABC-aware, i.e. it is being developed using a special ABC API which lets developers define the state of an application. The API consists of an interface which must be implemented (we have created a default implementation which can be subclassed with even less effort) and a way to annotate variables which represent state. An example of an annotation is given in Figure 3.

```
[StatefulField("current_url")]
public URL CurrentURL;

[StatefulField("current_position")]
public int CurrentPosition;
```

Figure 3 State annotations in a browser application.

By annotating the URL and the page position, an ABC-aware browser becomes stateful and can participate in activities.

4 Activity-Based Browsing

The extension to the ABC framework we have called ABWWW is a functional addition which enables users to:

- Access their activities via URLs and links.
- View and manipulate activities on the WWW using a browser.
- Use existing tools to organize, search and distribute activities.

- Use an ABC API to build activity-enabled distributed web-applications integrating a wide range of information sources via the ABC framework.

Still, the core of ABC remains intact providing OS level support collaboration and work-tasks. Integrating web-based information sources into ABC is basically a three step approach:

Firstly, we have built mechanisms in the current framework for activating activities using URLs, which fundamentally changes the semantics of links. Now, a link can point to an activity, and following the link will activate or *resume* the activity. An activity can integrate different sources of information and thereby provide a unique view of this information set. The URL mechanisms are also used to integrate existing WWW-tools into ABC. Since URLs now can communicate activities we can use existing bookmark-organizing tools, link-repositories etc. to organize and manage computational activities.

Secondly, by building protocol-bridges we can view and manipulate existing computational activities on the WWW. A browser will in effect become another client with which we can access our activities. The protocol-bridge provides a shared format for activity-manipulation and simplifies the task of further extending the ABC framework.

Thirdly, Perhaps more significantly we can use the basic infrastructure of ABC as a platform or framework for building activity-aware distributed web-applications. That is, applications whose state are automatically saved across sessions, which are automatically collaborative, and which can be made part of activities with all the benefits this provides.

Before discussing the implementation of the ABCWWW extension we will present the user-interface enhancements made to our desktop ABC client.

User interface

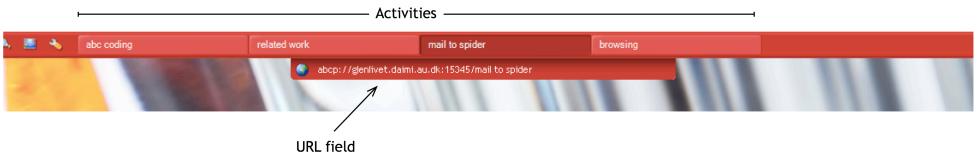


Figure 4 The *ActivityBar*

The graphical user-interface for ABC desktop clients seen in Figure 4 is called the *ActivityBar*. It is designed to mimic the taskbar of Windows XP but instead of buttons representing applications they now represent activities. This allows users to quickly switch between activities using the same basic interaction as they would when switching between programs in a standard Windows XP installation. The basic *ActivityBar* has been extended with a field showing the URL of the current activity as well as allowing the user to enter URLs for resuming remote activities.

This extension works in much the same manner as the location field of a browser. The UI is designed to offer a simple management interface for activities, to easily allow resuming and suspending your activities.

The ability to communicate activities as URLs can be utilized to manage databases of activities using link management tools available on the web or as stand-alone applications. Therefore, you can use your existing browser, as well as the tools for searching and organizing bookmarks that may be incorporated in it, to o maintain a directory of activities. There are also several web-based services which allow you to easily manage large amounts of bookmarks. An example is the del.icio.us[1] site which offers keyword-based search and tagging of links. By using URLs to identify activities we are thus given an existing plethora of mechanisms with which to organize them.

Figure 5 Using del.icio.us to organize activities.

Sharing activities between a group of people is also significantly easier using the URL scheme. An activity can thus be sent in an email, via instant messaging or simply posted on a WIKI. Integrating activities into existing WWW infrastructures is made significantly simpler using this approach.

ABWWW components

As outlined in Section 3 the extension for ABC consists of three parts. The first part has to do with the construction of an URL scheme for activities and implementing protocol handlers in the framework itself. The second part involves building a protocol-bridge which exposes activity-data to browsers and other viewers external to the framework itself. The third part of the extension is to construct a web-based API for building activity-enabled applications. An approach to this is

[1] http://del.icio.us

through an implementation of a Javascript library which enables application programmers to specify ABC components and ABC behaviour of HTML constructs. The library-functions are then responsible for communicating with the framework using the protocol-bridge.

Protocol handler

We use URLs to identify activities and more generally to probe the framework for information. The URL format for activities follow the standard basic URL format seen in Figure 6.

```
<scheme>:<authority>/<path>?<query>
```

Figure 6 Generic URL format

An ABC URL thus has the specific format given in Figure 7 and specific examples of ABC URLs are given in Figure 8.

```
abcp:<hostname>:<port>/
   <name or id of activity>?<query>
```

Figure 7 ABCP URL format

We have extended the ABC framework such that it is able to handle activitation of `abcp://` URLs. This is done partly by registering a protocol handler at the OS level such that whenever a user follows an ABCP link the framework is notified and partly by implementing mechanisms in the framework for parsing the URL, fetching the activity which the URL points at, establishing a publish-subscribe relationship with the remote host and finally resuming the activity on the local OS. URLs can also be used to query the framework for information. Figure 8 shows a URL which will make the framework return a list consisting of the public activities created by 'jbp'.

```
abcp://glenlivet.daimi.au.dk/iwi2006paper
abcp://glenlivet.daimi.au.dk/find?
   creator=jbp&type=public
```

Figure 8 ABCP URL examples

Protocol-bridge

The second part of ABWWW consists of a protocol-bridge implementation. The bridge is from the ABC protocol or ABCP where the data format is AML (an XML-based format for activities) to an HTTP protocol where the data format is XML and HTML. The bridge is implemented in PHP and XSLT. The PHP scripts

can probe the local network, talk to the ABC infrastructure, and then respond in HTTP via the Apache server on which the script runs. XSLT transforms the XML to HTML but a raw XML feed is also provided if the application making the request wishes to parse it according to some specific use.

API for web-applications

We are in the process of building the final part of the ABWWW system; the API which lets programmers write activity-enabled web-applications. The purpose of the API is twofold.

Firstly we provide mechanisms for programmers to specify the "state" of their applications, which will let the ABC framework take care of persisting and synchronizing the applications as well as including them in activities. This means that applications (as parts of activities) without much programming effort can be handed over to other users (a weak form of migration), that their state is preserved between activity-instantations and that they are able to be included in synchronous collaboration.

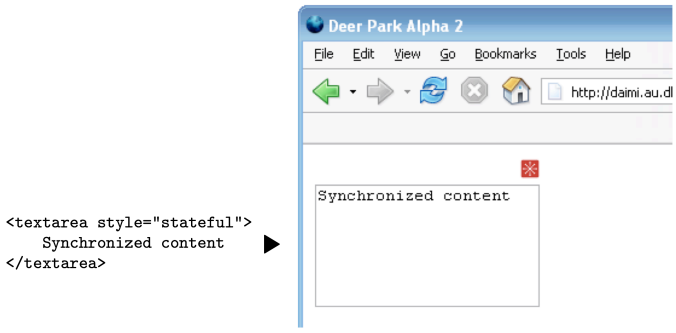


Figure 9 Web API example. The red ABC icon on top of the textarea indicates a stateful field.

Secondly the user can resume activities containing multiple web-applications, restoring their state and allowing a previously interrupted human activity to be resumed once again. This significantly reduces the mental overhead normally associated with finding the relevant applications and data used in some human activity. Also, the mobility of activities are enhanced, since it is possible to use different applications for the same "service". For instance, a text-editor might be both a web-application but also an application executing locally such as Notepad. So, depending on the context of the user it might make sense when resuming an activity in one instance to use the web-application while in another Notepad is more appropriate. The user needs only focus on the "semantics" of applications and not on the details of finding the program or file to be edited. In a collaborative scenario this means that when two persons are working in the same activity, one using web-based applications and the other local ones, they will see the same state in their applications, i.e. the web-texteditor will have the same content as the local text-editor.

For both programmers and users the ABC frameworks adds an abstraction layer enabling them the focus on the task at hand and not the technology at work.

We have implemented the URL scheme and the protocol-bridge and are currently working on the web API. It consists of a Javascript/AJAX library which handles all technical aspects, while the programmer can use 'tags' to indicate which parts of an application should be made 'stateful'. A 'tag' is currently a style attribute as seen on the text-field in Figure 9.

5 Future work

Our primary concern is finishing the web-based API and evaluating the complete system as described above. We are currently developing different ABC-aware application for helping surgeons access medical data while operating. These applications and the ABC framework including the web technology described in this paper is planned to be deployed for evaluation at a hospital during the summer of 2006. If successful, the plan is to have the ABC framework to be used in the daily work of a number of surgeons.

6 Conclusion

This paper presented ABWWW; an extension for ABC which brings support for activities onto the WWW. The extension consist of three parts; an URL scheme, a protocol-bridge and an API for web-developers which together allows activities to be created, suspended, resumed and communicated over the WWW. Furthermore the API provides a simple base on to which ABC-aware web-applications can be developed. Activities can then be used to integrate applications and data from multiple sources on the WWW as well as local machines.

Paper V

Differentiating between Accountable and Ephemeral Events in the ABC Hybrid Architecture for Activity-Based Collaboration

Jakob E. Bardram Jonathan Bunde-Pedersen Martin Mogensen

Abstract

An important issue in the design of a collaborative system is its architecture, which determines the responsibility, behavior, and distribution of the system components. In this paper we discuss the applicability of using a hybrid architecture which combines the benefits of the client-server and peer-to-peer architectures. In order to determine which events should float in the client-server part of the architecture, and which events should float in the peer-to-peer part of the architecture, we introduce the concepts of *accountable* and *ephemeral* events. The former is state event which the collaborative system needs to keep track on, while the latter are events which only exist for a short period of time and need not be saved or interpreted in some context. The paper presents the ABC architecture for activity-based collaboration, which is a hybrid architecture employing a stateless protocol for event handling. We describe the design and implementation of the ABC architecture and present a technical evaluation of it. This experiment demonstrates that even though this hybrid architecture rely on a server to dispatch accountable events, it still performs and scales very well. Hence, we conclude that hybrid architectures cannot be dismissed solely on a scalability and performance argument.

Published as: Jakob E. Bardram, Jonathan Bunde-Pedersen, and Martin Mogensen. Differentiating between accountable and ephemeral events in the abc hybrid architecture for activity-based collaboration.In *Proceedings of the IEEEInternational Conference on Collaborative Computing (CollaborateCom 2005)*, pages 168–176. IEEE Press, 2005.

1 Introduction

The ABC environment supports long-term pervasive collaboration [49] by providing support for asynchronous and synchronous activity sharing [17]. By modeling human activities as first-class computational entities in the architecture and the user interface, the ABC environment enables activity-based collaboration where;

- users can share an activity and hence work collaboratively on this activity,
- users can engage in *asynchronous collaboration* by taking turns in working on an activity,
- two or more users can engage in a concurrent *synchronous collaboration* by resuming an activity simultaneously,
- collaboration can be disconnected and reconnected in a pervasive and mobile environment, and users can continue working while disconnected, and
- users can do *collaborative roaming* between heterogeneous devices, i.e. can move a collaborative session from one device to another.

In addition, the ABC environment embody a programming framework and programming model, which allow programmers to create activity-based computing support tailored to their need and application areas. This also includes a programming model for using and extending the collaboration support in the ABC environment.

One of the key characteristics of the ABC environment is its hybrid collaborative architecture, which builds on a basic differentiation between what we have termed *accountable* and *ephemeral* events. In replicated collaborative architectures – which the ABC environment is an incarnation of – concurrency control between events on distributed clients is absolutely central in order to maintain correct behavior of the collaborative systems [135]. We use the term 'accountable' for this kind of distributed events, because the system needs to be accountable for the correctness and timing of these events in order to have a well-behaved collaborative system. Examples of accountable events are the classical text insert, move, and delete commands in collaborative editors or the state changes in general purpose frameworks like Corona [157] or GroupKit [150, 79]. There are, however, a range of other kinds of event which are not subject to the same kind of accountability. Such events are typically absolute values, independent of previous and subsequent events, and may even be missing or dismissed if needed. We call these events 'ephemeral' because they are short-lived and transient. Examples of such events are telepointer events, voice events, and other collaborative awareness events like the ones in the MAUI Toolkit [87].

The main goal of this paper is to discuss software architectures for collaborative systems with special focus on presenting a hybrid architecture for differentiating between accountable and ephemeral collaborative events. This is done in Section 2. Furthermore, we want to highlight the importance of the collaboration

protocol used – a topic which is seldom discussed in the CSCW literature. Especially, we want to demonstrate how a stateless collaborative protocol modeled over the HTTP protocol enables pervasive collaboration, including collaboration roaming, disconnection, re-joining, and heterogeneous devices. Section 3 presents the ABC collaborative architecture for activity-based collaboration. This architecture is an example of a hybrid architecture employing a stateless protocol for communication. Section 4 presents the current implementation of this hybrid architecture for activity-based collaboration, and Section 5 describes and discusses a technical evaluation of this architecture and its protocol. Finally, in Section 6 we relate our work to the works of others, before the paper is concluded in Section 7.

The main contributions of this paper are:

- We introduce the concepts of 'accountable' and 'ephemeral' events and discuss the architectural implications for collaborative systems. Especially, we point out how maintaining a server-based architecture for accountable events radically simplifies the creation of necessary collaboration mechanisms.
- We present a hybrid architecture and a stateless collaboration protocol for activity-based computing, which combines the benefits from the client-sever and the peer-to-peer software architectures.
- We evaluated this hybrid architecture and demonstrate that such an architecture scales and performs very well, despite the inclusion of a server for event propagation. Hence, we demonstrate that server-based collaboration infrastructures cannot be dismissed solely based on a scalability and performance argument.

2 Architectures for Collaborative Systems

An important issue in the design of a collaborative system is its architecture [58, 49], which determines the nature of the components of the system and the placement of these components on the computers of the various users participating in a collaborative session. Archetypical architectural patterns are the centralized, the replicated, the client-server, and the peer-to-peer architectures.

Centralized vs. Replicated Architectures

A traditional distinction is made between *centralized* and *replicated* architectures. In the centralized architecture, all of the users interact with a single program, which resides on one of the users' workstations. The program processes each user's input, and distributes all output to all of the users' workstations, thereby creating a copy of the user interface on each of these workstations. The major benefit of the centralized architecture is its simplicity and its ability to allow for collaboration transparency, i.e. that applications not created for collaboration can participate in collaborative sessions [104]. The main drawbacks of the centralized architecture

are its slow local responsiveness and that only one user can be active at the same time, thereby creating the need for explicit or implicit floor control mechanisms.

The replicated architecture, on the other hand, replicates the program on all of the users' workstations, and has each user interact with the local program replica. Traditionally, the main argument for using a replicated architecture is its support for fast responsiveness on the client workstation [79, 157]. The main challenge in such a replicated architecture is to synchronize the program replicas by somehow broadcasting the input of each user to all of them and to ensure consistency control mechanisms [135, 58] Another challenge is that applications can no longer be collaboration 'ignorant' but need to be aware about the fact that they may participate in collaborative sessions [104]. When creating a replicated architecture, two main architectural patterns typically exist: (i) the client-server architecture, and (ii) the peer-to-peer, or decentralized architecture. See figure 1.

Client-Server vs. Peer-to-Peer Architectures

The client-server architectural pattern is a well-known architecture for synchronizing replicated processes by distributing state information via a central server process. In collaborative systems, this server is furthermore used for synchronization, concurrency control, peer joining of clients, session management, and many other things. Normally, state information is only maintained on the server and then distributed to clients.

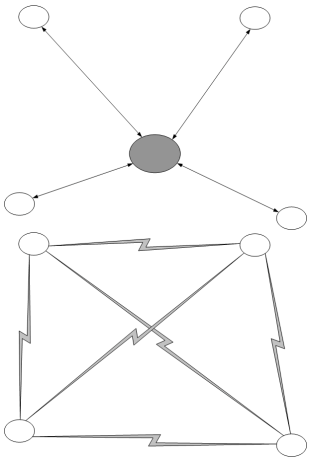


Figure 1 The Client-Server and the Peer-to-Peer Architectures

There are many advantages to the client-server architectural pattern. Concurrency control can be done in one place, i.e. on the server and there is hence no need for complex distributed concurrency algorithms. Since state and collaboration data only resides in one place, late joining problems and asynchronous collaboration problems are nearly non-existing [44]. Peer joining can be done easily by having the new peer contact the server directly; this also eliminates any need for peer discovery mechanisms. Also, because communication is flowing through

a central point in the system, only N (the number of peers) unicast connections are required.

Although advantages are bountiful in this architecture, there are also drawbacks. The architecture depends heavily on the common server, making the system very vulnerable to server-failures. Also scalability can be a problem, both with respect to central processing power and network bandwidth. The client-server architecture furthermore relies on an existing and stable infrastructure with a well-known server process, which eliminates any possibility of place independent collaboration [64].

To address some of these disadvantages to the client-server architecture, researchers have lately been suggesting peer-to-peer collaborative architectures [181]. A major research question seems to be if peer-to-peer architectures can be extended to support collaboration and not just file exchange. Collaborative peer-to-peer architectures replicate state information on all participating peers and employs distributed synchronization algorithms to keep the state synchronized.

These architectures have just about the opposite characteristics than the client-server architectures. Due to the distributed nature of state information and management, they have the advantages of robustness to network failures, fast responsiveness to user input [44], low network utilization, and the ability to enable collaboration anywhere [64].

But the advantages come at a high price. Complex synchronization algorithms, peer discovery methods, and search algorithms are often required [181, 44]. Furthermore, supporting late joining and asynchronous collaboration becomes highly complex because state and session information is no longer accessible in one place and hence has to be established by querying the whole network of peers. And since all peers need to communicate with all peers, N^2 unicast connections has to be established if not methods such as multicasting is used [76, 176]. Based on their design and implementation of a fully distributed peer-to-peer collaborative environment, Vogel et al. [76, 176] conclude that this induce a high degree of complexity. For example, the implementation of the concurrency algorithm alone took them three man months of work.

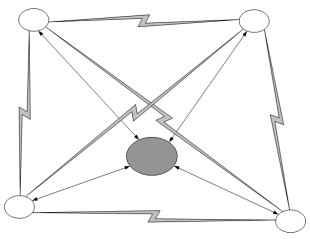


Figure 2 A hybrid architecture

Hybrid Architecture

Since the client-server and the distributed peer-to-peer architecture each has its advantages and disadvantages, it is interesting to investigate the possibility of creating semi-distributed [58] or hybrid [181] architectures. This is illustrated in Figure 2. When creating a hybrid architecture, the core question to answer is what part of the collaboration mechanisms would be using the client-server architecture and which parts should be using the peer-to-peer architecture. To help the designers make this decision, it is useful to differentiate between *accountable* and *ephemeral* events: accountable events should pass through a central server in order to be accounted for; the ephemeral event can be distributed peer-to-peer, since we do not want to save or access their data beyond the event. Typical accountable events passing through the server are;

- collaboration state information subject to concurrency control,
- session management, i.e. keeping track of users and what collaborative sessions they participate in, where, and on which kind of devices,
- peer or client management, i.e. keeping track of available peer in the network, the type of devices and their local resources, and
- general event passing and resource saving, which should be delivered in a reliable manner and should be stored for later use.

Ephemeral events passed on directly between peers include;

- stateless events, i.e. events which need not to be interpreted based on a series of prior events,
- multi-media events and streams, like voice and video,
- other kinds of bulk data, like files and other resources, and
- general event passing using a multicast i.e. unreliable network protocol.

There are several advantages to this hybrid architectural model. Since state management and consistency control is done on the server, simple – often optimistic – consistency algorithms can be employed. The task of accommodating late joiners and asynchronous collaboration is simple and there is no peer discovery required [181]. On the other hand, because a large portion of the collaborative events and awareness support is done peer-to-peer, responsiveness to user input and robustness to network failures are high. In addition, the system scales much better both with regard to network bandwidth and central processing power.

Still a hybrid architecture possesses some of the drawbacks from both the client-server and the peer-to-peer architectures. The architecture relies on a well-known server, making the system vulnerable to temporary server crashes or net-

work losses. In addition, the system may potentially suffer from scalability problems, due to the server bottleneck. Furthermore, an existing infrastructure is required, thereby eliminating the ability to support place independent collaboration. And $N + N^2$ unicast connections has to be established, if not multicasting is used.

However, these drawbacks can be reduced considerably in a number of ways. By using stateless communication protocols – like the HTTP protocol – the problem with network and server failures can be mitigated. By using client side caching and algorithms for re-joining a server after a disconnection, we can nearly eliminate the problem of temporary server crashes. The problem of scalability and server bottlenecks may be addressed with the right design of event flows. For example, based on earlier experiences with Rendezvous [133], Patterson et al. argue that "many synchronous multi-user applications (e.g., games, whiteboards, etc.) have only modest throughput requirements. [Therefore], it seems reasonable to centralize shared state" [134, p. 125].

In the rest of the paper we will present the ABC architecture for activity-based collaboration, which is a hybrid architecture using a stateless communication protocol, client side caching, and server-based algorithms for concurrency control and rejoining of disconnected peers.

3 The ABC Collaboration Architecture – A Replicated, Hybrid Architecture

The ABC Collaboration architecture is a replicated architecture, which is a hybrid between a client-server and a peer-to-peer architecture. The architecture is *replicated* due to the need for fast local responsiveness [79] and the need for supporting pervasive collaboration, i.e. that users can continue working if the network is reduced or disappears. Hence, we want to utilize the local resource and continue to support uses while disconnected. The key motivation for the *hybrid* architecture is the aim of combining the strengths of the client-server and the peer-to-peer architectures, as discussed above. For example, maintaining state synchronization and consistency is inherently easier when using a central server for concurrency control [135].

However, for large amount of data traffic, the peer-to-peer architecture scale much better since no single host need to relay all data. In the ABC architecture we hence rely on the activity server to synchronize *accountable* information about the state of the ongoing activity sharing session, and we rely on a peer-to-peer setup for distributing the *ephemeral* event data from the session. An example of accountable event data in the ABC platform is state information on the size and location of windows, whereas voice and telepointer data are examples of ephemeral event data. Looking at a typical activity sharing session between 2 users working for half an hour we have recorded the total amount of data floating between the clients and the server, and between the clients. These figures are shown in Table 1, which illustrates that the vast amount of data (99%) floats in the peer-to-peer connections. This is our main motivation for using the hybrid architecture – use a server

for maintaining consistency amongst state data and use a peer-to-peer setup for the large amount of voice and other collaboration awareness data.

Client-Server	*Telepointer (P2P)*	*Voice (P2P)*
68 KB	5.9 MB	146.5 MB
0.05%	3.85%	96.1%

Table 1 The distribution of network traffic in the ABC hybrid architecture.

In addition, the server is used for centralized *session management* which simplifies controlling and notifying participating users, and it is used for an easy handling of *late-comers* to a collaborative session. In the following, we will explain this in greater detail.

Overview

The ABC infrastructure for activity-based collaboration is illustrated in figure 1. This ABC environment is build to support *Activity-Based Computing*, which includes activity-centered service aggregation, activity roaming, activity adaptation, activity sharing, and context- and activity-awareness. It is beyond the limits of this paper to explain these concepts, but interested readers may consult [48, 17, 15].

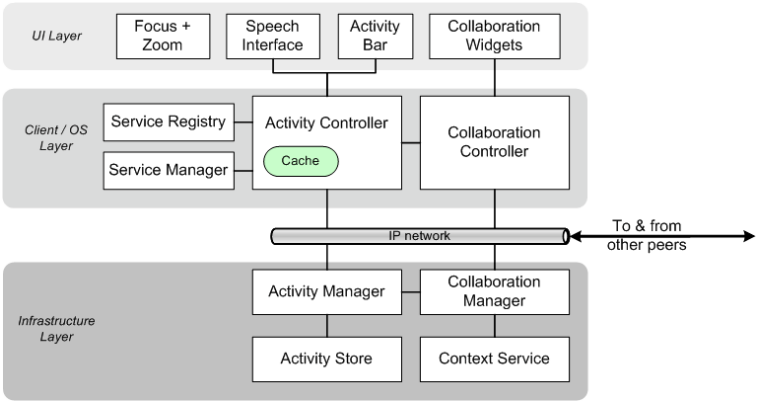


Figure 3 The ABC architecture for activity-based collaboration.

The ABC Framework consists of an infrastructure layer, a client layer, and an UI layer. The infrastructure layer is responsible for storing (the Activity Store) and runtime management of activities (the Activity Manager), for collaborative activity sharing (the Collaboration Manager), and for handling context-information (the Context Service). The infrastructure layer is typically deployed on a set of distributed powerful machines.

The client layer resides on each device participating in the ABC deployment. This layer is a middleware layer between the user-interface and the infrastructure layers. The core component is the Activity Controller, which is responsible for

controlling activities on the client and for communicating with the Activity Manager in the infrastructure layer. The State Manager is responsible for handling activity state when an activity is resumed and suspended on various clients. The Collaboration Controller is responsible for handling the collaborative activity sharing session, including peer-to-peer communication.

Different UI components reside on top of the ABC framework. For example, the Activity Bar replace the Windows Taskbar and allows the user to e.g. suspend and resume activities, create new activities, and to navigate within activities. These operations are also accessible via a speech interface component. Different components exist for different types of devices, like a desktop client, a wall-based client, a wearable client, or a table-based client. The UI layer also implements different collaboration widgets, like support for telepointers and voice links between participants in a synchronous activity sharing.

Infrastructure Layer

The collaboration manager keeps track on collaborative sessions. Peer joining hence becomes a simple task of contacting the server. The client opens a unicast connection to the server and they exchange session and client information. Even though the ABCP protocol (see the Protocol section) is stateless, we maintain an open unicast connection for performance reasons.

A key requirement in mobile and pervasive systems is the support for disconnected operation. Wireless links often provide only intermittent connectivity, so it is desirable to maintain a local state to enable the user to continue the work while disconnected, and to re-synchronize when the client gets opportunity to reconnect. In the ABC architecture, disconnected operation is supported by the local cache on each client. This cache will serve the client as long as the server is inaccessible. The client will automatically try to discover and re-connect to the server. When reconnected, the different local versions will be merged together by a merging algorithm running on the server. This newly merged activity will be distributed to the clients, thereby restoring one common activity state. This mechanism also mitigates the drawback of a single point-of-failure, since clients can continue working in case of server failure and rejoin their work once access to the server is restored.

The server also, in collaboration with the clients, ensures consistency among the participating peers. This is done by employing a server-based concurrency control algorithm. This makes it a simple task for the clients to maintain a consistent state. To be able to employ this concurrency scheme, it is required that all events affecting the state of the system, flows through the server – i.e. the accountable events. The server can parse all events and apply then to the server's representation of the system state. This makes late joining a peer to the simple task of transferring the state known by the server to the newly joining peer.

By having a stateless server and protocol, and by incorporating client side caches to enable the clients to work without server connectivity, we are able to use the hybrid architecture in transient environments. By distinguishing between ephemeral

and accountable events, and the way they are propagated around the system, we are able to off-load much data management from the server and in this way improve performance and scalability. We will return to this in Section 5.

Client Layer

When a client resumes an activity, this client is by default engaged in a collaborative session with other online users, who has resumed this activity. Hence, all state changes on the client is propagated to the server, which again distribute state changes to all participating clients. State change events are handled by the collaboration controller. The client has a unicast connection to the server and there is hence no need to know the other peers. The use of the local cache is nearly transparent to the client – the activity controller and the collaboration controller access only the local cache, which in turn handles connection and rejoining with the server. However, local 'connection lost' and 'connection reestablished' events are used in the client.

The collaboration controller also handles the ephemeral peer-to-peer events. This is done using multicast. Part of the activity sharing session stored on the server is information about multicast group IDs, which are used for peer-to-peer communication. Hence, the clients do not need to know each other. The collaboration controller has a programming interface which can be used to create new collaboration awareness widgets, which use the peer-to-peer multicast channels to communicate.

Protocol

As already discussed, an important part of the ABC architecture is the use of a stateless protocol both in client-server and peer-to-peer communication. The Activity-Based Computing Protocol (ABCP) is modeled like the HTTP protocol – a stateless, string-based, unicast socket protocol with URI and URL string syntax.

Basically, ABCP can be divided into three parts. The first part is used for activity management, i.e. to `GET`, `POST`, and `DELETE` activities using the activity manager. The second part is a general-purpose publish-subscribe event mechanism [68] where clients can subscribe to events and are notified on subsequent changes. This part of the protocol is used for collaborative event propagation, i.e. sending state change information and events on session changes, like when a particiapnt joins or leaves an activity. The third part of ABCP is dedicated to handling (storing and receiving) general-purpose resource information tied to an activity and hence a collaborative session. One particular important set of resources are the multicast group ID used in the peer-to-peer communication of ephemeral events.

The syntax of an ABCP request is:

```
ABCP/1.0 <method> /<argument>?(<key>=<value>&)*
```

For example, to get the last (latest) activity of a given user, the ABCP request is:

```
ABCP/1.0 GET /last_activity?user=bardram
```

To subscribe to notification events, the ABCP request is:

```
ABCP/1.0 SUBSCRIBE /host?name=firkin.daimi.au.dk&port=12345&activity=432&scope=meta
```

where the `scope` parameter is used to scope the amount of events. This is done to let devices with limited resources (network or cpu) to participate in collaborative sessions, but with reduced collaboration support. State and event information for accountable events use the Activity Markup Language (AML) which is an XML model for activity-based computing.

4 Implementation

The ABC architecture is currently implemented in a version 4.1. The infrastructure layer is implemented in Java and runs like an application server on one or more host machines. The client and UI layers are implemented in C# in the .NET framework and runs on the Windows XP operating system. The ABCP protocol and the AML data representation ensures platform and programming language independence.

Using the ephemeral event mechanism, we have implemented support for telepointers and voice links. The voice link implementation is a rather naive implementation that sends uncompressed sound samples and it hence does not scale to more than two users. Since multi-media research is not part of our core interest, we have decided not to address this at the moment.

The implementation of the consistency control algorithm is an optimistic extension of an ordered broadcast protocol, which ensures that all broadcasts are received in the same sequence at all sites [38]. Since all events passes through the server, the order can simply be established by time stamping the event on the server. The clients can then replay the event in the correct order. To avoid any locking, local events are executed immediately and then dispatched to the server. By replaying all events – also the client's own events – in correct order, consistency is maintained.

The merge algorithm used for activity and client rejoining is able to merge state information based on component IDs. In case of conflicting merges, all states are stores in the merged state. Hence, conflicts are propagated to the users to consider and resolve. This is a common strategy in other merging algorithms [100].

5 Evaluation

As argued above, the use of a client-server architecture for handling accountable events greatly simplified the design and implementation of a collaborative system. We claim that a hybrid architecture, where the server is only used for accountable events in combination with a stateless protocol, scales and performs adequately. In this section we describe an experiment to verify this claim.

In the experiment, we focus on the client-server part of the architecture since the server may become a bottleneck and the client is involved in maintaining consistency. Our experiment is furthermore divided into two parts: one part evaluating the server, focusing on *performance*, *reliability*, and *scalability*; and one part focusing on evaluating *consistency performance* of the client. More specifically, we are evaluating the mechanisms on both client and server which handles synchronization of state and event passing. The evaluation is performed in two separate experiments described in the following. In each experiment an activity is created and a number of users join and resume it. The experiment then performs a number of actions within the activity, simulating a real world collaborative situation. By having only one activity for large numbers of users, we stress both the server and client implementation because any change must be propagated to the all users and all changes automatically conflict making it harder for both server and client to agree on a consistent state.

Experiment 1 – Server Performance, Reliability, and Scalability

We measure the *performance* of the server as the time it takes to distribute a single event to one or many clients. The *scalability* is measured in terms of how increasing the number of users and clients affect the performance of the server. *Reliability* is measured as the number of times an event is lost (or extremely late). Reliability is the most critical factor – if the state of clients are allowed to diverge, then the system will in practice become useless. All clients run on a single machine in order for us to time the arrival of events consistently on all clients. To simulate that some more adverse network conditions (than having both computers connected via 100 Mbps) we included simulated delays in one run. We use the same delays as Chung and Dewan [49]: 72 ms simulating a user in Germany, 162 simulating a modem user in Germany and 370 ms simulating a user in India (assuming that we are located in the US).

Method

In the experiment we use a simulator program, which can simulate any number of clients. It replaces the UI layer in our architecture (see figure 1) and simulates each client by starting an activity controller in a separate process. The server runs on a Dell computer with a 3,0 GHz P4 processor and 1 Gb of RAM. This allow all activity data on the server to be stored in RAM. The simulator program and clients run on an IBM ThinkCentre with a P4 3,4 GHz processor and 1 Gb of RAM. The

two computers were connected via 100 Mbps Ethernet.

Experiment one is:

1. Initialize n clients.
2. Choose a random client and send a state change event through this client to the server. Record time T_0.
3. Wait for all n clients to receive the event. Record time T_1. If this does not happen, then the evaluation will have failed.
4. Record $T_1 - T_0$. Repeat steps 1–4 k times.
5. Record average time with n clients.
6. Increase n to $n + 1$ and go to step 1.

The maximum number of clients was set to 100, which in reality would mean that 100 users were collaborating in the same shared activity. k was set to 5.

Results

The results of experiment one is shown in Figure 4 as mean time as a function of the number of clients. The response time of the client-server round trip remains under 1 second throughout the experiment. Furthermore, all clients receive all events which can be seen from the fact that the graph never diverges. The graph is approximately linear, which indicates that the architecture scales well. Figure 4 also reveals that the simulated delays only affect the results by a constant factor.

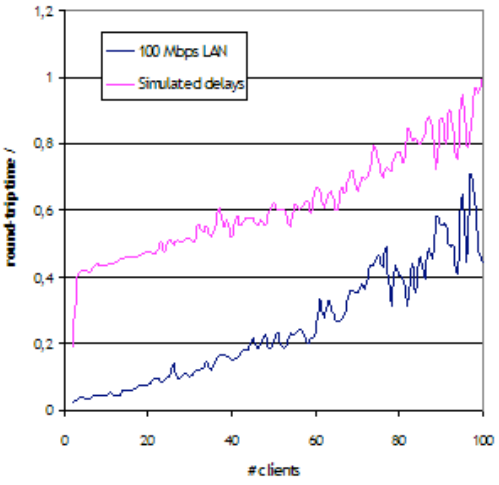


Figure 4 Round-trip time in seconds as a function of the number of clients

Experiment 2 – Client Consistency and Performance

We evaluated the client-side concurrency algorithm by measuring if it maintains *consistency* across an increasing number of clients. This is measured in terms of whether or not a long series of events are handled correctly such that all clients reach a consistent state. This experiment also allows us the measure how the architecture performs under simulated user activity while increasing the load.

Method

The same hardware setup was re-used from experiment one. In experiment one, only one isolated event was sent to the server and the simulator measured when this event was received back. To properly test the algorithm implemented in the client we need to have several simultaneous events from different clients on the network. Hence, in experiment two, the test was extended to simulate a number of different clients working concurrently in the same activity for some time and then wait for all clients to settle into a consistent state. The outline of experiment two is:

1. Initialize n clients.
2. Choose l random clients. For some time t, simulate user activity by inducing state change events on the activity controller, which is sent to the server. Record time T_0 after the last event has been sent.
3. Wait for all clients to report the same state. If this does not happen, the algorithm has a flaw or some event has not been propagated to all clients. Record time T_1.
4. Record time $T_1 - T_0$. Repeat steps 1-4 k times.
5. Record average time with n clients.
6. Increase n to $n + 1$ and go to step 1.

Again, the maximum number of client was 100, k was 5, and t was one second, which enabled the client to send between 30 and 70 events.

Results

The results of experiment two is shown in Figure 5, showing mean time as a function of the number of clients. There are no serious diverging peaks which indicates that the clients have agreed on a *consistent* state in all runs. The graph approximates a linear function which, together with the same fact about the graph in Figure 4, indicates that the client *performs* well and that the combined system also *scales* well.

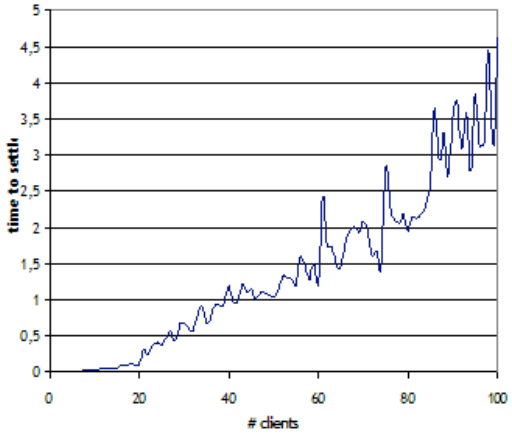


Figure 5 The time for all client information to settle into a consistent state as a function of the number of clients

Discussion

The main lesson from our evaluation is that a hybrid architecture scales and performs very well. Hence, in our work we have discovered that the often heard argument that server-based collaboration systems scale and perform insufficiently is not true. The server is capable of handling at least 100 users within a single activity and this is sufficient for the purpose to which this architecture was built. Furthermore, the experienced delays for all users in a 100-user session is less than one second (including simulated displays). It is difficult to say if this is an acceptable delay since it depends very much on the applications involved. In a simple chat-application a one second delay is acceptable while this may not be the case with a surgical simulation. Also, estimating the maximum number of users is troublesome – but the nature of real-time collaborative work arguable imposes a limit which is well below the 100 limit used in our experiments.

Putting in a server strongly simplified handling core issues like concurrency control, session management, event notification, integration, and later-comers to a session. For these reasons, a server-based collaboration manager still seems to be a valid choice. In our case, the vast majority of data was stateless telepointer and voice data, which was sent directly between peers (see Table 1).

The two technical experiments reported above is devoid of user-interface considerations, and we have not considered the time it takes for programs to respond to events, but simply the difference between a client sending an event and the rest of clients receiving them. However, time for user-interface updates are independent of the underlying communication and distribution mechanisms, which is the main subject of this paper.

6 Related Work

As already discussed the design of the ABC hybrid architecture and its differentiation between accountable and ephemeral events is inspired by most of the work on collaborative applications, systems, and frameworks since the early 90'es within CSCW. Systems like LIZA, Rendezvous, Suite, and Visual Obliq use a centralized architecture, while DistEdit, GroupDesign, GroupKit, the Grove text editor, and Corona uses a replicated and distributed architecture. Some of these replicated architectures use a client-server approach for session management, consistency control, and client re-joining, while others use a peer-to-peer architecture in order allow ad-hoc cooperation and avoiding single-point of failures.

GroupKit [150, 79], for example, has a 'mostly' replicated, peer-to-peer architecture. They maintain a central node called the 'registrar', but this node is only used as a connection point for session creation. Hence, this server only plays the role of peer discovery discussed above. Application programmers in GroupKit "must attend to issues such as concurrency control, fault-tolerance, and synchronization" [79, p. 142] which, as illustrated by Vogel et al. [176], may imply a tremendous effort in a peer-to-peer architecture.

In many respects, the collaboration architecture of ABC extends the notification sever and its NSTP protocol [134]. In ABC, the design of the generic event mechanism in the ABCP protocol resembles the NSTP protocol – this is a semantic-free event mechanisms where the interpretation of the semantics resides with the application using it. In addition, however, the ABC collaboration environment provides the ephemeral multicast peer-to-peer event channel – a feature that the creators of the NSTP protocol state as an extension they were considering, but to our knowledge never implemented.

To our knowledge, there are no other examples of a hybrid model between a client-server and a peer-to-peer architecture which utilize a stateless collaboration protocol. There has been some more recent work on architectural adaptation to create dynamically changing between centralized, replicated, and hybrid collaboration architectures [49]. Because the ABC environment does not support such kind of adaptation, this work has minor relation to ours at the moment. We are, however, working on incorporating architectural adaptation into the ABC environment. We find this especially useful in creating better support for collaboration roaming by gracefully adapting between a centralized and replicated architecture as people move between devices.

Another line of future research we are pursuing is to enable ad-hoc peer-to-peer collaboration [64], which we recognize as an important aspect of activity-based collaboration. We would argue, however, that our current hybrid architecture would be sufficient to support this by deploying activity servers on all clients, have them discover each other dynamically, and implement an election algorithm for choosing one server to use. Because the ABCP protocol already supports re-joining, work on a decentralized peer server can later be re-joined with another, more central server. Hence, we would argue that our current hybrid architecture is rather robust because it supports many modes of collaboration – asynchronous,

synchronous, and ad-hoc collaboration

7 Conclusion

This paper has addressed software architectures for collaborative systems. Based on our experience with the ABC environment we have argued that a replicated, hybrid architecture is well-suited for a range of collaborative systems. A hybrid between a client-server and a peer-to-peer architecture can exploit the advantages of both these architectural patterns and route collaborative event either peer-to-peer or via a server, when appropriate. We introduced the terms 'accountable' and 'ephemeral' events to differentiate between two important type of events in almost all collaborative systems. Accountable events are events which we need to keep track on, i.e. where we need to ensure consistency control, event identification, state, and history. Designing and implementing collaborative systems which use a server for accountable events is much more simple than a distributed peer-to-peer architecture. On the other hand, using peer-to-peer communication channels for ephemeral events is more appropriate, since these events are not critical, may arrive out of order, are not interdependent, and may even be dismissed or lost.

We presented the ABC collaborative infrastructure which supports activity-based collaboration based on a hybrid architecture. In ABC, accountable events include activity state and user session events, whereas ephemeral events include telepointers and voice link data. Handling concurrency control, activity state merging, session management, late comers, and disconnected collaboration becomes inherently more simple using a server. For example, it took us three man *days* to implement our concurrency control algorithm – not three man month as reported by Vogel et al. [176].

Two well-known arguments for not using a server-based architecture in collaborative systems are the single-point-of-failure argument, and the scalability and performance argument. The ABC environment mitigates the single-point-of-failure challenge by enabling client-side caching and we have in a technical experiment demonstrated that the ABC hybrid architecture scale and performs very well – far beyond the number of users it is targeted for.

Currently we are working on extending the hybrid architecture to support ad-hoc peer-to-peer collaboration, while maintaining a server-based hybrid architecture. Because of it general applicability, its simplicity, and its adequate scalability, we find hybrid architectures for collaborative system very useful.

Acknowledgments

The research presented in this paper has been funded by ISIS Katrinebjerg competence center, Aarhus, Denmark. Currently, the ABC project is funded by the Danish Research Council under the NABIIT program. Henrik B. Christensen has been much involved in the early research on activity-based computing.

Paper VI

The ABC Adaptive Fusion Architecture

Jonathan Bunde-Pedersen Martin Mogensen Jakob E. Bardram

Abstract

Contemporary distributed collaborative systems tend to utilize either a client-server or a pure peer-to-peer paradigm. A client-server solution may potentially spawn direct connections between the clients to offload the server thereby creating a hybrid architecture. A pure peer-to-peer paradigm may on the other hand fully eliminate the need for a server. However, some situations call for the strengths of both approaches without relying on either of them. A system might both be used in environments where an infrastructure is present and in environments where it is not. In this paper we present an architecture and early implementation of a system capable of adapting to its operating environment, choosing the best fit combination of the client-server and peer-to-peer architectures. The architecture creates a seamless integration between a centralized hybrid architecture and a decentralized architecture, relying on what we have termed *Peer-to-peer Distributed Shared Objects* (PDSO). The proposed solution has been implemented and early evaluation has begun. Furthermore, the approach has been utilized to create a real distributed collaborative system for collaboration in hospitals.

1 Introduction

Our research is based on long term empirical studies of work in the healthcare domain [1], and we use this as our application area. The Activity-Based Computing (ABC) middleware [19, 48,1] is a generic middleware component which allows

Published as: Jonathan Bunde-Pedersen, Martin Mogensen, and Jakob E. Bardram. The ABC Adaptive Fusion Architecture. In *MPAC'06: Proceedings of the 4th international workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC2006)*, New York, NY, USA, 2006. ACM Press.

users and developers to organize their applications into activities, which form the basis for both synchronous and asynchronous collaboration. This article focuses on the distributed architecture of this middleware, which provides a novel adaptation mechanism between pure decentralized operation and a centralized hybrid approach.

Different suggestions to improve on the shortcomings of existing RPC-style interaction with remote single-object in RMI, CORBA, .NET Remoting, and DCOM have been suggested. For example, asynchronous RPC [182, 107], CORBA Event and Notification Services [174], and publish-subscribe [131, 130]. One specific approach to improve Java RMI is to support dynamic caching of shared objects on the accessing nodes, as done in Javanaise [83]. All of these approaches mitigate the challenges of unstable intermitted network connections and lack of scalability and performance but does, as such, not directly support object replication. The main contribution of our work, is the ability to enable every unit in the system to continue asynchronous collaboration even in situations without network connectivity, due to the full replication of state on all peers. Combined with the ability to adapt the architecture to the execution environment, this gives a very robust and scalable communication platform for building distributed collaborative systems.

The rest of this paper is organized as follows: First we will give a brief introduction to the concepts of ABC. Section 2 will present a scenario, which is used to motivate our fusion architecture. Section 3 presents the architecture itself, while Section 4 contains future work. Lastly, Section 5 gives a conclusion.

On Activity-Based Computing

Activity-Based Computing is a research area devoted to helping users cope with the increased complexity of daily computer use as well as providing a more suited interaction metaphor (than the desktop) for pervasive environments. The main concept driving ABC research is that of *activities* [48,I].

Computational activities as a concept are the grouping of applications and files into logical bundles (or activities). This allows a user to easily switch between different work-tasks by resuming and suspending a bundle as a whole.

A *human activity* is for instance the task of "writing a paper", and the corresponding computational activity is then a description of the applications involved e.g. a text-editor, a bibliographical tool, and a browser along with their state. The state of an application is the information which is deemed relevant to save across instantiations or state which should be synchronized in collaborative sessions. For example a text-editor may need the filename and position of the cursor as its state.

We deal with computational activities which can be created, suspended, and resumed on any activity-enabled computing device in the environment at any point in time. They can be shared asynchronously, or shared among several persons working simultaneuosly.

Our previous work has resulted in a middleware which supports this interac-

tion with computational activities, providing mechanisms for organising ones work and applications into activities. Activities are stateful and when we talk about the state of an activity we refer to the already mentioned state of all the individual applications. Tracking this state allows us to *suspend* an activity by persisting the entire state and exiting all applications. Later, the persisted state can be retrieved and the activity *resumed* by instantiating the contained applications and restoring their state. Furthermore, we allow multiple users to participate in the same activity, synchronising the state of all applications in the activity across two or more machines.

2 Scenario

The following scenario presents a sequence of events which illustrate the levels of connectivity and access to resources which may occur in an emergency scenario, and which we seek to support with the architecture described in this paper.

> *Hansen, a paramedic, is on duty when two cars collide on the highway. He and his co-driver are based at a hospital 10 minutes away and they speed towards the crash site. Underway Hansen prepares himself by loading a basic emergency activity onto his PDA. This activity is prepared with applications into which Hansen can plot various information about injuries and otherwise relevant medical information. Once at the site Hansen delivers first aid and records information about the patients on his PDA. All emergency personnel have access to editing and adding information to each patient. When he gets in range of the ambulance the information is synchronized with a client inside the vehicle which immediately forwards the data through a GPRS or similar connection to the hospital. Sensors attached to the patients are included as part of the activity, and data is streamed over this connection as well as they return to the hospital. Meanwhile, doctors have been monitoring incoming changes to the activity and have begun to populate their local copies with information about the medical history of the patients. The doctors and other medical staff are thus prepared to take over the activity which is completely transferred and merged with the hospital infrastructure. The activity is now stored on a server in the hospital and further treatment information is recorded directly into it.*

The scenario illustrates the use of activities in three different localities, each having a characteristic level of infrastructure access. The first level is found in the field where access to basic resources is, at best, scarce. Here, the roaming devices have intermittent infrastructure connectivity as they move in and out of range of the ambulance. It is not possible to rely on network connections and those connections which may be established have very low bandwidth. We find the second level of access inside or close to the ambulance, where the paramedic equipment may be hooked into the LAN of the vehicle which again may be connected with a remote infrastructure (the hospital) through a GPRS or similar service. The last level of access is found when the device gets in connectivity range of a stable infrastructure. Here, full access to the remote resources can be established and we may

rely on all connections. In all three cases the roaming device may have transient connections to peers close by and may at these sessions exchange information in a peer-to-peer fashion.

In the following section we will present our architecture for the ABC middleware and how we support the above scenario.

3 Architecture

An important issue in realising the aforementioned scenario, is the choice of architecture to base the system upon. We will begin by describing the architecture of the core ABC middleware. Then we will give two sections describing the architecture of the communication modules; the hybrid architecture for accessing an infrastructure tier and the distributed object scheme which is used for peer-to-peer communication. Finally, we will present how these two modules are fused together and how the adaptation to a given environment proceeds.

Overall architecture

The core of the ABC middleware has a basic two tiered architecture. An infrastructure layer provides support for persistence, event propagation, and methods for synchronizing activities. This layer may reside on one or more servers and is independent of the client layer. The infrastructure layer is responsible for persisting activities through the *Activity Store* component, for managing runtime state and events through the *Activity Manager* and for distributing state-changes and other collaboration-related data through the *Collaboration Manager*. The *Context Service* component keeps track of users and machines as well as their context. The layer is commonly deployed on one or more servers – physically separated from the client layer.

The client layer runs on all active devices and interfaces with the local operating system, applications and data. The *Activity Controller* provides handles for resuming and suspending an activity, and manages connections to the infrastructure as well as to other clients. It is also within this layer we have implemented the peer-to-peer communication mechanisms and the distributed object scheme.

The connection to the infrastructure uses a stateless HTTP-like protocol called ABCP. The ABCP protocol is an XML based protocol with storage methods GET and POST as well event-based communication through PUBLISH and SUBSCRIBE methods. The storage methods act as their HTTP counterparts and allows for retrieval and posting of activity-data. Activities are described using an XML format so they are highly portable. The client itself also maintains a cache which permits users to keep working in an activity in situations no external connections are possible. Once a connection has been re-established, the cache will synchronize with the infrastructure once again.

The *Collaboration Controller* allows an activity to be shared. When sharing an

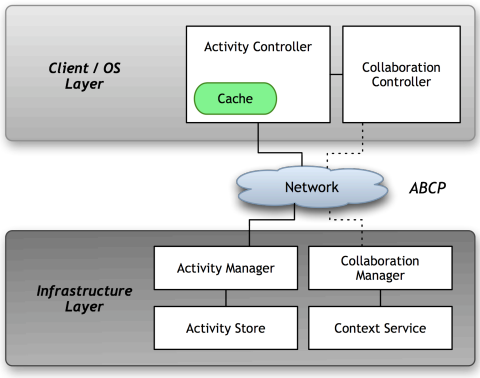


Figure 1 The overall architecture.

activity the state of all services in the activity is synchronized on all participating devices.

Centralized hybrid

If an infrastructure layer is available, then the middleware functions as a centralized hybrid architecture. The infrastructure layer is used for storage of vital information while non-vital is distributed in a peer-to-peer fashion. This approach ensures delivery guarantees from the transport layer by making a key differentiation between what we have termed as *accountable and* ephemeral events [V]. In replicated collaborative architectures concurrency control between events on distributed clients is absolutely central in order to maintain correct behavior of the distributed system [135].

We use the term 'accountable' for events relying on concurrency control, because the system needs to be accountable for the correctness and timing of these events in order to be considered as a well-behaved collaborative system. Examples of accountable events are the classical text insert, move, and delete commands in collaborative editors or the state changes in general purpose frameworks like Corona [157] or GroupKit [150, 79]. An IP-based infrastructure would use TCP or reliable multicast to distribute such events. Accountable events are thus routed through the infrastructure and made subject to concurrency control before being distributed further.

There are, however, a range of other kinds of events which are not subject to the same kind of accountability. Such events are typically absolute values, independent of previous and subsequent events, and may even be missing or dismissed if needed. We call these events 'ephemeral' because they are short-lived and transient. Examples of such events are telepointer events, voice events, and other collaborative awareness events like the ones in the MAUI Toolkit [87]. An IP-based infrastructure would typically use multicast datagrams to distribute such events. Ephemeral events are thus distributed optimistically in a peer-to-peer fashion.

There are several advantages in using a hybrid architectural model and then dif-

ferentiating between *accountable* and *ephemerable* events. Since state management and concurrency control is done on the server, simple – often optimistic – concurrency algorithms can be employed. The task of accommodating late joiners and asynchronous collaboration is simple also and there is no peer discovery required [181]. On the other hand, because a large portion of the collaborative events and awareness support is done peer-to-peer, responsiveness to user input and robustness to network failures are high. In addition, the system scales much better both with regard to network bandwidth and central processing power [V].

Still a hybrid architecture possesses some of the drawbacks from both the client-server and the peer-to-peer architectures. However, these can be reduced considerably in a number of ways.

Firstly, by using stateless communication protocols, as we do between client and infrastructure layers, the problem with network and server failures can be mitigated. If a client fails, the server is not left with any state to maintain and the client can reconnect seamlessly once it recovers.

Secondly, by using client side caching and algorithms for re-joining a server after a disconnection, we have nearly eliminated the problem of temporary server crashes. If the infrastructure layer fails, clients may be unable to contact the server for a while, but communication can be resumed and clients re-synchronized once the infrastructure has recovered.

Decentralized ad-hoc

The centralized hybrid approach still relies on a server to synchronize and to enable live sharing of data. In order to enable total infrastructure independent operation we need an additional pure peer-to-peer scheme. We have created a fully decentralized architecture, relying on what we have termed *Peer-to-peer Distributed Shared Objects* (PDSO), since we rely on local object replicas keeping themselves synchronized using an underlying peer-to-peer infrastructure. This approach is an extension of existing research on distributed shared objects. The fundamental principles behind the design of our approach are:

Physical distribution Instead of viewing a distributed object as an entity running on a single host with others accessing it remotely, we physically distribute a copy of the object to all hosts using this object in an application. Hence, applications access and use objects as local objects which ensures fast responsiveness. Objects are distributed on creation (remote instantiation) and removed from the local address space on deletion (distributed garbage collection).

Synchronized objects The state of any distributed shared object is kept synchronized in real time, if possible. Hence, state changes are propagated to all object replicas. State synchronization is handled by the underlying infrastructure, but the objects themselves are involved in potential conflict resolution, using domain specific conflict resolution algorithms.

Peer-to-peer update Physically distributed objects rely on a peer-to-peer – or object-to-object – synchronization strategy. Hence, no central entities like an object broker or an object registry are involved in object registration or lookup. Each object is responsible for looking up and synchronizing with its replicas. This principle makes distributed programming simple from the developers point of view since there are no configuration overhead associated with development and deployment.

Responsive Objects are used in highly interactive applications and need to embody a fast update protocol. This rules out pessimistic concurrency control which typically uses some kind of distributed transactional scheme [161, 115, 157] or distributed locks [106].

Distribution-aware Objects are distribution-aware. This means that a shared object must be declared to be distributed, must handle potential conflict resolution, and must consider the kind of delivery guarantees wanted in the network transport layer.

The principles involved in peer-to-peer distributed object sharing is illustrated in Figure 2, showing a distributed object with a replica in four different address spaces (A1–A4), using object-to-object communication pathways to keep the replicas synchronized and sending remote instantiation and garbage collection events.

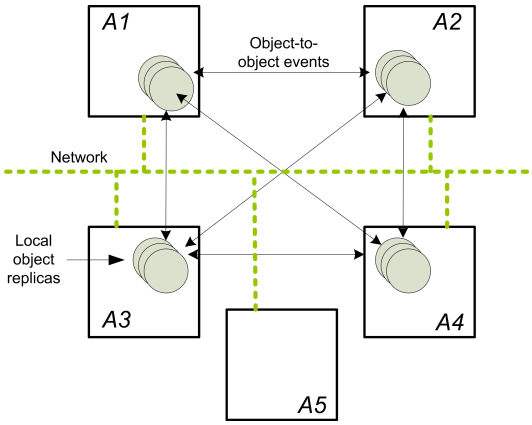


Figure 2 A set of peer-to-peer distributed shared objects (PDSO) distributed over four address spaces (A1–A4). Each address space holds a local replica of the object which is synchronized by object-to-object eventing. Address space A5 does not currently participate in the object sharing but may join one or more of the objects.

The main idea is that a distributed object, called a Peer-to-peer Distributed Shared Object (PDSO) consists of several local replicas that keep their state synchronized. Each local replica is identified by an Object Identifier (OID), a PDSO consists of the set of local replicas with the same OID. A set of PDSOs can be tied together by use of distributed variables; we call such a set a group.

To be more precise, we are using the following terms:

OID Object Identifier. The OID is used to name a single instance of a local object replica. Several local object replicas can have the same OID, but not within the same namespace.

$PDSO$ Peer-to-peer Distributed Shared Object. A set of local object replicas, that keep their state synchronized. A $PDSO$ is defined as the set of local object replicas named by the same OID. I.e. $PDSO(s) = \{local\ replicas\ x | OID(x) = s\}$

$Group$ A set of $PDSO$s, defined by the transitive closure of a specified $PDSO$ x. I.e. all PDSOs in the object graph that can be reached from x.
$Group(PDSO(x)) = \{PDSO(y) | there\ is\ a\ path\ from\ PDSO(x)\ to\ PDSO(y)\ in\ the\ object\ graph\}$.

Figure 3 shows five PDSOs distributed over four address spaces. The PDSOs are named A, B, C, D, and E respectively. Each distributed object is comprised of several local replicas, all named with the same object identifier (OID). The local replicas comprising the PDSO named B have been highlighted. Also shown in the figure are three groups, namely $Group(A)$, $Group(C)$, and $Group(D)$. The groups are the transitive closure of the named PDSO. $Group(A)$ is therefore comprised of $PDSO(A)$ and $PDSO(B)$, whereas $Group(D)$ equals $PDSO(D)$ because the edges in the object graph is directed. Notice also that two peers, address space A2 and A4, are members of more than one group.

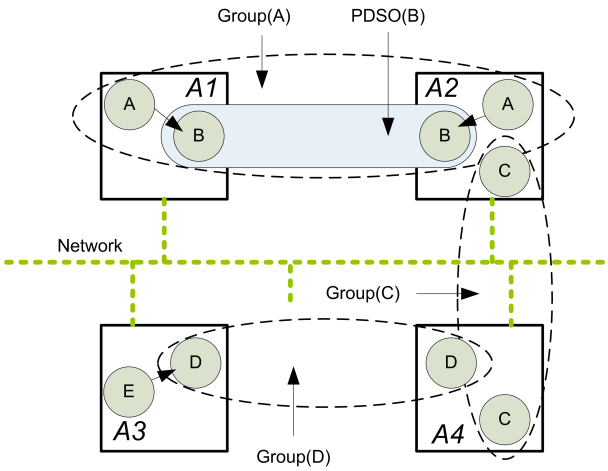


Figure 3 Five PDSOs (A–E) distributed over four address spaces (A1–A4). Each address space hold a local replica of the PDSOs in the groups the peer is member of.

Using the concept of peer-to-peer distributed shared objects, we are able to create a fully decentralized communication architecture, capable of supporting the same types of collaboration as the centralized hybrid architecture. Delivery guarantees are handled in the same manner as the centralized hybrid architecture - by distinguishing between *accountable* and *ephemeral* communication. The decentralized communication architecture enables us to support ad-hoc collaboration in environments without existing infrastructure.

Adaptive fusion

The infrastructure needed to support the scenario of 2, has to incorporate properties from both the centralized and the decentralized architectures. Therefore, we have created a fusion architecture, capable of adapting to the environment and delivering the properties needed in the given setting. The architecture adapts its communication strategy based on information obtained from the incorporated service discovery mechanisms. It will try to provide as good a platform for collaboration as possible.

In the scenario the paramedics move around in the real world, thereby changing the environment in which their personal digital devices reside. At the hospital, the supporting infrastructure is fully evolved, including high-speed, reliable network connections. This permits the use of centralized architectures, possible utilizing direct communication between the clients, thereby leveraging the server load. Moving away from the hospital, the environment changes. Network connections decrease in both bandwidth and stability. This means that collaborative widgets, such as tele-pointers and voice-links, has to give way to ensure bandwidth for essential data communication.

Upon arrival at the accident scene, infrastructure may be non-existing or unreliable. The only infrastructure at the accident scene, is the infrastructure that the paramedics bring themselves. This means that collaborative middleware has to utilize pure decentralized architectures in order to guarantee operation. The following sections will give an in depth description of the different modes the architecture will operate in at the different environments.

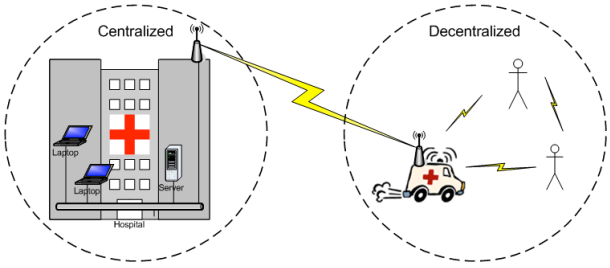


Figure 4 The Adaptive Fusion Architecture is capable of adapting to the environment. Based on service discovery mechanisms and resource awareness, the infrastructure will morph from centralized to decentralized over various fusion architectures.

The implemented middleware uses a service discovery mechanism, which will tell the system which resources are available in the environment. Based on these information, the system will choose the best mode of communication. In the first phase of the scenario, the paramedics are located at the hospital, which means, that they can access the stable infrastructure of the hospital. The service discovery mechanism will obtain a reference to the server and the different clients will be set up according to the earlier presented hybrid architecture. The collaboration enabled in this setting is fully evolved. State changes will flow through the server and be forwarded to other clients participating in the collaboration. Tele-pointers,

voice-links, and other collaboration widgets will flow directly peer-to-peer between the clients. This setup is very stable and easy to deploy and maintain. System administrators can guarantee quality of service and data backup mechanisms, thereby easily accommodating the legal and political requirements for data storage on a hospital.

As the paramedics enter the ambulance, they leave the stable environment of the hospital, and enter another environment where network connections are more transient and bandwidths are low. In this setting the paramedics may use the client residing in the ambulance. This client will be a member of the server based network, but due to the lack of bandwidth, only a degraded form of collaboration is supported. As in the hospital state changes will be sent through the server, but the client will not be able to establish tele-pointers, voice-links, and other collaborative widgets, which uses ephemeral events as their distribution mechanism. If the client should loose the network connection, and thus the connection to the server, the paramedics are able to continue their work on a local copy of the activity provided by the *cache* component. Upon reconnection, the client will automatically merge the work done into the activity residing on the server.

At the accident scene, the middleware cannot rely on any existing infrastructure. The infrastructure may have been destroyed in the accident or there may not be any infrastructure there at all. The paramedics bring their own infrastructure based in the ambulance, but perhaps the paramedics have to move away from the ambulance and thereby also away from the infrastructure. In such a setting, the personal digital devices of the paramedics will engage in an ad-hoc network. The service discovery mechanisms will be unable to detect a server, but be able to detect the other personal digital devices in the area. Using the earlier described *peer-to-peer distributed shared objects*, the middleware will utilize a completely decentralized architecture, enabling collaboration between the paramedics at the accident scene.

The collaboration supported will be exactly the same as at the hospital, including state changes, voice-links, tele-pointers, etc. The difference between this scene and the hospital scene, is that both ephemeral and accountable events will be distributed in a peer-to-peer manner. Even if the different clients in the ad-hoc network should loose connectivity for a period of time, the individual clients will offer the ability to continue work while disconnected, and upon reconnection the incorporated synchronization mechanisms will resynchronize the network. If one of the clients in the ad-hoc network should come within range of the ambulance, the client in the ambulance will join the ad-hoc network, while still being a client in the server based network. This enables the ambulance client to act as a bridge, relaying state changes in the decentralized network to the centralized network and the other way around. This means that it is possible for a client in the decentralized network to collaborate with a client in the centralized network, creating a network architecture which is a fusion of a centralized and a decentralized architecture.

4 Future work

Evaluation of the fusion of these schemes as well as the adaptivity mechanism is still future work. Both schemes have been evaluated separately and these results show that: The centralized hybrid scheme scales reasonable well (to at least 50 simultaneous users) on a standard 10Mbps LAN - and the differentiation between *accountable* and *ephemerable* events leads to a significant reduction in server-based traffic. Rejoining a client is accomplished almost instantly because only accumulated changes are used to resynchronize the state of the client, so the amount of data needed to be transferred remains small. A detailed technical evaluation is available in [V]. The Peer-to-peer Distributed Shared Object scheme has been evaluated along four dimensions: Completeness, Complexity, Performance, and Utility. The results have been reported elsewhere [122], and show satisfying results in every aspect.

Ongoing evaluation mainly focuses on the utility aspects of utilizing the peer-to-peer distributed objects as platform for various pervasive applications.

5 Conclusion

This paper presented a fusion architecture for the ABC middleware, which enables activity-based clients to collaborate in both environments with an infrastructure as well as in those without. The adaptation mechanism ensure that work can continue in both environments and that synchronous collaborative features degrade gracefully. The architecture furthermore demonstrates a novel way of incorporating two disparate communication modes without relying on a hybrid infrastructure.

Paper VII

A Flexible Infrastructure and Programming Interface for Distributed Application Spaces

Jonathan Bunde-Pedersen Jakob E. Bardram

Abstract

Interaction between disparate applications executing on a single machine is not very common occurrence, and the same can be said for applications distributed on multiple machines. Applications are often considered as more or less monolithic entities, each running within their own private space, not affected by the state of other applications. This paper presents an infrastructure component called AEXO which enables *distributed application spaces* to be formed. These spaces form the virtual environment in which otherwise disparate applications can share data and functionality, and from which highly adaptive pervasive computing end-user applications can be constructed. The infrastructure is based minimal core which manages a distributed data-structure, and is extensible via dynamic loading of components. Furthermore, we present a programming interface for application spaces which enable developers to annotate parts of their applications containing data which is then made public and shared through the AEXO infrastructure. Two proof-of-concept implementations of applications spaces are also presented, both of which were both deployed for limited real-world use.

1 Introduction

Much pervasive computing research share the vision of enabling a computing model in which applications supporting users' tasks are no longer tied to one (personal)

Submitted for: COMSWARE 2009. The Fourth International Conference on COMmunication System softWAre and middlewaRE.

device, but are distributed across several heterogenous devices, which work in concert to support these tasks. For example, to allow users in a smart meeting room to seamlessly use their own portal devices in tight integration with devices like interactive boards and tabletops inside the meeting space.

Building such distributed applications is, however, still very challenging and a significant body of research is addressing more generic infrastructures for enabling this new kind of computing environment. Examples include support for control re-direction [95, 36] and view-redirections [137, 170] in multi-display environments; context distribution between applications [152, 16], application re-direction between devices [149, 75, 22].

A common theme to these environments is, however, that they do not go significantly beyond the existing model of having one application tied to one device in a given period of time; in infrastructures for control re-direction (e.g. iROS), the applications are running on each device and uses the event heap to coordinate input events; context-aware infrastructures (e.g. the Context Toolkit [152] or JCAF [16]) are used to distribute data and events on context changes to applications running on different devices; and the whole idea behind application re-direction (e.g. Gaia [75], Aura [149], or ABC [22]) is to move running applications between devices. In essence, the application and its software components (e.g. model, views, and controllers) are not distributed across several devices, but the same application is running on each device and to some degree kept synchronized.

This is not surprising. It is still challenging for application developers to create applications which are truly distributed across a set of devices. For example, an application which is running partly on a large interactive display, partly on a mobile phone, and which enables these two devices to work as one application when they are, in some sense, close to each other. To realize this, we would need to build and deploy components like model, views, and controllers, that are distributed but yet act, as if they ran in the same address space.

In this paper, we introduce the concept of a *distributed application space* (DAS) in which several devices run separate partitions of the same application, and in which this application partition can be dynamically reconfigured based on e.g. the discovery and exit of devices. Section 2 defines the DAS concept and provides an example.

Section 3 presents AEXO, which is a runtime and programming infrastructure for creating distributed application spaces. AEXO builds on a core data- and event-distribution model which allows networked applications to share and distribute data (e.g. the application model) as well as the logic for managing and monitoring this data (e.g. the application controller). AEXO handles the intricate details of managing the ad-hoc formation of a heterogenous set of devices. It is highly extensible in a device independent manner; new functionality can be added to the application space at runtime, which then becomes available across the entire distributed space to all partitions in the application space.

Section 4 describes the application space programming interface (ASPI) that enable developers to create distributed applications for e.g. a multi-display envi-

ronments. We will present the binding-scheme for the .NET ASPI, but we also provide a more basic integration mechanism for both the Java and iPhone platforms.

Section 5 describes two distributed application spaces which have been built using AEXO. The first application space targets meeting room support in a smart space environment similar to e.g. Gaia [149]. The second focus on activity-based computing in a hospital environment, enabling mobile clinicians to access clinical data from public displays distributed across a hospital. By showing how the infrastructure and the ASPI support the development of such distributed application spaces, these two systems are proof-of-concepts on the utility of AEXO .

Section 6 contains related work and the paper is concluded in Section 7.

The contribution of this paper is hence to (i) introduce the concept of a distributed application space, i.e. a collection of end-user applications which span several heterogenous devices; (ii) to present AEXO as a new type of runtime and programming environment which enables the creation of such distributed application spaces; and (iii) to present a proof-of-concept for the utility of AEXO by presenting two different types of distributed application spaces built using the infrastructure.

2 Distributed Application Spaces

We define a *Distributed Application Space* (DAS) as a set of software components distributed across several networked devices, which together form one logical end-user computer system. Such components typically include end-user *views*, like graphical user interfaces; *controllers* that implements the application- and business-logic; and *models* holding shared data.

These software components and devices may dynamically reconfigure themselves by e.g. utilizing the functionality of shared controller component, by utilizing new views for displaying data, or by mounting and accessing shared data models. The combined functionality of the DAS is thus the sum of the components running in the total set of available devices, and the functionality may hence vary dynamically according to the number of devices and components participating at a given point of time.

Figure 1 shows an example of a DAS. From an end-user point of view, this DAS is a distributed image capture and access application. Users can view images stored on their mobile device (device A); when entering a meeting room with a public display (device B), this display becomes able to access and show public pictures from nearby mobile devices; and from their mobile devices (A) users are able to use the public camera (mounted on device B) to take pictures of the entire room with its participants. This example illustrates a DAS which is distributed across several devices (As and Bs) and how the functionality of this DAS is dynamically configured according to devices, components, and data being brought together.

Figure 1 also illustrates how AEXO is used to realize this distributed capture and access DAS. The DAS is divided into *application partitions*, each of which may run

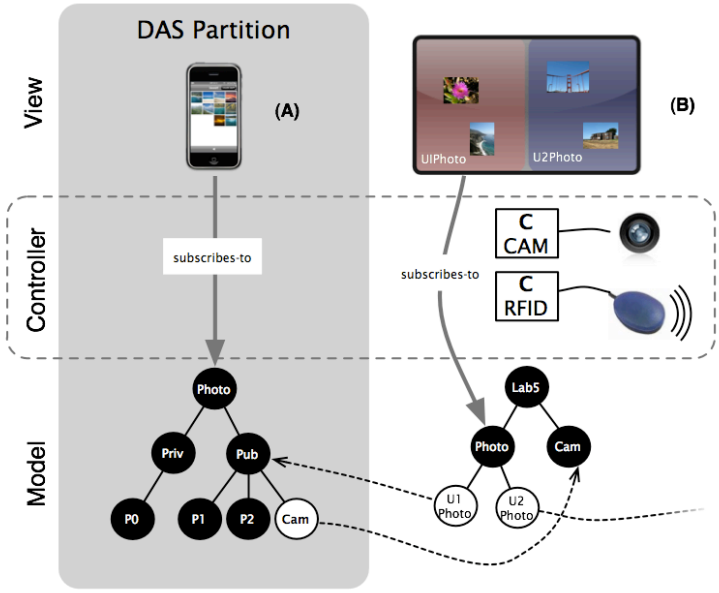


Figure 1 An example DAS. Two partitions are shown, one on a mobile device (A), and one on a stationary display (B).

independently of the rest of the DAS. A partition consists of at least one model, controller or view but may not be useful until enrolled into a more complete DAS. In this example the components of the mobile device (A) is a single application partition. The mobile device runs a photo-browser view which is connected to photos stored in a local AEXO data structure. The photo view is able to display the photos contained in the model, which is divided into a *private* (`/Photo/Priv`) and a *public*(`/Photo/Pub`) submap. In the meeting room (named *Lab5*), the public display (device B) runs a different photo-browser view, which is able to display multiple collections of photos. This view shows photo collections from a submap (`/Lab5/Photo/`) in its local model.

The controllers in this DAS are deployed in the AEXO instance running on device B. It consists of two controller components. The first one, labeled *CAM* in Figure 1, controls the public camera in the meeting room; whenever a request for the content of the node `/Lab5/Cam` is made, the controller uses the camera to take a picture and store it in the `/Lab5/Cam` node. The *RFID* controller is responsible for discovering devices coming in close proximity of the display, and to show these devices' public photos. This is done by mounting the mobile device's `/Photo/Pub` submap into the local `/Lab5/Photo` submap of device B. This change in the local model is then detected by the *CAM* controller which asks the device's model to mount the `/Lab5/Cam` node in its `/Photo/Pub` submap. Mounting a remote submap is done by inserting a proxy node into the local map, and this node then takes care of forwarding requests, routing events to and from the remote location. The `/Photo/Pub/Cam` on device A can now forward requests to device B prompting the *CAM* controller to grab an image from the camera. Thus the node in device A now

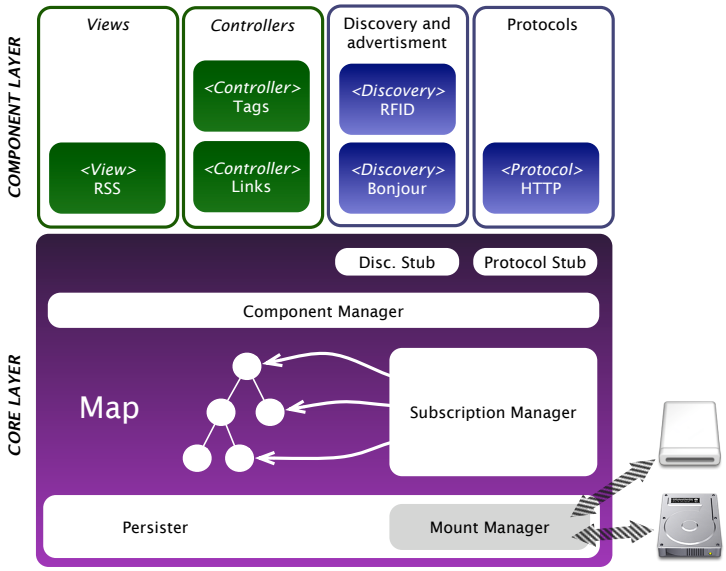


Figure 2 An overview of the AEXO core system with example components installed

represents a "dynamic" photo which is updated whenever a request for that node is made.

Note how the DAS in Figure 1 is formed dynamically as devices enter and leave the meeting room. The public photos of the devices will appear on the shared display, because the model represented in the view running on the shared display has been extended onto the mobile devices' model. Furthermore, the mobile device gains new functionality. It can now grab photos from the public camera in the room by reading from its `/Photo/Pub/Cam` node. Note also that any photos that device A is taking using the public camera will appear both on device A's displays as well as on the shared screen, because the models of the two views are partially shared.

3 The aexo Infrastructure

Figure 2 shows the overall architecture of AEXO which consists of a core layer and a component layer. The core layer contains a few basic components. A hierarchical map is the central data-structure which contains all data in a DAS. The map also hosts subscriptions and the subscription manager then provides hooks for publishing and subscribing to changes in a specific part of the map. Furthermore, the core infrastructure has built-in support for mounting external or remote submaps and provides persistent storage for the information contained in the map. The core components will be discussed further in the following sections.

Hierarchical Map

In AEXO, data is stored as nodes in a hierarchical map as illustrated in Figure 1. Each node is referenced via an URL like scheme. The map is basically a tree in which links between nodes are allowed. This structure makes it simple to store data which can be expressed as graphs, but also provides a suitable structure for event subscriptions and propagation. Subscriptions are made on a submap basis. For example, a subscription for `/Lab5/Photo` will receive notification of any changes to this node, and to all nodes in the submap below.

The API for accessing the map is intentionally kept very simple. The operations for reading from the map are `read <path>` and `access <path> <args>`. The former simply returns the data element stored at the given path, while the latter performs some extra logic to, if possible, evaluate the data element. The result of the `access` operation depends on what type of data element is stored in the node. If the element is a reference to a file, `access` might take a range as argument and only return the data in that range from the file. Modifications to the map are done with the `write <path> <data>` or the `delete <path>` operation. Subscriptions are set-up with the `subscribe <path> <callback>` method, where callback should be an URL or for local listeners; a subscriber-object.

The *Persister* provides functionality for serializing and deserializing the structure of the map and the elements contained in it. It is invoked asynchronously for every modification, ensuring that changes are always written to disk. The serialization maps directly onto the filesystem, converting the map to an equivalent folder hierarchy with leaf nodes stored as either xml or binary files. Links in the map are represented as leaf nodes with protocol, host and path information of the endpoint written to a file.

Subscription Manager

The *Subscription Manager* handles both internal and external subscriptions for nodes in the map. Internal subscriptions are those created by components running in the same address space as the AEXO infrastructure (i.e. dynamically loaded as byte code), while external subscriptions are those existing in another address space, potentially on a remote device. Subscribers to a node will be notified about changes to the node and the submap below. Event notifications denoting *modifications* of existing nodes and notifications denoting *additions* to the map bubble from the node, which was changed or created, and upwards. Notifications of *deletions* travel both upwards and downwards since a delete removes the entire submap and listeners in the submap must thus be notified that they are now disconnected and no longer part of the map.

A notification also carries a *context object* which is a description of what changes caused the notification. Initially the entity writing to the map need to supply a descriptive text for the change, which is then bundled with the actual change (the path, new and old value) into the context of the change. Subscription owners which upon receiving a notification again write to the map adds another entry to

the context. This results in increased transparency for event chains which may become arbitrarily complex and enables the infrastructure to supply human-readable explanations as to *why* any given change happened. All events are asynchronously processed but a logical order is maintained, i.e. if modification M^1 happens before change M^2 then no subscriptions will receive notification of M^2 before M^1.

Component Manager

The *Component Manager* handles loading and unloading of components and manages the resources which are allocated for each component. Components are loaded at runtime from jar files in a given directory. If a new component is copied to the directory it will automatically be loaded and instantiated by the component manager. Each component runs within a contained environment, but is given a reference to the map in which they can then read, write and install listeners.

A specific submap is reserved for each component, here the component may write their configuration data. Using the map to store this information enables other components or external applications to read and write to the configuration of the component, or use the existence of such a configuration submap as evidence that the component has been loaded. This configuration can be considered as a portable "environment" for the component and it will be preserved between component loads and unloads.

The system has specific support for two kinds of components; discovery and protocol. The discovery component is used by the map to advertise its presence and its shared submaps, and to look for and mount other shared submaps. Currently, aexo implements a Bonjour discovery component, and one which uses an external RFID reader. Protocol components allows the infrastructure to dynamically load communication protocols, which are discussed further in in the next section. All other components implement either view or controller logic specific to any one DAS. View components are components which export a specific view of the data contained in the map. These should not be confused with the views of an application space, which may be implemented in disjunct applications. A common task of controller-component is to maintain a specific perspective on one submap in another. For example, to look for location information on nodes in one submap and then use this information to maintain another, location-centric submap, i.e. where nodes represent locations and their children represent entities contained in these locations. External applications are then able to take advantage of this controller logic by subscribing to changes in strategic locations in the controller maintained submaps. Thus, it is possible to encode controller logic directly in the infrastructure, allowing transient applications to take advantage of application space specific logic which is external to the application itself.

Component Stubs

The *Protocol Stub* is where components implementing a protocol interface is registered. A protocol component is simply a special type component which provides

methods for receiving map operations, translating them to an external format and then forwarding the request and visa versa for incoming requests. Since there are only five basic types of operations on the map and one notification format it is relatively straightforward to implement new protocols. The protocol however also need to determine a format with which to transfer nodes and elements. Basic XML and binary serialization is provided by the core map but a protocol may choose to implement a custom scheme.

Another task which the protocol stub also handles is resolving of external references or mounts. Mounts are basically stored in the map as proxied nodes, which then use the protocol stub to resolve which protocol to use for forwarding. It is however not entirely transparent to the modification issuer whether the node that is requested is stored locally or remotely. If the node is stored remotely then a `RemoteNodeException` is thrown, and the issuer of the request can choose to either abandon the request, or tell the exception that the request should proceed. This forces the entity accessing the map to explicitly decide whether or not specific remote requests should proceed. The reason for not, in this case, having a completely transparent request execution is that distribution transparency is more relevant and useful in pervasive environments which by definition are highly distributed.

The *discovery stub* has similar functionality as the protocol stub, but instead of protocols it keeps track of discovery/advertisment components. Once a new such component is loaded the map will begin to advertise its presence using that component. Other components may then utilize the discovery or advertise mechanisms as well, e.g. one of our components simply uses the information to maintain a subtree with information on any AEXO instances that are discovered.

4 Application Space Programming Interface

The programming model for building application spaces in the AEXO infrastructure is based on the model-view-controller scheme. The following describes which roles models, controllers and views have in an DAS and presents a ASPI binding-scheme for .NET.

In the context of AEXO a *model* is data represented in the hierarchical map. The map itself allows arbitrary graphs of data to be represented, proving it versatile and able to accomodate a wide range of data. It does not make any restrictions on the type of data stored at each node, but is not optimized for very data intensive use. The distribution of the map, the model, can be done on a submap basis, that is, looking a the complete map for any distributed application space, then there is support for any arbitrary submap to be distributed to any given device running an instance of the infrastructure. It is even possible to chain AEXO maps, such that a request for a given node on one machine has to travel across multiple mounts on multiple machines to get to the data. Paths are resolved at runtime for every request. Exporting an otherwise internal application model to AEXO infrastructure is a way of "reversing" the structure of the application, exposing its core object model. Once in the map the model may be externally modified or shared with other

applications by having them "import" and subscribe to the model representation. AEXO, in this case, also provides a strict decoupling of applications, *any* entity can provide the information stored in the map and the lack of forced structure even makes possible for applications to mix their models. As soon as a data-structure has been bound in the map, its components, the nodes in the map can thus be made to form relationships with other nodes, representing components in other data-structures (which are possibly bound to applications running on other machines). This cross-cutting of information brings a great deal of expressiveness to distributed application spaces, and allows complex relationships to be made in the models that are shared in a given space.

The functionality of the *controller* is provided in AEXO by a combination of the controller-components loaded by the infrastructure and the application partitions of the DAS. The task of controller-components is to maintain and organize certain parts of the map based on data and events from certain other parts. That is, to define the dynamic relationships between the data stored in the map and to create coherent interfaces from otherwise disparate data-sources. The logic represented by the controller-components will then be logic that is specific for this DAS configuration, specific for the AEXO instance that runs the DAS and therefore tailored to the physical context of the system. This means that as applications enter a DAS they are automatically augmented with new controller-logic essentially *reconfiguring* them for the space in which they now execute, as is the case for the example from Figure 1. Controller-logic may also be implemented in the DAS applications themselves – this logic however of course migrates with the application and should thus constitute functionality unique to the specific application, and not to the space. The combined controller-logic available in a DAS therefore depends on both which controller-components are loaded but also which partitions are currently present.

Views are the interfaces which end-users interact with a full DAS. Mostly, they are built as stand-alone programs and linked to data and controller-logic in the infrastructure using either the .NET ASPI directly or a more basic protocol to send and receive data and events. In a fashion similar to that of controller-logic, the structure of a view can be composed partly by the structure of the submap which represents the data, partly by the actual application displaying it. AEXO can move much view-functionality out of the external application thus leveraging resource-constrained devices. But it also goes the other way; complex views, which are more application- than space-specific, may be constructed as part of the actual application. This adds a great deal of flexibility to how the views in a pervasive application spaces are constructed, represented and most importantly distributed. Furthermore, views can be kept completely in the infrastructure by having components which are tied to a specific AEXO instance implement them – as is the case of the RSS component described in 5.

AEXO thus provides (i) a expressive distributed model, (ii) a platform supporting both space- and application specific distributed controllers and (iii) data-based views for low-fidelity devices while still allowing developers to implement complex views on complex devices.

```
1  // DisplayPhotoModel is the ASPI for the public display model
2  public class DisplayPhotoModel {
3     [Reflective("local://Lab5/Photo/U{list._counter}Photo",
4                 Detection=Automatic,
5                 Binding=OnEnable,
6                 Recursion=AllAttributed,
7                 Synchronization=TwoWay})]
8     public List<PhotoCollection> Photos
9     {
10        get; set;
11    }
12 }
```

Listing 1 Property annotations

Application Composition

Applications can access an instance of AEXO directly using any of the loaded protocols. An easily accessible choice is the component implementing the HTTP protocol. Here the map operations are mapped onto HTTP methods, a `read` becomes a `GET`, `write` a `POST` and so on. In order for an application to interact with the map the application then needs to read, write to the appropriate paths as it would do to any other HTTP server. To receive notifications, the application needs to run a HTTP server as well. There are many choices for embedded servers for a variety of languages. We have built wrappers for this scheme in Java and in Objective-C for the iPhone.

Another option for .NET applications is to use our ASPI binding-scheme, which enables objects and their properties to be mapped directly onto the infrastructure, essentially exporting or importing model data. The hierarchical nature of the core data-structure, the map, lends itself easily to this scheme, since runtime objects and their properties form graphs which can be represented directly in the map. This mapping or binding can then be kept synchronized to keep both the map and the application updated, or to perform one-way synchronization only. The binding can act as a communication channel as well, by allowing other applications to change the live-object property by writing to the corresponding node in the map.

To facilitate this direct binding between live-objects and the distributed map we have built support for annotating .NET properties with AEXO specific binding and synchronization attributes.The syntax used for property-binding can be seen in the example given in Listing 1.

The `Reflective` annotation class lets developers specify a number of properties on the binding, the example above has values for all options but only the path, the first argument, is required. We allow certain dynamic components to be included in the path (as seen in line 3), e.g. the path `{bonjour._http._aexo_}/Lab5` will be resolved to a the `/Lab5` path in the map which advertises it presence using Bonjour under the `_http._aexo` service type, e.g. `http://1.2.3.4/Lab5`. Similarly,

the library provides options for expanding paths, which is useful for instance for recursive data-types or list-based data-types. The `{list._counter}` expression in Listing 1 is an example of this. It is used to map multiple nodes matching the path to a local list. When the `Photos` property is bound to a map, the nodes in the map which match the path are bound to elements in a local list, e.g. `Lab5/Photo/U1Photo` is bound to the 2nd element in the list, and as new nodes appear in the map, the list is expanded with new elements.

The optional *detection*-property (line 4) decides when local changes are written to the infrastructure. By setting the detection to *manual* for instance, the application may delay the write to a later time. This can be used to e.g. bundle updates to minimize connection time.

The *binding*-property (line 5) determines when binding takes place. The options are to let the developer manually bind the object or to continuously try to bind it. The latter will try to keep the object bound to the infrastructure, and could be used to e.g. host objects on a mobile device. When the device moves between networks the object will be bound to the different AEXO instances that are available, thus broadcasting the presence of the (application on the) device.

The *recursion* property (line 6) denotes if and how the local object-graph (starting with the current object and following all reference type objects) should be traversed and bound onto the infrastructure. Currently, there are three options. `AllAttributed` indicates that all public properties which are annotated with the `Reflective` attribute are automatically bound. The `AllPublic` options includes all public properties. `None` only maps the current object, ie. does not traverse the object-graph.

There are three modes of *synchronization* (line 7). First, `TwoWay` keeps the map and local object in total synchronization. It is the state in the infrastructure which is considered the master and and conflicts are resolved by using this state. By using `AppToMap` the infrastructure state is considered read-only and any changes made in it will not affect the local model. The reverse is the case with the `MapToApp` option.

5 Example Applications

The goal of AEXO is to provide an infrastructure and a programming model which enables the design and implementation of distributed application spaces. Focus has hence been on providing a platform which is highly extensible. A valid and often used method for evaluating to what degree an infrastructure fulfills such non-functional goal is to use the infrastructure to build as small set of non-similar applications [65].

This section presents two distributed application spaces, which has been built on top of AEXO. These examples are both non-trivial and represents real-world applications for real users. As such they demonstrate the utility of AEXO for building distributed application spaces.

InSpace Conference Room

The InSpace conference room system [178] was designed to support interaction around a conference table with multiple users interacting with a shared repository of data. The conference table itself was equipped with RFID sensors at each seating position which were used for detecting the RFID-tagged laptops. The room itself was equipped with a projector and a large LCD display. The participants seated at the table would have an application partition installed on their laptops which enabled them to interact with the combined application space of the room, i.e. the laptops and the machines running the two display devices. The DAS therefore consisted of an AEXO instance, a broker for event-routing, the transient applications running on participants laptops and the fixed views, running in separate application partitions, for presenting data on the shared displays. The AEXO infrastructure was used for storage, distribution and organization of the shared data.

Figure 3 *Left:* Laptop is placed on the table and detected by the system. *Right:* The conference room table and shared display.

The typical data shared during meetings were documents, text-, pdf- or MS Word files and presentations. All files were stored in a submap specific to each meeting, but this was never accessed directly by any of the participating applications. Instead, a controller-component would organize another submap based on the "tags" applied to the shared data, and furthermore, would add automatic tags to all shared items. These automatic tags were based on context information, for instance one tag applied was the title of the current meeting which the controller extracted from a calendaring system, while others represented the people present in the room. Other tags were manually applied using the application running on each participants laptop. The resulting tag-centric submap maintained by the controller was then used by the display devices. Items tagged with e.g. "projector" was automatically retrieved by the machine attached to the projector and shown in a grid of available media on the shared display. Likewise, the laptop applications could set-up subscriptions for a given tag, e.g. "meeting x" to have a local copy of the submap extracted from AEXO and stored on their local device.

Another component implemented for the conference scenario was the RSS view. This component provides an RSS feed for all changes made to any given submap which are exported in a human-readable format when a participant access

the submap representing the feed. This enabled participants to get a history of changes for e.g. the submap representing the "meeting x" by entering the URL `http://aexo/RSS/tags/meeting_x!` in their feed-reader. The postfixed `!` gets the http `GET` request translated in the HTTP component to an `access` request which then makes the RSS-feed data element format itself as an rss feed which is returned for the viewer to parse.

The application space of the conference room backed by an AEXO instance had thus both a shared distributed model, consisting of the data shared during a conference session, as well as an infrastructure-based controller-layer which was responsible for "tag"-based organization and distribution of files and events. The controller-logic was specifically tailored for the conference room. The DAS here had views which were distributed on multiple machines, the presentation view executed on the machines attached to the projector and the LCD display and the views for adding content and controlling the presentations ran on participants laptops.

The InSpace system was implemented in Java therefore we could not use the ASPI to bind the views to the DAS. Instead we used the HTTP protocol mediated through a custom built broker akin to the Java Messaging Service. The effort in lines-of-code needed to build the laptop application partitions were 1,848 while the shared display partitions were 2,516. The effort needed to send and receive messages and events were 128 LOC for the laptop partitions and 153 LOC for the shared display, and a further 520 LOC were needed for a protocol component to allow AEXO to support the broker scheme.

Clinical Context-aware Activity-Browser

Activity-based computing is an approach to pervasive computing which seeks to support and model human activities as computational entities. A computational activity models a real-world human activity, and encompass relevant data, resources, and participants. An activity is stored and managed by an infrastructure. It can be resumed and suspended, move between different devices, be shared amongst several participants, and adapted to the context in which it is used. The goal of activity-based computing is to support work characterized by frequent interruptions and a high degree of mobility – work such as that you would find in e.g. hospitals.

The *Clinical Context-aware Activity-Browser* (CCAB) system is a DAS designed to support activities in hospitals. It consists of a number of public displays, some fixed larger displays and some tablet-sized mobile devices, all running the CCAB client. These clients provide context-adapted *views* of the activities modeled in the DAS, and controls for manipulating and browsing content of the activities. A typical activity could for instance be the treatment of a patient. The applications and data contained in it would then be e.g. the electronic patient record, lab test results, and radiology images. Each activity also contains meta-information such as the participants, i.e. the personnel assigned to the patient, and information about the physical location of the activity.

Figure 4 The deployment of the CCAB distributed application inside the OR showing a list of relevant activities for this OR and associated medical data.

CCAB was built on top of AEXO, which provides support for storing and distributing activities and the data contained in them by mapping the hierarchical structure of an activity onto the map. The infrastructure keeps track of contextual information such as the location of personnel and mobile devices, and organize and merge this information into the activity model. Location tracking of personnel and devices is done using active RFID. In AEXO a set of controller components map tracking information from the RFID readers to the corresponding nodes in the map. CCAB clients listen to changes in this location sub-map and are continuously updated to display information from the submap which match their current location. For instance, if a doctor is detected to be using CCAB client A, the controller-components would makes sure that the submap representing CCAB client A is updated to include the activities of this doctor. This has the effect that the mobile devices are continuously showing the relevant activities for their current location, and, in the case of a mobile device, the personnel carrying the device.

In essence, the hospital is modelled as one large DAS in which the CCAB clients represent the distributed views, and the components provide controller logic to distribute events and data based on the context of these views. Thus, the model-view-controller scheme modelled in the CCAB system is fully distributed across the entire DAS and is dynamically updated as devices and personnel physically moved around inside the hospital. CCAB was implemented on the .NET platform using the ASPI binding scheme, the total lines-of-code was 5,832 of which 427 were dedicated to the shared model. Only 26 model properties were annotated in the complete model, they constituted the complete effort to enable the CCAB browser to interact with the AEXO infrastructure. This constitutes a significant reduction in effort measured in lines-of-code compared to the InSpace Conference Room.

The CCAB system including the AEXO platform was deployed and tested at a large hospital by a team of doctors and nurses in realistic situations. Figure 4 shows the deployment of the system on a large public display inside an operating room.

6 Related Work

The notion in the Gaia project of an active space [149, 46] comes close to our definition of a distributed application space. Active spaces are virtual locations that host the execution of applications, which themselves may be composed of distributed components. The approach taken in AEXO is more lightweight, enabling distributed applications or views to "export" model and controller functionality to the space, and does not require the middleware to actually execute on the hosts running the views. Aura [75] is a middleware for pervasive computing focusing on resource adaption and provides support task-based computing where applications can move between heterogenous devices. Applications are hence not distributed as such, but are mobile whereas AEXO enables the formation of distributed applications spaces on a much finer granularity. The *one.world* architecture [81] enables adaptable applications by exposing contextual change to applications, to encourage ad hoc composition of services, and to facilitate sharing between devices and applications. The main differences between AEXO and these infrastructures lies, however, in the approach to application integration. Gaia requires services to conform to specific application-types (context-services, presence-services etc) while the informal data structure of AEXO encourages a more dynamic approach. The architecture of AEXO is different in that protocols, discovery and all but the very basic functionality is dynamically added to the core system and not integrated by default, a structure which enables a greater degree of flexibility. Moreover, AEXO enables the formation of distributed applications spaces on a much finer granularity by enabling the application's components to be partitioned across several devices while working in concert. The BASE middleware [28] shares the idea of a uniform protocol independent sharing of data between devices with AEXO. BASE provides a system which may run on a wide range of heterogenous devices and has low-level support for protocol adaptation. It differs from AEXO in its goals, which are to provide a foundation for pervasive applications and component systems and decoupling the communication model from the application logic, while AEXO is built with a higher level of abstraction in mind – and specifically to support application component distribution. Satoh [153] describes an agent-based system for building adaptive distributed applications. Agents which incorporate the components of an application can be dynamically composed to form new functionality. Similarly the PCOM [29] framework exports a toolchain to build dynamic and customizable distributed applications from singular components. Both these systems allow great flexibility in application composition, but do not deal with data distribution, cross-application events, or truly distributed applications. AEXO supports application composition in fully distributed applications, allowing applications executing on multiple devices to share state.

7 Conclusion

In this paper we formulated the concept of a distributed application space. A concept that spans pervasive environments and allows the individual components in

applications to be distributed across the entire space. We described a novel infrastructure implementation called AEXO, which enables these spaces and which orchestrates the distribution of application components and interaction between application partitions in a DAS. Furthermore, we have shown by two proof-of-concept deployments that (i) the concept of a DAS simplifies interconnection by providing application partitions with a shared infrastructure into which functionality and data can be made public and shared. That (ii) the DAS programming model for .NET provides increased expressive leverage, letting developers integrate their application partitions more easily by using the property annotation scheme. This scheme was shown, by example, to reduce the effort measured in lines-of-code substantially compared to using a standard message-based system.

Acknowledgements

The aexo part of the InSpace conference room system was made while the first author was visiting Keith Edwards and his pixi lab at Georgia Institute of Technology. The implementation of the complete system involved a great deal of people and our part was only a small piece of this puzzle. We wish to thank all those people. Similarly, we wish to thank Afsaneh Doryoab and Steffen Sørensen for their work on the CCAB project.

Publications

[1] Stinne A. Ballegaard, Jonathan B. Pedersen, and Jakob E. Bardram. Where to, roberta?: reflecting on the role of technology in assisted living. In *NordiCHI '06: Proceedings of the 4th Nordic conference on Human-computer interaction*, pages 373–376, New York, NY, USA, 2006. ACM.

[2] Jakob E. Bardram and Jonathan Bunde-Pedersen. Iaso – an activity-based computing platform for wearable computing. In *ICDCSW'05: Proceedings of the Fifth International Workshop on Smart Appliances and Wearable Computing (IWSAWC) (ICDCSW'05)*, pages 484–490, Washington, DC, USA, 2005. IEEE Computer Society.

[3] Jakob E. Bardram, Jonathan Bunde-Pedersen, and Martin Mogensen. Differentiating between accountable and ephemeral events in the abc hybrid architecture for activity-based collaboration. In *Proceedings of the IEEE International Conference on Collaborative Computing (CollaborateCom 2005)*, pages 168–176. IEEE Press, 2005.

[4] Jakob E. Bardram, Jonathan Bunde-Pedersen, and Mads Soegaard. Support for activity-based computing in a personal computing operating system. In *CHI'06: Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, USA, 2006. ACM Press.

[5] Jonathan Bunde-Pedersen and Jakob E. Bardram. Towards an activity-based world-wide-web. In *IIWeb'06: Proceedings of The 3rd IIWeb Interdisciplinary Workshop for Information Integration on the Web*, Edinburgh, Scotland, 2006. ACM Press.

[6] Jonathan Bunde-Pedersen, Martin Mogensen, and Jakob E. Bardram. The abc adaptive fusion architecture. In *MPAC '06: Proceedings of the 4th international -workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC 2006)*, New York, NY, USA, 2006. ACM Press.

[7] Mikkel B. Kjærgaard and Jonathan Bunde-Pedersen. A formal model for context-awareness. Technical Report RS-06-2, 2006.

[8] Mikkel B. Kjærgaard and Jonathan Bunde-Pedersen. Towards a formal model of context awareness. In *First International Workshop on Combining Theory and Systems Building in Pervasive Computing 2006 (CTSB 2006)*, 2006.

[9] S. Voida, M. Mckeon, C. Le Dantec, C. Forslund, P. Verma, B. Mcmillan, J. Bunde-Pedersen, K. Edwards, E. Mynatt, and A. Mazalek. inspace: Co-designing the physical and digital environment to support workplace collaboration. Technical Report GIT-GVU-08-03, 2007. 59, 182

Bibliography

[1] Computer Supported Cooperative Work in Clinical Practice. In J. Bender, J. P. Christensen, J. R. Scherrer, and P. McNair, editors, *Proceedings of the 13th International Congress of the European Federation for Medical Informatics*, pages 853–857. IOS Presss, 1995. 159

[2] G. D. Abowd, A. K. Dey, R. Orr, and J. Brotherton. Context-awareness in wearable and ubiquitous computing. In *Proceedings of the 1st IEEE International Symposium on Wearable Computers*, page 179. IEEE Computer Society, 1997. ISBN 0-8186-8192-6. 103

[3] P. D. Adamczyk and B. P. Bailey. If not now, when?: the effects of interruption at different moments within task execution. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 271–278. ACM Press, 2004. ISBN 1-58113-702-8. doi: http://doi.acm.org/10.1145/985692.985727. 84

[4] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. In L. Cardelli and L. Cardelli, editors, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 74–102. Springer, 2003. URL http://dblp.uni-trier.de/rec/bibtex/conf/ecoop/AldrichSCN03. 18

[5] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, July 1997. doi: http://dx.doi.org/10.1145/258077.258078. URL http://dx.doi.org/10.1145/258077.258078. 18

[6] J. P. Almeida, M. van Sinderen, D. A. C. Quartel, and L. F. Pires. Designing interaction systems for distributed applications. *Distributed Systems Online, IEEE*, 6(3), 2005. doi: http://dx.doi.org/10.1109/MDSO.2005.13. URL http://dx.doi.org/10.1109/MDSO.2005.13. 18

[7] K. M. Anderson, R. N. Taylor, and Jr. Chimera: hypertext for heterogeneous software environments. In *ECHT '94: Proceedings of the 1994 ACM European conference on Hypermedia technology*, pages 94–107, New York, NY, USA, 1994. ACM. ISBN 0-89791-640-9. doi: http://dx.doi.org/10.1145/192757.192783. URL http://dx.doi.org/10.1145/192757.192783. 131

[8] Apple. Networking bonjour. . URL http://developer.apple.com/networking/bonjour/. 57

[9] Apple. Apple - mac os x leopard - features - time machine, . URL http://www.apple.com/macosx/features/timemachine.html. 40

[10] J. Austin Henderson and S. Card. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Transactions on Graphics (TOG)*, 5(3):211–243, 1986. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/24054.24056. 37, 84, 99

[11] L. Bannon. From human factors to human actors: the role of psychology and human-computer interaction studies in system design. pages 25–44, 1992. URL http://portal.acm.org/citation.cfm?id=125458. 13

[12] L. Bannon, A. Cypher, S. Greenspan, and M. L. Monty. Evaluation and analysis of users' activity organization. In *CHI '83: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 54–57. ACM Press, 1983. ISBN 0-89791-121-0. 37, 83, 97, 99

[13] L. J. Bannon and S. Bødker. Beyond the interface: Encountering artifacts in use. *Designing Interaction: Psychology at the human-computer interface*, 1991. URL http://www.ul.ie/~idc/library/papersreports/LiamBannon/13/LBsb9.html. 13

[14] E. Bardram. Activity-based computing: support for mobility and collaboration in ubiquitous computing. *Personal Ubiquitous Comput.*, 9(5):312–322, 2005. ISSN 1617-4909. doi: http://dx.doi.org/10.1007/s00779-004-0335-2. URL http://dx.doi.org/10.1007/s00779-004-0335-2. 116, 119

[15] J. E. Bardram. From Desktop Task Management to Ubiquitous Activity-Based Computing. In V. Kaptelinin and M. Czerwinski, editors, *Integrated Digital Work Environments: Beyond the Desktop Metaphor*. MIT Press, 2005. To appear. 131, 148

[16] J. E. Bardram. The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications. In H. Gellersen, R. Want, and A. Schmidt, editors, *Proceedings of the 3rd International Conference on Pervasive Computing (Pervasive 2005)*, volume 3468 of *Lecture Notes in Computer Science*, pages 98–115, Munich, Germany, May 2005. Springer Verlag. 172

[17] J. E. Bardram. Activity-Based Computing: Support for Mobility and Collaboration in Ubiquitous Computing. *Personal and Ubiquitous Computing*, 9 (5):312–322, July 2005. ISSN 1617-4909. URL http://dx.doi.org/10.1007/s00779-004-0335-2. 4, 15, 67, 74, 84, 86, 88, 93, 129, 142, 148

[18] J. E. Bardram. Activity-Based Computing – Principles, Implementation, and Evaluation. Manuscript submitted to the 'ACM Transactions on Computer-Human Interaction (ToCHI)'. Submission date: April 2004. 94, 131

[19] J. E. Bardram. Activity-Based Support for Mobility and Collaboration in Ubiquitous Computing. In L. Baresi, editor, *Proceedings of the Second International Conference on Ubiquitous Mobile Information and Collaboration Systems*

(UMICS 2004), Lecture Notes in Computer Science, pages 169–184, Riga, Latvia, Sept. 2004. Springer Verlag. 102, 105, 108, 159

[20] J. E. Bardram and C. Bossen. Mobility Work – The Spatial Dimension of Collaboration at a Hospital. *Computer Supported Cooperative Work.*, 14 (2):131–160, 2005. ISSN 0925-9724. URL http://dx.doi.org/10.1007/s10606-005-0989-y. 84, 86

[21] J. E. Bardram and H. B. Christensen. Supporting pervasive collaboration in healthcare — an activity driven computing infrastructure. URL http://www.pervasive.dk/publications/files/abc_infrastructure_bardram.pdf. 74

[22] J. E. Bardram and H. B. Christensen. Pervasive computing support for hospitals: An overview of the activity-based computing project. *Pervasive Computing, IEEE*, 6(1):44–51, 2007. doi: http://dx.doi.org/10.1109/MPRV.2007.19. URL http://dx.doi.org/10.1109/MPRV.2007.19. 114, 116, 119, 172

[23] J. Barton and T. Kindberg. The cooltown user experience. *HP Labs Technical Report (HPL-2001-22)*, 2001. URL http://www.hpl.hp.com/techreports/2001/HPL-2001-22.html. 62

[24] P. Baudisch, N. Good, and P. Stewart. Focus plus context screens: combining display technology with visualization techniques. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 31–40. ACM Press, 2001. ISBN 1-58113-438-X. doi: http://doi.acm.org/10.1145/502348.502354. 5, 85

[25] P. Baudisch, N. Good, V. Bellotti, and P. Schraedley. Keeping things in context: a comparative evaluation of focus plus context screens, overviews, and zooming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 259–266. ACM Press, 2002. ISBN 1-58113-453-3. doi: http://doi.acm.org/10.1145/503376.503423. 104

[26] M. Bauer, T. Heiber, G. Kortuem, and Z. Segall. A collaborative wearable system with remote sensing. In *Proceedings of the 2nd IEEE International Symposium on Wearable Computers*, page 10. IEEE Computer Society, 1998. ISBN 0-8186-9074-7. 29, 102, 104

[27] M. Beaudouin-Lafon, editor. *Computer Supported Cooperative Work*. John Wiley and Sons, New York, 1999. 195, 197, 203

[28] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. Base " a micro-broker-based middleware for pervasive computing. In *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0769518931. URL http://portal.acm.org/citation.cfm?id=826336. 185

[29] C. Becker, M. Handte, G. Schiele, and K. Rothermel. Pcom - a component system for pervasive computing. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*

(PerCom'04), Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0769520901. URL http://portal.acm.org/citation.cfm?id=978688. 74, 185

[30] V. Bellotti, N. Ducheneaut, M. Howard, and I. Smith. Taking email to task: the design and evaluation of a task management centered email tool. In *CHI '03: Proceedings of the conference on Human factors in computing systems*, pages 345–352. ACM Press, 2003. ISBN 1581136307. doi: http://dx.doi.org/10.1145/642611.642672. URL http://dx.doi.org/10.1145/642611.642672. 5

[31] V. Bellotti, N. Ducheneaut, M. Howard, and I. Smith. Taking email to task: the design and evaluation of a task management centered email tool. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 345–352. ACM Press, 2003. ISBN 1-58113-630-7. doi: http://doi.acm.org/10.1145/642611.642672. 85

[32] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody. Role-based access control for publish/subscribe middleware architectures. In *DEBS '03: Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8, New York, NY, USA, 2003. ACM Press. ISBN 1581138431. doi: http://dx.doi.org/10.1145/966618.966622. URL http://dx.doi.org/10.1145/966618.966622. 62

[33] M. S. Bernstein, J. Shrager, and T. Winograd. Taskposé: exploring fluid boundaries in an associative window visualization. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 231–234, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-975-3. doi: http://dx.doi.org/10.1145/1449715.1449753. URL http://dx.doi.org/10.1145/1449715.1449753. 40

[34] M. Bieber and F. Vitali. Toward support for hypermedia on the world wide web. *Computer*, 30(1):62–70, 1997. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=562928. 131

[35] J. T. Biehl and B. P. Bailey. Aris: an interface for application relocation in an interactive space. In *GI '04: Proceedings of Graphics Interface 2004*, pages 107–116, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society. doi: http://dx.doi.org/10.1145/642611.642666. URL http://dx.doi.org/10.1145/642611.642666. 115

[36] J. T. Biehl, W. T. Baker, B. P. Bailey, D. S. Tan, K. M. Inkpen, and M. Czerwinski. Impromptu: a new interaction framework for supporting collaboration in multiple display environments and its field evaluation for co-located software development. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 939–948, New York, NY, USA, 2008. ACM. ISBN 9781605580111. doi: http://dx.doi.org/10.1145/1357054.1357200. URL http://dx.doi.org/10.1145/1357054.1357200. 32, 39, 114, 115, 172

[37] M. Billinghurst, J. Bowskill, M. Jessop, and J. Morphett. A wearable spatial conferencing space. In *Proceedings of the 2nd IEEE International Symposium on Wearable Computers*, page 76. IEEE Computer Society, 1998. ISBN 0-8186-9074-7. 29, 102, 104

[38] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/7351.7478. 151

[39] J. A. Black, J. I. Hong, M. W. Newman, W. K. Edwards, S. Izadi, J. Z. Sedivy, and T. F. Smith. Speakeasy: A platform for interactive public displays. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.61.1579. 75

[40] N. O. Bouvin. Unifying strategies for web augmentation. In *HYPERTEXT '99: Proceedings of the tenth ACM Conference on Hypertext and hypermedia : returning to our diverse roots*, pages 91–100, New York, NY, USA, 1999. ACM Press. ISBN 1581130643. doi: http://dx.doi.org/10.1145/294469.294493. URL http://dx.doi.org/10.1145/294469.294493. 131

[41] S. K. Card and A. Henderson. A multiple, virtual-workspace interface to support user task switching. In *CHI '87: Proceedings of the SIGCHI/GI conference on Human factors in computing systems and graphics interface*, pages 53–59, New York, NY, USA, 1987. ACM Press. doi: http://dx.doi.org/10.1145/29933.30860. URL http://dx.doi.org/10.1145/29933.30860. 39

[42] L. Cardelli and A. D. Gordon. Mobile ambients. In *FoSSaCS '98: Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, pages 140–155, London, UK, 1998. Springer-Verlag. ISBN 3540643001. URL http://portal.acm.org/citation.cfm?id=759638. 73

[43] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19:332–383, 2001. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.8411. 62

[44] A. Ceglar and P. Calder. A new approach to collaborative frameworks using shared objects. In *ACSC '01: Proceedings of the 24th Australasian conference on Computer science*, pages 3–10, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-0963-0. 45, 144, 145

[45] B. Chandramouli, J. M. Phillips, J. Yang, and J. Yang. Value-based notification conditions in large-scale publish/subscribe systems? In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 878–889. VLDB Endowment, 2007. ISBN 9781595936493. URL http://portal.acm.org/citation.cfm?id=1325851.1325950. 62

[46] S. Chetan, J. Al-Muhtadi, R. Campbell, and M. D. Mickunas. Mobile gaia: a middleware for ad-hoc pervasive computing. In *Consumer Communications and Networking Conference, 2005. CCNC. 2005 Second IEEE*, pages 223–228, 2005. doi: http://dx.doi.org/10.1109/CCNC.2005.1405173. URL http://dx.doi.org/10.1109/CCNC.2005.1405173. 5, 60, 185

[47] H. B. Christensen. Using Logic Programming to Detect Activities in Pervasive Healthcare. In *International Conference on Logic Programming, ICLP 2002*, Copenhagen, Denmark, Aug. 2002. Springer Verlag. 86, 88

[48] H. B. Christensen and J. E. Bardram. Supporting Human Activities – Exploring Activity-Centered Computing. In *Proceedings of Ubicomp 2002*, pages 107–116. Springer Verlag, 2002. 86, 102, 131, 148, 159, 160

[49] G. Chung and P. Dewan. Towards dynamic collaboration architectures. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 1–10. ACM Press, 2004. ISBN 1-58113-810-5. doi: http://doi.acm.org/10.1145/1031607.1031609. 44, 50, 142, 143, 152, 156

[50] B. Congleton, M. S. Ackerman, and M. W. Newman. The prod framework for proactive displays. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 221–230, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-975-3. doi: http://dx.doi.org/10.1145/1449715.1449752. URL http://dx.doi.org/10.1145/1449715.1449752. 40

[51] G. Convertino, D. C. Neale, L. Hobby, J. M. Carroll, and M. B. Rosson. A laboratory method for studying activity awareness. In *NordiCHI '04: Proceedings of the third Nordic conference on Human-computer interaction*, pages 313–322. ACM Press, 2004. ISBN 1-58113-857-1. doi: http://doi.acm.org/10.1145/1028014.1028063. 95, 96

[52] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001. doi: http://dx.doi.org/http://dx.doi.org/10.1109/32.950318. URL http://dx.doi.org/http://dx.doi.org/10.1109/32.950318. 62

[53] M. Czerwinski, E. Horvitz, and S. Wilhite. A diary study of task switching and interruptions. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 175–182. ACM Press, 2004. ISBN 1-58113-702-8. doi: http://doi.acm.org/10.1145/985692.985715. 84, 98

[54] Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):318–340, 1989. 95, 96, 122

[55] A. M. Davis, H. Bersoff, and E. R. Comer. A strategy for comparing alternative software development life cycle models. *IEEE Trans. Softw. Eng.*, 14 (10):1453–1461, 1988. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.6190. URL http://dx.doi.org/10.1109/32.6190. 9

[56] Delicious. Delicious. URL http://delicious.com/. 25

[57] R. DeVaul, M. Sung, J. Gips, and A. S. Pentland. Mithril 2003: Applications and architecture. In *Proceedings of the 7th IEEE International Symposium on Wearable Computers*, page 4. IEEE Computer Society, 2003. ISBN 0-7695-2034-0. 29, 102

[58] P. Dewan. Architectures for collaborative applications. In Beaudouin-Lafon [27], pages 169–194. 44, 46, 143, 144, 146

[59] P. Dourish, W. K. Edwards, A. Lamarca, and M. Salisbury. Using properties for uniform interaction in the presto document system. In *UIST '99: Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 55–64, New York, NY, USA, 1999. ACM Press. ISBN 1581130759. doi: http://dx.doi.org/10.1145/320719.322583. URL http://dx.doi.org/10.1145/320719.322583. 61, 119

[60] P. Dourish, W. K. Edwards, J. Howell, A. Lamarca, J. Lamping, K. Petersen, M. Salisbury, D. Terry, and J. Thornton. A programming model for active documents. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 41–50, New York, NY, USA, 2000. ACM Press. ISBN 1581132123. doi: http://dx.doi.org/10.1145/354401.354410. URL http://dx.doi.org/10.1145/354401.354410. 61

[61] A. N. Dragunov, T. G. Dietterich, K. Johnsrude, M. Mclaughlin, L. Li, and J. L. Herlocker. Tasktracer: a desktop environment to support multi-tasking knowledge workers. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 75–82, New York, NY, USA, 2005. ACM Press. ISBN 1581138946. doi: http://dx.doi.org/10.1145/1040830.1040855. URL http://dx.doi.org/10.1145/1040830.1040855. 39

[62] A. N. Dragunov, T. G. Dietterich, K. Johnsrude, M. McLaughlin, L. Li, and J. L. Herlocker. Tasktracer: a desktop environment to support multi-tasking knowledge workers. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 75–82. ACM Press, 2005. ISBN 1-58113-894-6. doi: http://doi.acm.org/10.1145/1040830.1040855. 86

[63] W. K. Edwards, M. W. Newman, J. Sedivy, T. Smith, and S. Izadi. Challenge: recombinant computing and the speakeasy approach. In *MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 279–286, New York, NY, USA, 2002. ACM. ISBN 1-58113-486-X. doi: http://dx.doi.org/10.1145/570645.570680. URL http://dx.doi.org/10.1145/570645.570680. 74

[64] W. K. Edwards, M. W. Newman, J. Z. Sedivy, T. F. Smith, D. Balfanz, D. K. Smetters, H. C. Wong, and S. Izadi. Using speakeasy for ad hoc peer-to-peer collaboration. In *CSCW '02: Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 256–265. ACM Press, 2002. ISBN 1-58113-560-2. doi: http://doi.acm.org/10.1145/587078.587114. 45, 74, 145, 156

[65] W. K. Edwards, V. Bellotti, A. K. Dey, and M. W. Newman. The challenges of user-centered design and evaluation for infrastructure. In *CHI '03: Proceedings of the conference on Human factors in computing systems*, pages 297–304. ACM Press, 2003. ISBN 1581136307. doi: http://dx.doi.org/10.1145/642611.642664. URL http://dx.doi.org/10.1145/642611.642664. 181

[66] Y. Engeström. *Learning by expanding: An activity-theoretical approach to developmental research*. Orienta-Konsultit Oy, 1987. ISBN 9519593322. URL http://www.amazon.co.uk/exec/obidos/ASIN/9519593322/citeulike00-21. 13, 14

[67] P. Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29(1), January 2007. ISSN 0164-0925. doi: http://dx.doi.org/10.1145/1180475.1180481. URL http://dx.doi.org/10.1145/1180475.1180481. 62

[68] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003. 150

[69] S. Feiner and A. Shamash. Hybrid user interfaces: breeding virtually bigger interfaces for physically smaller computers. In *Proceedings of the 4th annual ACM symposium on User interface software and technology*, pages 9–17. ACM Press, 1991. ISBN 0-89791-451-1. doi: http://doi.acm.org/10.1145/120782.120783. 104

[70] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. URL http://portal.acm.org/citation.cfm?id=932295. 55

[71] S. Finger, M. Terk, E. Subrahamanian, C. Kasabach, F. Prinz, D. P. Siewiorek, A. Smailagic, J. Stivoric, and L. Weiss. Rapid design and manufacture of wearable computers. *Communications of the ACM*, 39(2):63–70, 1996. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/230798.230806. 29, 102

[72] C. Floyd. A systematic look at prototyping. *Approaches to Prototyping*, pages 1–18, 1984. 9

[73] A. Fox, B. Johanson, P. Hanrahan, and T. Winograd. Integrating information appliances into an interactive workspace. *IEEE Computer Graphics and Applications*, 20(3):54–65, 2000. ISSN 0272-1716. doi: http://dx.doi.org/10.1109/38.844373. URL http://dx.doi.org/10.1109/38.844373. 61

[74] E. Freeman and D. Gelernter. Lifestreams: a storage model for personal data. *SIGMOD Rec.*, 25(1):80–86, March 1996. ISSN 0163-5808. doi: http://dx.doi.org/10.1145/381854.381893. URL http://dx.doi.org/10.1145/381854.381893. 40

[75] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste. Project aura: toward distraction-free pervasive computing. *Pervasive Computing, IEEE*, 1(2):22–31, 2002. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1012334. 5, 60, 172, 185

[76] W. Geyer, J. Vogel, L.-T. Cheng, and M. Muller. Supporting activity-centric collaboration through peer-to-peer shared objects. In *GROUP '03: Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, pages 115–124. ACM Press, 2003. ISBN 1-58113-693-5. doi: http://doi.acm.org/10.1145/958160.958179. 145

[77] V. M. Gonzalez and G. Mark. "constant, constant, multi-tasking craziness": managing multiple working spheres. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 113–120. ACM Press, 2004. ISBN 1-58113-702-8. doi: http://doi.acm.org/10.1145/985692.985707. 84

[78] J. D. Gould and C. Lewis. Designing for usability: key principles and what designers think. *Commun. ACM*, 28(3):300–311, March 1985. ISSN 0001-0782. doi: http://dx.doi.org/10.1145/3166.3170. URL http://dx.doi.org/10.1145/3166.3170. 9

[79] S. Greenberg and M. Roseman. Groupware toolkits for synchronous work. In Beaudouin-Lafon [27], pages 135–168. 47, 142, 144, 147, 156, 163

[80] S. Greenberg and M. Rounding. The notification collage: posting information to public and personal displays. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 514–521, New York, NY, USA, 2001. ACM Press. ISBN 1581133278. doi: http://dx.doi.org/10.1145/365024.365339. URL http://dx.doi.org/10.1145/365024.365339. 115

[81] R. Grimm. One.world: Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3):22–30, July 2004. ISSN 1536-1268. doi: http://dx.doi.org/10.1109/MPRV.2004.1321024. URL http://dx.doi.org/10.1109/MPRV.2004.1321024. 6, 74, 185

[82] C. Gutwin, S. Greenberg, and M. Roseman. Workspace awareness support with radar views. In *CHI '96: Conference companion on Human factors in computing systems*, pages 210–211, New York, NY, USA, 1996. ACM Press. ISBN 0897918320. doi: http://dx.doi.org/10.1145/257089.257286. URL http://dx.doi.org/10.1145/257089.257286. 27

[83] D. Hagimont and D. Louvegnies. Javanaise: distributed shared objects for internet cooperative applications. In *In Proc. Middleware'98, The Lake District*, 1998. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.4418. 160

[84] D. J. Haniff and C. Baber. Wearable computers for the fire service and police force: Technological and human factors. In *Proceedings of the 3rd IEEE International Symposium on Wearable Computers*, page 185. IEEE Computer Society, 1999. ISBN 0-7695-0428-0. 103

[85] A. D. Henderson and S. Card. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Trans. Graph.*, 5(3):211–243, July 1986. ISSN 0730-0301. doi: http://dx.doi.org/10.1145/24054.24056. URL http://dx.doi.org/10.1145/24054.24056. 39

[86] C. Hess, M. Rom□ and R. Campbell. Building Applications for Ubiquitous Computing Environments. In *International Conference on Pervasive Computing (Pervasive 2002)*, pages 16–29. Springer-Verlag, 2002. 5, 85, 114, 115

[87] J. Hill and C. Gutwin. The MAUI Toolkit: Groupware Widgets for Group Awareness. *Computer Supported Cooperative Work*, 13(2):539–571, 2004. ISSN 0925-9724. 47, 142, 163

[88] J. Hollan, E. Hutchins, and D. Kirsh. Distributed cognition: toward a new foundation for human-computer interaction research. *ACM Trans. Comput.-Hum. Interact.*, 7(2):174–196, June 2000. ISSN 1073-0516. doi: http://dx.doi.org/10.1145/353485.353487. URL http://dx.doi.org/10.1145/353485.353487. 14

[89] H. Hsieh and F. Shipman. Activity links: supporting communication and reflection about action. In *HYPERTEXT '05: Proceedings of the sixteenth ACM conference on Hypertext and hypermedia*, pages 161–170, New York, NY, USA, 2005. ACM Press. ISBN 1595931686. doi: http://dx.doi.org/10.1145/1083356.1083388. URL http://dx.doi.org/10.1145/1083356.1083388. 40, 131

[90] E. Hutchins. How a cockpit remembers its speeds. *Cognitive Science*, 19: 265–288, July 1995. URL http://www.ingentaconnect.com/content/els/03640213/1995/00000019/00000003/art90020. 13, 14

[91] S. Izadi, H. Brignull, T. Rodden, Y. Rogers, and M. Underwood. Dynamo: a public interactive surface supporting the cooperative sharing and exchange of media. In *UIST '03: Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 159–168, New York, NY, USA, 2003. ACM Press. ISBN 1581136366. doi: http://dx.doi.org/10.1145/964696.964714. URL http://dx.doi.org/10.1145/964696.964714. 41, 115

[92] R. J. K. Jacob. Eye tracking in advanced interface design. pages 258–288, 1995. URL http://portal.acm.org/citation.cfm?id=216164.216178. 29

[93] D. Johansen, K. J. Lauvset, and K. Marzullo. An extensible software architecture for mobile components. In *ECBS '02: Proceeding of the 9th IEEE International Conference on Engineering of Computer-Based Systems*, pages 231–237, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0769515495. URL http://portal.acm.org/citation.cfm?id=648549. 18

[94] B. Johanson, A. Fox, and T. Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1 (2):67–74, 2002. ISSN 1536-1268. doi: http://dx.doi.org/10.1109/MPRV.2002.1012339. URL http://dx.doi.org/10.1109/MPRV.2002.1012339. 61, 114, 115

[95] B. Johanson, G. Hutchins, T. Winograd, and M. Stone. Pointright: experience with flexible input redirection in interactive workspaces. In *UIST '02: Proceedings of the 15th annual ACM symposium on User interface software and technology*, pages 227–234, New York, NY, USA, 2002. ACM Press. ISBN 1581134886. doi: http://dx.doi.org/10.1145/571985.572019. URL http://dx.doi.org/10.1145/571985.572019. 115, 172

[96] E. Kandogan and B. Shneiderman. Elastic windows: evaluation of multi-window operations. In *CHI '97: Proceedings of the SIGCHI conference on Human*

factors in computing systems, pages 250–257. ACM Press, 1997. ISBN 0-89791-802-9. doi: http://doi.acm.org/10.1145/258549.258720. 5, 85

[97] V. Kaptelinin. Umea: translating interaction histories into project contexts. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 353–360. ACM Press, 2003. ISBN 1-58113-630-7. doi: http://doi.acm.org/10.1145/642611.642673. 85

[98] V. Kaptelinin. Umea: translating interaction histories into project contexts. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 353–360. ACM Press, 2003. ISBN 1581136307. doi: http://dx.doi.org/10.1145/642611.642673. URL http://dx.doi.org/10.1145/642611.642673. 39

[99] T. Kindberg and J. Barton. A web-based nomadic computing system. *HP Labs Technical Report (HPL-2000-110)*, 2000. URL http://www.hpl.hp.com/techreports/2000/HPL-2000-110.html. 62

[100] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/146941.146942. 151

[101] G. Kortuem, M. Bauer, and Z. Segall. Netman: the design of a collaborative wearable computer system. *Mobile Network and Applications*, 4(1):49–58, 1999. ISSN 1383-469X. doi: http://dx.doi.org/10.1023/A:1019122125996. 29, 102, 104, 111

[102] R. E. Kraut, M. D. Miller, and J. Siegel. Collaboration in performance of physical tasks: effects on outcomes and communication. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 57–66. ACM Press, 1996. ISBN 0-89791-765-0. doi: http://doi.acm.org/10.1145/240080.240190. 104

[103] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992. ISSN 0360-0300. doi: http://dx.doi.org/10.1145/130844.130856. URL http://dx.doi.org/10.1145/130844.130856. 54

[104] D. Li and R. Li. Transparent sharing and interoperation of heterogeneous single-user applications. In *CSCW '02: Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 246–255. ACM Press, 2002. ISBN 1-58113-560-2. doi: http://doi.acm.org/10.1145/587078.587113. 143, 144

[105] J. Liedtke. On micro-kernel construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250, New York, NY, USA, 1995. ACM. doi: http://dx.doi.org/10.1145/224056.224075. URL http://dx.doi.org/10.1145/224056.224075. 53

[106] I. Lipkind, I. Pechtchanski, and V. Karamcheti. Object views: language support for intelligent object caching in parallel and distributed computations. *SIGPLAN Not.*, 34(10):447–460, 1999. ISSN 0362-1340. doi:

http://dx.doi.org/10.1145/320385.320433. URL http://dx.doi.org/10.1145/320385.320433. 165

[107] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: http://dx.doi.org/10.1145/53990.54016. URL http://dx.doi.org/10.1145/53990.54016. 160

[108] J. Lumsden and S. Brewster. A paradigm shift: alternative interaction techniques for use with mobile & wearable devices. In *Proceedings of the 2003 conference of the Centre for Advanced Studies conference on Collaborative research*, pages 197–210. IBM Press, 2003. 104

[109] B. MacIntyre, E. D. Mynatt, S. Vodia, K. M. Hansen, J. Tullio, and G. M. Corso. Support for Multitasking and Background Awareness Using Interactive Peripheral Displays. In *Proceeding of ACM User Interface Software and Technology 2001 (UIST01)*, pages 11–14, Orlando, Florida, USA, Nov. 2001. 5, 85

[110] B. Macintyre, E. D. Mynatt, S. Voida, K. M. Hansen, J. Tullio, and G. M. Corso. Support for multitasking and background awareness using interactive peripheral displays. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 41–50, New York, NY, USA, 2001. ACM Press. ISBN 158113438X. doi: http://dx.doi.org/10.1145/502348.502355. URL http://dx.doi.org/10.1145/502348.502355. 40, 115

[111] P. Madsen and N. T. Johansen. Rover: A data management infrastructure for activity-based computing - extending the support for user mobility. Master's thesis, 2005. 66

[112] T. W. Malone. How do people organize their desks?: Implications for the design of office information systems. *ACM Trans. Inf. Syst.*, 1(1):99–112, January 1983. ISSN 1046-8188. doi: http://dx.doi.org/10.1145/357423.357430. URL http://dx.doi.org/10.1145/357423.357430. 21

[113] S. T. March and G. F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, December 1995. doi: http://dx.doi.org/10.1016/0167-9236(94)00041-2. URL http://dx.doi.org/10.1016/0167-9236(94)00041-2. 7, 9

[114] G. Mark, V. M. Gonzalez, and J. Harris. No task left behind?: examining the nature of fragmented work. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 321–330. ACM Press, 2005. ISBN 1-58113-998-5. doi: http://doi.acm.org/10.1145/1054972.1055017. 84

[115] M. P. Mccahill and J. Lombardi. Design for an extensible croquet-based framework to deliver a persistent, unified, massively multi-user, and self-organizing virtual environment. In *C5 '04: Proceedings of the Second Inter-*

national Conference on Creating, Connecting and Collaborating through Computing, pages 71–77, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2166-5. doi: http://dx.doi.org/10.1109/C5.2004.13. URL http://dx.doi.org/10.1109/C5.2004.13. 165

[116] J. Mccarthy, T. Costa, and E. Liongosari. Unicast, outcast & groupcast: Three steps toward ubiquitous, peripheral displays. pages 332–345. 2001. doi: http://dx.doi.org/10.1007/3-540-45427-6_28. URL http://dx.doi.org/10.1007/3-540-45427-6_28. 40

[117] J. F. Mccarthy, D. W. Mcdonald, S. Soroczak, D. H. Nguyen, and A. M. Rashid. Augmenting the social space of an academic conference. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 39–48, New York, NY, USA, 2004. ACM Press. ISBN 158113810-5. doi: http://dx.doi.org/10.1145/1031607.1031615. URL http://dx.doi.org/10.1145/1031607.1031615. 40

[118] Microsoft. Com interop. . URL http://msdn.microsoft.com/en-us/library/6bw51z5z.aspx. 67

[119] Microsoft. Research desktop - microsoft research. . URL http://research.microsoft.com/en-us/projects/researchdesktop/default.aspx. 39

[120] M. Mikic-Rakic and N. Medvidovic. A connector-aware middleware for distributed deployment and mobility. In *ICDCSW '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0769519210. URL http://portal.acm.org/citation.cfm?id=839280.840686. 18

[121] S. Minneman, S. Harrison, B. Janssen, T. Moran, G. Kurtenbach, and I. Smith. A confederation of tools for capturing and accessing collaborative activity. URL http://delivery.acm.org/10.1145/220000/215316/p523-minneman.htm?key1=215316\&key2=7100173411\&coll=Portal\&dl=GUIDE\&CFID=68416221\&CFTOKEN=14165286. 40

[122] M. Mogensen. Distributed objects in loose coupled local area networks. Technical Report, Computer Science Department, University of Aarhus, 2005. 53, 169

[123] P. Moody, D. Gruen, M. J. Muller, J. Tang, and T. P. Moran. Business activity patterns: A new model for collaborative business applications. *IBM Systems Journal*, 45(4), 2006. URL http://www.research.ibm.com/journal/sj/454/moody.html. 5, 115

[124] T. P. Moran. Unified activity management: Explicitly representing activity in work-support systems. In *European Conference on Computer Supported Cooperative Work*, September 2005. URL http://www.daimi.au.dk/~bardram/ecscw2005/papers/moran.pdf. 5, 66, 115

[125] M. J. Muller, W. Geyer, B. Brownholtz, E. Wilcox, and D. R. Millen. One-hundred days in an activity-centric collaboration environment based on shared objects. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 375–382. ACM Press, 2004. ISBN 1-58113-702-8. doi: http://doi.acm.org/10.1145/985692.985740. 5, 85

[126] T. Nakajima and I. Satoh. A software infrastructure for supporting spontaneous and personalized interaction in home computing environments. *Personal and Ubiquitous Computing*, 10(6):379–391, October 2006. ISSN 1617-4909. doi: http://dx.doi.org/10.1007/s00779-005-0056-1. URL http://dx.doi.org/10.1007/s00779-005-0056-1. 74

[127] J. J. Ockerman and A. R. Pritchett. Preliminary investigation of wearable computers for task guidance in aircraft inspection. In *Proceedings of the 2nd IEEE International Symposium on Wearable Computers*, page 33. IEEE Computer Society, 1998. ISBN 0-8186-9074-7. 103

[128] D. L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. URL http://portal.acm.org/citation.cfm?id=800099.803218. 54

[129] J. Pascoe. Adding generic contextual capabilities to wearable computers. In *Proceedings of the 2nd IEEE International Symposium on Wearable Computers*, page 92. IEEE Computer Society, 1998. ISBN 0-8186-9074-7. 103

[130] Patrick, R. Guerraoui, and J. Sventek. Distributed asynchronous collections: Abstractions for publish/subscribe interaction. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 252–276, London, UK, 2000. Springer-Verlag. ISBN 3-540-67660-0. URL http://portal.acm.org/citation.cfm?id=758679. 160

[131] Patrick, R. Guerraoui, and C. H. Damm. On objects and events. *SIGPLAN Not.*, 36(11):254–269, November 2001. ISSN 0362-1340. doi: http://dx.doi.org/10.1145/504311.504301. URL http://dx.doi.org/10.1145/504311.504301. 160

[132] Patrick, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. ISSN 0360-0300. doi: http://dx.doi.org/10.1145/857076.857078. URL http://dx.doi.org/10.1145/857076.857078. 62

[133] J. F. Patterson, R. D. Hill, S. L. Rohall, and S. W. Meeks. Rendezvous: an architecture for synchronous multi-user applications. In *CSCW '90: Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pages 317–328. ACM Press, 1990. ISBN 0-89791-402-3. doi: http://doi.acm.org/10.1145/99332.99364. 48, 147

[134] J. F. Patterson, M. Day, and J. Kucan. Notification servers for synchronous groupware. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer*

supported cooperative work, pages 122–129. ACM Press, 1996. ISBN 0-89791-765-0. doi: http://doi.acm.org/10.1145/240080.240232. 48, 147, 156

[135] A. Prakash. Group editors. In Beaudouin-Lafon [27], pages 103–134. 142, 144, 147, 163

[136] T. Rattenbury and J. Canny. Caad: an automatic task support system. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 687–696, New York, NY, USA, 2007. ACM Press. ISBN 9781595935939. doi: http://dx.doi.org/10.1145/1240624.1240731. URL http://dx.doi.org/10.1145/1240624.1240731. 115

[137] Realvnc. Realvnc - vnc - the original cross-platform remote control solution. URL http://www.realvnc.com/vnc/index.html. 5, 39, 172

[138] J. Rekimoto. Time-machine computing: a time-centric approach for the information environment. In *UIST '99: Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 45–54, New York, NY, USA, 1999. ACM Press. ISBN 1581130759. doi: http://dx.doi.org/10.1145/320719.322582. URL http://dx.doi.org/10.1145/320719.322582. 40

[139] J. Rekimoto. Time-machine computing: a time-centric approach for the information environment. In *UIST '99: Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 45–54. ACM Press, 1999. ISBN 1-58113-075-9. doi: http://doi.acm.org/10.1145/320719.322582. 5, 85

[140] J. Rekimoto and M. Saitoh. Augmented surfaces: a spatially continuous work space for hybrid computing environments. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 378–385, New York, NY, USA, 1999. ACM Press. ISBN 0201485591. doi: http://dx.doi.org/10.1145/302979.303113. URL http://dx.doi.org/10.1145/302979.303113. 114

[141] Roberto. Yet another configurable extensible event service. URL http://awareness.ics.uci.edu/~rsilvafi/yancees/. 62

[142] Roberto and D. F. Redmiles. Striving for versatility in publish/subscribe infrastructures. In *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*, pages 17–24, New York, NY, USA, 2005. ACM. ISBN 1-59593-204-4. doi: http://dx.doi.org/10.1145/1108473.1108478. URL http://dx.doi.org/10.1145/1108473.1108478. 62

[143] G. Robertson, M. van Dantzich, D. Robbins, M. Czerwinski, K. Hinckley, K. Risden, D. Thiel, and V. Gorokhovsky. The task gallery: a 3d window manager. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 494–501. ACM Press, 2000. ISBN 1-58113-216-6. doi: http://doi.acm.org/10.1145/332040.332482. 5, 26, 85, 91

[144] G. Robertson, M. van Dantzich, D. Robbins, M. Czerwinski, K. Hinckley, K. Risden, D. Thiel, and V. Gorokhovsky. The task gallery: a 3d window manager. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 494–501, New York, NY, USA, 2000. ACM Press. ISBN 1581132166. doi: http://dx.doi.org/10.1145/332040.332482. URL http://dx.doi.org/10.1145/332040.332482. 39

[145] G. Robertson, E. Horvitz, M. Czerwinski, P. Baudisch, D. R. Hutchings, B. Meyers, D. Robbins, and G. Smith. Scalable fabric: flexible task management. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 85–89. ACM Press, 2004. ISBN 1-58113-867-9. doi: http://doi.acm.org/10.1145/989863.989874. 5, 84, 85

[146] D. B. Roe and J. G. Wilpon, editors. *Voice communication between humans and machines*. National Academy Press, Washington, DC, USA, 1994. ISBN 0-309-04988-1. URL http://portal.acm.org/citation.cfm?id=191946. 29

[147] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: a middleware platform for active spaces. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):65–67, October 2002. ISSN 1559-1662. doi: http://dx.doi.org/10.1145/643550.643558. URL http://dx.doi.org/10.1145/643550.643558. 60

[148] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A middleware infrastructure for active spaces. *Pervasive Computing, IEEE*, 1(4):74–83, 2002. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1158281. 61

[149] M. Roman, B. Ziebart, and R. H. Campbell. Dynamic application composition: customizing the behavior of an active space. In *Pervasive Computing and Communications, 2003. (PerCom 2003). Proceedings of the First IEEE International Conference on*, pages 169–176, 2003. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1192739. 73, 172, 173, 185

[150] M. Roseman and S. Greenberg. Building real-time groupware with groupkit, a groupware toolkit. *ACM Trans. Comput.-Hum. Interact.*, 3(1):66–106, 1996. ISSN 1073-0516. doi: http://doi.acm.org/10.1145/226159.226162. 47, 142, 156, 163

[151] D. Salber and J. Coutaz. Applying the wizard of oz technique to the study of multimodal systems. pages 219–230. 1993. doi: http://dx.doi.org/10.1007/3-540-57433-6_51. URL http://dx.doi.org/10.1007/3-540-57433-6_51. 120

[152] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, New York, NY, USA, 1999. ACM Press. ISBN 0201485591. doi: http://dx.doi.org/10.1145/302979.303126. URL http://dx.doi.org/10.1145/302979.303126. 172

[153] I. Satoh. Mobilespaces: A framework for building adaptive distributed applications using a hierarchical mobile agent system. In *ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0769506011. URL http://portal.acm.org/citation.cfm?id=851788. 18, 73, 74, 185

[154] W. Schafer and D. Bowman. Supporting distributed spatial collaboration: An investigation of navigation and radar view techniques. *GeoInformatica*, 10(2):123–158, June 2006. ISSN 1384-6175. doi: http://dx.doi.org/10.1007/s10707-006-7576-3. URL http://dx.doi.org/10.1007/s10707-006-7576-3. 27

[155] M. Schoeman and E. Cloete. Architectural components for the efficient design of mobile agent systems. In *SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, pages 48–58, , Republic of South Africa, 2003. South African Institute for Computer Scientists and Information Technologists. ISBN 1581137745. URL http://portal.acm.org/citation.cfm?id=954020. 18

[156] C. Schuckmann, L. Kirchner, J. Schümmer, and J. M. Haake. Designing object-oriented synchronous groupware with coast. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 30–38, New York, NY, USA, 1996. ACM. ISBN 0-89791-765-0. doi: http://dx.doi.org/10.1145/240080.240186. URL http://dx.doi.org/10.1145/240080.240186. 61

[157] H. S. Shim, R. W. Hall, A. Prakash, and F. Jahanian. Providing Flexible Services for Managing Shared State in Collaborative Systems. In T. Rodden, J. Hughes, and K. Schmidtk, editors, *Proceedings of the Fifth European Conference on Computer Supported Cooperative Work*, pages 237–252, Lancaster, UK, Sept. 1997. Kluwer Academic Publishers. 47, 142, 144, 163, 165

[158] F. M. Shipman, H. Hsieh, P. Maloor, and M. J. Moore. The visual knowledge builder: a second generation spatial hypertext. In *HYPERTEXT '01: Proceedings of the twelfth ACM conference on Hypertext and Hypermedia*, pages 113–122, New York, NY, USA, 2001. ACM Press. ISBN 1591134207. doi: http://dx.doi.org/10.1145/504216.504245. URL http://dx.doi.org/10.1145/504216.504245. 131

[159] D. Siewiorek, A. Smailagic, L. Bass, J. Siegel, R. Martin, and B. Bennington. Adtranz: A mobile computing system for maintenance and collaboration. In *Proceedings of the 2nd IEEE International Symposium on Wearable Computers*, page 25. IEEE Computer Society, 1998. ISBN 0-8186-9074-7. 29, 102, 103

[160] A. Smailagic and D. P. Siewiorek. A case study in embedded-system design: The vuman 2 wearable computer. *IEEE Design and Test*, 10(3):56–67, 1993. ISSN 0740-7475. doi: http://dx.doi.org/10.1109/54.232473. 29, 102

[161] D. A. Smith, A. Kay, A. Raab, and D. P. Reed. Croquet - a collaboration system architecture. In *Creating, Connecting and Collaborating Through Computing, 2003. C5 2003. Proceedings. First Conference on*, pages 2–9, 2003. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1222325. 165

[162] G. Smith, P. Baudisch, G. Robertson, M. Czerwinski, B. Meyers, D. Robbins, and D. Andrews. Groupbar: The taskbar evolved. In *Proceedings of OZCHI 2003*, pages 34–43, 2003. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.9622. 39

[163] G. Smith, P. Baudisch, G. G. Robertson, M. Czerwinski, B. Meyers, D. Robbins, and D. Andrews. Groupbar: The taskbar evolved. In *Proceedings of OZCHI 2003*, 2003. 5, 84

[164] S. Sørensen. Context-aware public display for activity-based computing. Master's thesis, August 2008. 32

[165] J. P. Sousa and D. Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *Proceeding of the 3rd Working IEEE/IFIP Conference on Software Architecture*, Montreal, August 25-31 2002. 102

[166] M. Stefik, G. Foster, D. G. Bobrow, K. Kahn, S. Lanning, and L. Suchman. Beyond the chalkboard: computer support for collaboration and problem solving in meetings. *Commun. ACM*, 30(1):32–47, January 1987. ISSN 0001-0782. doi: http://dx.doi.org/10.1145/7885.7887. URL http://dx.doi.org/10.1145/7885.7887. 114

[167] N. Streitz, C. Röcker, T. Prante, R. Stenzel, and D. Van Alphen. Abstract situated interaction with ambient information: Facilitating awareness and communication in ubiquitous work environments, 2003. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.6186. 41

[168] N. A. Streitz, J. Geissler, T. Holmer, S. Konomi, C. Müller-Tomfelde, W. Reischl, P. Rexroth, P. Seitz, and R. Steinmetz. i-land: an interactive landscape for creativity and innovation. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 120–127, New York, NY, USA, 1999. ACM Press. ISBN 0201485591. doi: http://dx.doi.org/10.1145/302979.303010. URL http://dx.doi.org/10.1145/302979.303010. 6, 61, 114, 115

[169] L. A. Suchman. *Plans and Situated Actions : The Problem of Human-Machine Communication (Learning in Doing: Social, Cognitive & Computational Perspectives)*. Cambridge University Press, November 1987. ISBN 0521337399. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0521337399. 14

[170] D. S. Tan, B. Meyers, and M. Czerwinski. Wincuts: manipulating arbitrary window regions for more effective use of screen space. In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, pages 1525–1528, New

York, NY, USA, 2004. ACM. ISBN 1581137036. doi: http://dx.doi.org/10.1145/985921.986106. URL http://dx.doi.org/10.1145/985921.986106. 172

[171] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, October 2006. ISBN 0132392275. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0132392275. 44

[172] C. Tashman. Windowscape: a task oriented window manager. In *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 77–80, New York, NY, USA, 2006. ACM Press. ISBN 1595933131. doi: http://dx.doi.org/10.1145/1166253.1166266. URL http://dx.doi.org/10.1145/1166253.1166266. 40

[173] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for gui software. *Software Engineering, IEEE Transactions on*, 22(6):390–406, 1996. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=508313. 73

[174] S. Vinoski. Corba: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997. doi: http://dx.doi.org/10.1109/35.565655. URL http://dx.doi.org/10.1109/35.565655. 160

[175] D. Vogel and R. Balakrishnan. Interactive public ambient displays: transitioning from implicit to explicit, public to personal, interaction with multiple users. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 137–146, New York, NY, USA, 2004. ACM Press. ISBN 1581139578. doi: http://dx.doi.org/10.1145/1029632.1029656. URL http://dx.doi.org/10.1145/1029632.1029656. 41

[176] J. Vogel, W. Geyer, L.-T. Cheng, and M. Muller. Consistency control for synchronous and asynchronous collaboration based on shared objects and activities. *Computer Supported Cooperative Work (CSCW)*, 13(5):573–602, December 2004. doi: http://dx.doi.org/10.1007/s10606-004-5064-6. URL http://dx.doi.org/10.1007/s10606-004-5064-6. 145, 156, 157

[177] J. Vogel, W. Geyer, L.-T. Cheng, and M. J. Muller. Consistency control for synchronous and asynchronous collaboration based on shared objects and activities. *Computer Supported Cooperative Work*, 13(5-6):573–602, 2004. 5, 85

[178] S. Voida, M. Mckeon, C. Le Dantec, C. Forslund, P. Verma, B. Mcmillan, J. Bunde-Pedersen, K. Edwards, E. Mynatt, and A. Mazalek. inspace: Co-designing the physical and digital environment to support workplace collaboration. Technical Report GIT-GVU-08-03, 2007. URL http://www.gvu.gatech.edu/learning/tr0803.php. 59, 182

[179] S. Voida, E. D. Mynatt, and W. K. Edwards. Re-framing the desktop interface around the activities of knowledge work. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*,

pages 211–220, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-975-3. doi: http://dx.doi.org/10.1145/1449715.1449751. URL http://dx.doi.org/10.1145/1449715.1449751. 39

[180] L. S. Vygotsky. *Mind in Society - The Development of Higher Psychological Processes.* Michael Cole, Vera John-Steiner, Sylvia Scribner, and Ellen Souberman (eds). Harvard University Press, Massachusetts, USA, 1978. 13

[181] J. Walkerdine, L. Melville, and I. Sommerville. Dependability properties of p2p architectures. In *Peer-to-Peer Computing*, pages 173–174, 2002. 45, 46, 145, 146, 164

[182] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 306–315, New York, NY, USA, 1988. ACM. ISBN 0-89791-284-5. doi: http://dx.doi.org/10.1145/62083.62111. URL http://dx.doi.org/10.1145/62083.62111. 160

[183] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3): 66–75, September 1991. 115

[184] M. Weiser. The computer for the 21st century. pages 933–940, 1995. URL http://portal.acm.org/citation.cfm?id=213017. 74

[185] S. Xia, D. Sun, C. Sun, D. Chen, and H. Shen. Leveraging single-user applications for multi-user collaboration: the coword approach. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 162–171, New York, NY, USA, 2004. ACM. ISBN 158113810 5. doi: http://dx.doi.org/10.1145/1031607.1031635. URL http://dx.doi.org/10.1145/1031607.1031635. 47

[186] M. J. Zieniewicz, D. C. Johnson, D. C. Wong, and J. D. Flatt. The Evolution of Army Wearable Computers. *IEEE Pervasive Computing*, 1(4):30–40, Oct. 2002. 103

[187] J. Zobel. *Writing for Computer Science.* Springer, May 2004. ISBN 1852338024. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/1852338024. 9