CfPC Technical Report
Real-Time Collaboration in Activity Based Architectures

Jakob E. Bardram and Henrik Bærbak Christensen
University of Aarhus
Computer Science Department
Aabogade 34, 8200 Aarhus N, Denmark
{bardram,hbc}@daimi.au.dk

Abstract

With the growing research into mobile and ubiquitous computing, there is a need for addressing how such infrastructures can support collaboration between users while removed from the desktop. In this paper, we present the principles of Activity Based Computing (ABC), focusing on its support for collaboration in a ubiquitous computing environment. We discuss how this activity-centered design principle establish a conceptual and software architectural basis for session management in real-time, synchronous collaboration. We present the architecture of the ABC Framework, which is an implementation of these principles and discuss how collaboration support is made as part of the activity-centered architecture. We also discuss the need for more fine-grained state management policies in ubiquitous collaborative systems.

1 Introduction

Pervasive computing technologies where a multitude of interconnected computing devices are freely and easily available hold the promise to support nomadic and collaborative work to a much higher degree than traditional stationary computers. However, to fully utilize the potential that new hardware devices and networks provide we have to look critically at the software side and the architectural aspects underlying it.

A major problem is that prevailing software platforms are based on the desktop metaphor, that essentially assumes stationary and concentrated work in which computational artifacts plays the central role: editing documents, handling e-mail, composing music, etc. Nomadic work, of which clinical work at a hospital is a major example, is just the opposite: characterized by mobility, interruptions, collaboration, and a focus on people, not computers.

The Activity Based Computing framework (ABC) [1, 8] and Project Aura [4, 10] have proposed a new paradigm: Activity Based or Task Level computing. In these paradigms, the human notion of a task or an activity is directly supported by the software architecture, thereby trying to provide a more explicit support for mobility, interruptions, and frequent change of devices. So far these proposals have focused primarily on the architectural aspects of handling an individual's personal set of activities: how to suspend and restore them, how to migrate them between devices, how to represent them, how to develop activity aware applications, etc. However, in real life people collaborate to complete tasks and an activity aware architecture must address this important, but complex, topic.

The main contribution of this paper is to report on architectural aspects of supporting *collaboration* in an activity based computing context. We report that the activity based computing paradigm is a natural and strong paradigm for supporting collaboration, especially in order to treat the problem of session management. Next, we argue that an activity management system alone cannot adequately handle collaboration due to the high volume of fine-grained information that must be exchanged between participating devices, and propose an architecture that introduce additional components to manage collaborative information. We present our initial architecture for collaboration support in the ABC framework and discuss observations made during evaluation in a number of workshops.

The paper is organized as follows. Section 2 describes the notion of activities and ABC in more detail. Section 3 describes the extension to ABC to support collaboration mediated through activities while a description and discussion of the architecture is found in section 4 and 5. Finally, we relate to other work within the field and conclude in section 6 and 7.

2 Activity Based Computing

Weiser defined Ubiquitous Computing as a situation where large numbers of computers were embedded in everyday artifacts and used and handled by users without much thought. Hardware advances have lead to very powerful computers with networking abilities that are very small and we have seen an explosion of mobile devices with a range of different and interesting characteristics. These devices form the foundation on which to build pervasive computing systems supporting nomadic work and a much more flexible utilization of available computational resources.

An excellent example is healthcare work in hospitals. Clinicians' work is characterized by mobility, the need to negotiate large amounts of complex information, constant interruptions, and a high degree of collaboration. A future scenario where physicians may use the computer built into the patient's bed to review a patient record, browse the medicine handbook on a hand-held he has just grabbed from the nearest docking-station, or discuss an X-ray image on a wall-sized screen with the expert radiologist located at his home, seems promising and appealing. There is no need to carry heavy or bulky equipment, rather patient records are available at any nearby device whose characteristics fit the task at hand.

However, the software architectures underlying the systems running on the new devices have taken their inspiration from the well known document-centered and application-centered paradigms known from the desktop personal computer. The prevailing document-centered desktop paradigm does not exploit the potential of pervasive computing well. Desktop computing is geared towards stationary and concentrated work in which computational artifacts plays the central role: editing documents, handling e-mail, composing music, etc. Nomadic work, in contrast, is characterized by mobility, interruption, and is concentrated on people, issues or artefacts that are only indirectly related to computing.

Several research projects have responded to this challenge by suggesting that the *human activity* or *task* is important to support, not the document nor the application [10, 16, 8]. Observing a physician using a computer for medical work would reveal that she is viewing an X-ray image, browsing the medicine handbook, and navigating the medicine schema in the patient's medical record. However, if you ask her, she will tell you that she is prescribing medicine for this patient. Mainstream computing platforms support the *application level*: the X-ray viewer, the handbook browser, and the mecidine schema application; however, it has no notion of the real activity, namely to prescribe medicine. This *abstraction gap* is illustrated in figure 1.

The Activity Based Computing framework (ABC) [1]
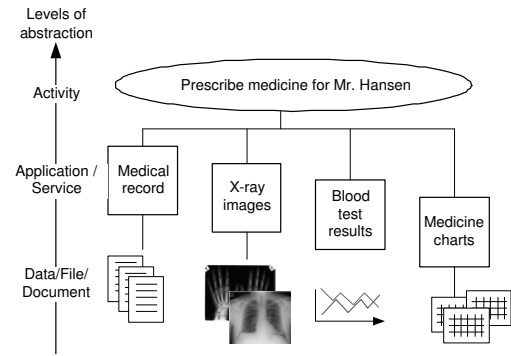
Figure 1. A single activity involves many computational services, which again each manipulates a number of data.

and Project Aura [4] have proposed to support human activities and tasks as *first class objects at the architectural level*. Activities embody every running applications' run-time state, and may be suspended and resumed on any computational device. This supports mobility and interruption: the physician in the middle of prescribing medicine that is forced to attend another patient briefly, simply suspends the activity (storing the state of X-ray viewer, medicine schema, etc.). Later, she finds an available computer and resumes the activity restoring all applications as she left them.

Technically speaking ABC defines an activity as:

> **Activity:** An abstract, but comprehensive, description of the run-time state of a set of computational services.

Figure 1 sketch this as an UML class diagram.

By 'computational services' we mean applications with some well defined responsibility, like a text editor, a medicine catalogue browser, an X-ray image viewer, medicine schema editor, prescription tool, web browser, etc. Service is more abstract than application, for instance we may view X-rays on both a laptop and a PDA even though the display application (software) is radically different. Thus several different device-specific applications may provide the same service.

By 'set of services' we mean that any human activity often require a number of services running concurrently. To prescribe the physician needs access to X-rays, the medicine book, the patient record, blood sample graphs, etc.

By 'run-time state' we mean that ABC can 'snapshot' the set of running applications' state and store it, transfer it to another computer, and generally treat the state snapshot set as a first class object to be manipulated.

By 'abstract, but comprehensive, description' we mean that the state description can be understood and interpreted
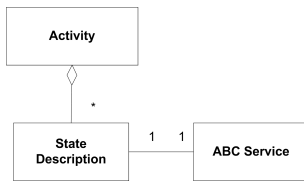
Figure 2. The activity concept as UML class diagram.

by a number of radically different applications as long as they provide the same service (abstract property), and the state description should contain enough information for the service to reestablish a context very similar to the context when the activity was snapshot (comprehensive property). For example to reestablish a particular X-ray image with the proper pan and zoom on a number of different applications as long as they are a X-ray viewer services. The semantics of the state description is exclusively defined by the service itself.

Given this description of activities, we then define:

> **Activity Based Computing:** A computing infrastructure, which support suspending and resuming activities across heterogeneous execution environments and supports activity based collaboration

3 Collaborative Activities

Before we present the support for collaborative activities in the ABC architecture, we will give an impression of this collabotation support by an end-user scenario. The following scenario is fully supported by the ABC framework as is, and have been evaluated through role-playing in workshops by nurses and physician from the University Hospital of Aarhus.

3.1 A Radiology Conference

Dr. Urban is working on his activity concerning patient Petersen. This includes open views on the patient record's medicine schema, medical notes, and X-rays (part of the ABC user-interface is shown in figure 4). Something on the X-rays alerts Dr. Urban and he wants to discuss the details with the radiologist on duty. He invites *the radiologist to participate in the activity. The radiologist on duty – Dr. John Jensen – is* notified *and accepts to join the activity. Upon joning, his laptop screen is reconfigured as it* activates *Dr. Urban's activity, that is, services that support viewing X-rays, browsing medicine schemas, etc., are started; proper patient information fetched, and graphical*

views adjusted to reflect the same data and visual presentation as Dr. Urban is working on. Next, as the activity is collaborative, additional collaborative tools are started, for instance telepointers and audio link.

Having established the context for the collaboration, Dr. Urban describes the question for Dr. Jensen, by voice, by the ActivityChat (see section 3.3), by arranging windows showing X-ray images etc., and by showing areas of interest using the telepointer.
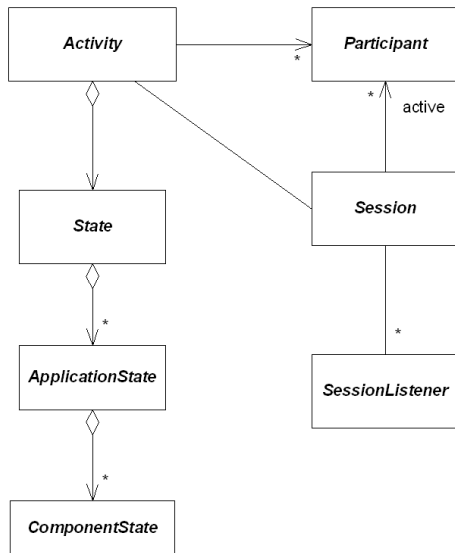
3.2 Collaborative Activities

The scenario stresses a very important aspect of activity based computing with regards to collaboration, namely that *a human activity serves as the basis for collaboration.* Dr. Urban is doing a specific task, and finds that he needs the assistence of a colleague. Thus collaboration springs from the concrete activity. Indeed, often it is a concrete task that you as a worker, as a family member, or as a person is engaged in, that poses some aspects that you need some second opinion on, cannot do alone, or in other aspects require the assistance of another person.

The key point is that as activities are first class objects in activity based architectures they naturally becomes the center-point of collaboration. A recurring problem in Computer Supported Collaborative Work (CSCW) systems is that of *session management*, i.e. how do users get in contact with each other, how do they know who 'makes the first move', how is communication initialized, and what is the purpose of the collaborative effort? Activities provide a feasible solution as they act as the 'collaborative session' defining the persons involved (Urban and the radiologist), the purpose of the collaborative effort (X-rays of the patient), as well as serve as the technical backbone for the communication.

At first sight, activities may even suffice. As activities conceptually "snapshot" the run-time state of a set of application we can, at least in theory, use this mechanism to provide full collaborative WYSIWIS[1], namely by treating each user event in any service as an implicit snapshot operation, followed by a broadcast of the resulting activity to all participants of the activity, and perform an immediate resume operation on the participants' devices. Thus every device would be kept in complete lock-step with all others.

In practice, however, this is not feasible. The primary concern is performance. Consider a collaborative effort with five persons engaged in the same activity. If a person moves a scroll bar ten pixels, then state information is collected in all services; bundled together in a state object; send to the server; broadcasted to all collaborative clients; unpacked; and parsed by each ABC service, which reconfigures the state of the service according to this state object.
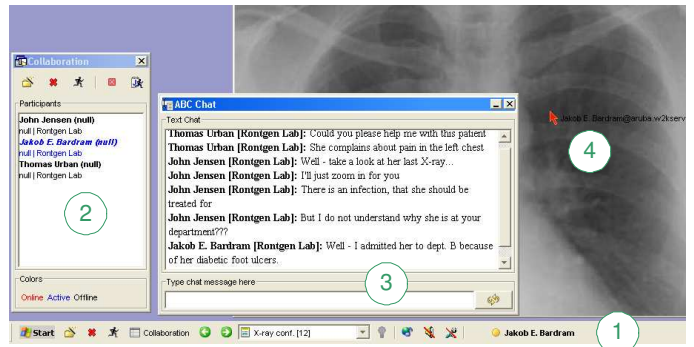
---
[1]What You See Is What I See.

Figure 3. Collaborative Activity (UML class diagram).

Figure 4. The ABC Client showing a radiology conference activity: (1) the activity controller; (2) the collaboration frame listing the participants in the activity, (3) the Activity-Chat used for chatting between the activity's participants, an (4) an ABC service for viewing X-ray images with a telepointer.

This is clearly infeasible. A secondary concern is conflict resolution and merging policies of state information. For instance if two different users enter a character in text box at nearly the same time. Then two independent activities are broadcast but neither of them reflects the change in the other. This will lead to inconsistencies or merging algoritms will have to be employed to sort things out.

We have responded to this challenge by keeping the main 'snapshot-broadcast-resume' idea, but to introduce a more fine-grained perception of state information in activities; and provide infra-structure to distribute fine-grained state changes in a more light-weight manner. The more complex activity concept is outlined in figure 3.

An activity now maintains a list of *participants* working on it. Associated to the activity is a *session object* that provides auxiliary collaborative aspects such as voice-link and telepointers. The session object further more maintains a list of *active* participants, i.e. the participants that are actively engaged in collaborative work. Objects implementing the *SessionListener* interface may attach themselves to sessions using the observer pattern protocol and are notified when sessions change state, like active participants joining or leaving the work on a given activity.

Finally, the set of state descriptions is refined. Instead of a single state description per ABC service, it is now refined into a set of *component descriptions*. A component description is a fine-grained state description typically holding the state of a single graphical user interface component, like a text entry field, a scroll bar, etc. It may, of course, also be non user interface related information. This means that

rather than packing-broadcasting-resuming full state information for a number of services we need only pack, broadcast, and 'resume' information for a very small delta-state, like for instance only the typed character in a text field, the 10 pixel movement of a scroll-bars, etc.

Furthermore, we have implemented a "mirror" of a subset of the Java Swing component that are able to pack and parse component descriptions meaning that developers of ABC services may build GUI's of what seems like standard Swing components and not have to worry about how to program state packing and parsing.

3.3 ActivityChat – The Developers Scenario

The scenario above illustrates what ABC applications would look like for the end-user. By showing how a simple chat program can be made, this section provides a view on the ABC framework seen from the application developer's perspective.

The chat program is designed to be a chat between an activity's particpants. When the participants work on the activity, they each see the chat window within the ABC Client (figure 4). The chat application supports both synchronous chatting, where people online chat with each other simultaneously, as well as asynchronous chat where users take turn in the conversation when they work on the activity.

Figure 5 shows the important parts of the source code for this ActivityChat application. The methods `getABCState()` and `setABCState()` are called by the ABC middleware during suspend and resume respectively. The `ChatState` object maintains a copy of the chat

4

```
package dk.pervasive.abc.apps.chat;

import dk.pervasive.abc.state.ComponentState;

public class ChatApplication extends ABCFrame
  implements ChatListener {

  private ChatPanel panel = null;

  private void draw() {
    ...
    this.getContentPane().add(getChatPanel());
    getChatPanel().addChatListener(this);
    ...
  }

  ...

  public void setABCState(ComponentState state) {
    if(state instanceof ChatState) {
      ChatState s = (ChatState)state;
      getChatPanel().setChatText(s.getDocument());
    }
  }

  public ComponentState getABCState() {
    ChatState state = new ChatState();
    state.setDocument(getChatPanel().getChatText());
    state.setComponentId(APP_KEY);
    return state;
  }

  public void chatChanged() {
    // super.stateChanged();
    fireABCStateChanged();
  }
}
```

Figure 5. The source code for the ActivityChat shown in figure 4.

text, enabling this text to be restored and save upon activity resume and suspension.

The ActivityChat is using some of the ABC Swing components for fine-grained state management and collaboration. The applications inherits from the generic `ABCFrame` class, which is an ABC wrapper class for JFrame in Java Swing. The `ABCFrame` class handles state information for the size, position, and visibility of an application frame. In the `ChatPanel` class the ABC version of the `JScrollPane` is used in the chat panel shown in figure 4. This class manages state information on positions of the vertical and horizontal scrollbars. Hence, by using these ABC Swing components, the chat application only needs to handle state information specifically related to the chat—in this case the chat text. The `ABCFrame` and the `dk.pervasive.abc.swing.JScrollPane` all implements the `StatefulComponent` interface and such components knows how to set and get the component's

state in cooperation with the State Manager running on each client. The ABC Swing components have the same names as their fellow Swing components just belonging to the `dk.pervasive.abc.swing.*` package. Hence the programmer creates a Swing GUI as usual and makes it ABC-aware by just replacing e.g. the `import javax.swing.JScrollPane` with `import dk.pervasive.abc.swing.JScrollPane`. Hence, minimum overhead is associated with creating ABC- and collaboration-aware GUIs.

4 Architecture

The Activity Based Computing is a framework that allows activity-enabled services to be programmed with minimal effort. It basically provides two things:

1. A run-time infrastructure (middleware). This consists of both a central server that stores and manages a set of activities and services related to activities; and a client-side run-time middleware component. The latter allows client-side services to communicate with the server; allows the user to manage, inspect, and suspend/resume activities; and handles distribution of fine-grained component state changes to activity participants.

2. A framework for developing activity aware services. This takes the form of a small set of interfaces and contracts that developers must implement and adhere to in order for their services to communicate with the ABC middleware. Basically, services must implement functionality to pack their run-time state into a state object (for suspend) and parse this object back into a running service (for resume). Collaboration support is provided through an elaboration of (a subset of) Java Swing components that provide default packing and parsing of state descriptions.

4.1 Main components

As activities must be able to be resumed on any device running the ABC middleware, we have chosen a repository architectural style. As outlined in the UML class diagram 6 an *Activity manager* with associated *Activity Store* runs on a central server. On each client runs an *Activity Controller* that commmunicate activity objects with the central activity manager. The deployment is exemplified in diagram 7, showing two ABC clients handling a collaborative activity.

The responsibilities of the components are:

- *Activity Manager* is a server-side process that manages activity objects. Activities reside on the server side; clients communicate with them using remote method
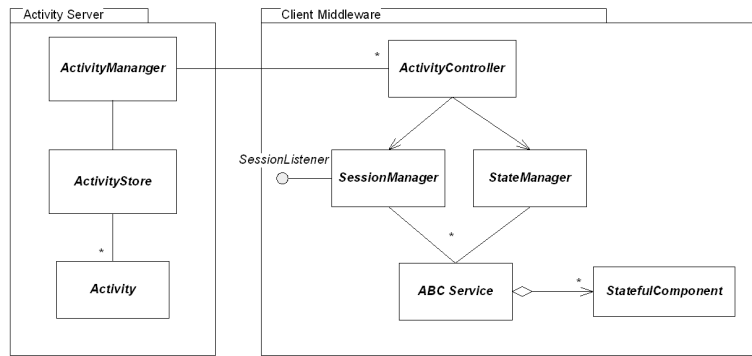
Figure 6. UML class diagram showing main functional components of the ABC architecture

invocations, for intance to retrieve their state descriptions (application and component states).

- *ActivityStore* is a persistent store/database of activities that are not at the moment operational on any device. Thus the list of suspended (i.e. pending) activities form a to-do list for a user.

- *Activity Controller* is a client side component that handles the communication with the server side activity manager. The activity controller has a user interface that allows users to browse his/her activities, and resume any from the list. Our present user interface is marked as (1) in figure 4.

- *State Manager* is responsible for handling resume and suspend of a given activity. During resume a remote reference to an activity is provided by the controller; the state manager ensures that the proper set of ABC services are running as defined by the activity; and next passes each service the appropriate application and component states for parsing and service reconfiguration. During suspend, the state manager asks each running ABC service for its composite states in turn; creates/changes an activity and sends it to the controller. For activities where two or more participants are active, the state manager is notified on every component state change (delta state changes); this delta state is forwarded to the activity server that broadcasts it to every participant's device. Upon reception of a delta state, the state manager immediately notified the relevant ABC service that is then responsible for interpreting and adjusting accordingly.

- *ABC Service* provides domain specific functionality and must enable state object packing (for suspend) and parsing (for resume). Collaborative, graphical, services can utilize the collaborative Swing components as described in the previous section.

- *Session Manager* is responsible for the collaborative tools like telepointers and audio link. As performance are more important than lost information, session managers communicate directly using UDP as shown in the deployment view in figure 7. The session manager also registers itself as listener on the running activity's session object and is thus notified when participants join or leave the activity; this information is visualized in the collaboration frame, marked as (2) in figure 4.

5 Discussion

As outlined above, collaborative information to be shared takes two forms in ABC: delta state changes in ABC services and dataflows in collaborative tools like voice link and telepointers. The two types are treated differently in the architecture as service state changes are broadcast through the activity server using an RMI protocol while tool state changes are broadcast directly between clients using and UDP protocol. Why two mechanisms?

There are several reasons for that. First, the semantics is different. Activity changes must be guarantied to be broadcast hence the more restrictive protocol; a few lost packages on telepointer activity is in contrast not very critical. This warrents treating the data differently. Second, it is important that the server side activity manager is kept up-to-date with the precise state information within an activity and this is most easily accomplished by having the activity delta state go through the server. This makes handling late-comers easy as they just resume the activity as it is defined by the server—by defintion this is the activity in its most recent form.

We have implemented the architecture discussed above in the ABC Framework [1], which currently exists in a third version. We have evaluated this framework and the ABC principles in a number of workshops by scenario-based role-playing of typical work situations at a hospital. A wide range of nurses and physicians have participated in
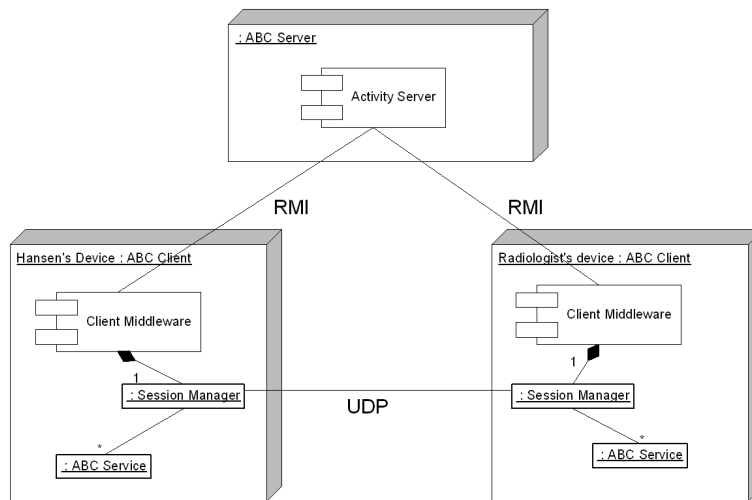
6

Figure 7. Deployment view

these workshops. One of the scenarios that we have evaluated was outlined in section 3.1. We video recorded the workshops and interviewed the clinicians afterwards.

While is it difficult to get quantitative data from this kind of workshop evaluations, they still provide qualitative insight [2]. An important observation was how collaboration was initiated between clinicians in a natural way. The radiology scenario was role-played and the clinicians appreciated this alternative to face-to-face meetings. We did not experience any major problems or questions regarding establishing the connection between the participants. The main issue raised concerned the issue of either participating in an activity or not. The users was requesting a way to maintain some peripheral awareness about activities in which they participate in, but which was not they were currently working with. For example, the audio link is established between participants in the same activity. Thus if a participant switch to another activity and back again, the audio link is temporarily lost. This took some time getting used to.

Several issues need more attention. Ideally, activities define the set of services they require, and each service may be realized by many different applications. For instance, an X-ray image viewer service is realized by one application on a laptop and another on a PDA. However, introducing fine-grained state descriptions easily leads into state descriptions that are device specific in order to render complex GUIs like medicine schemas or dialog boxes. Our introduction of the Java Swing mirror library is a great help for de-

velopers in defining activity collaboration between devices that may run Java Swing but is problematic for devices that does not—indeed there is a tendency for developers of collaborative activity aware services to focus only on Swing. One may argue that the proposal is sound at the architectural level—developers just have to make the component state descriptions abstract—and what remains is "just" HCI issues of defining concrete applications for ABC services for each of the device types used. In practice, however, we think that the problem is more complex, as the user interface and architectural issues cannot be decoupled so easily. GUIs on a laptop, PDA, and mobile phone are often different at a very fundamental level in order to make them usable, and defining a single abstract description to embody all may be unfeasible.

6 Related Work

The work of the ABC framework clearly related to the work in ubiquitous environments, as discussed in the introduction. Our concept of activity based computing resembles the task-driven computing concept in Aura [10, 16]. But the ABC framework has a greater focus on local mobility within a work setting and not remote mobility as discussed in Aura. Furthermore, the ABC framework is inherently designed to support collaboration – asynchronous as well as synchronous – both of which are absent in the Aura project.

Looking at related work in the research field of CSCW there is a range of technical research that relates to the implementation of the ABC framework. First, the real-time collaboration features of the ABC framework relate to the work on application sharing systems. Such systems can in general adopt either a centralized or a replicated architecture. Different architectures have different tradeoffs, ad-

---

[2] As pointed out by [3] it is difficult to use formative and summative methods in the evaluation of ubicomp systems: "In order to understand the impact of ubiquitous computing on everyday life, we navigate a delicate balance between prediction of how novel technologies will serve a real human need and observation of authentic use and subsequent coevolution of human activities and novel technologies [7, p. 46]."

vantages, and application domain [6, 13]. In a centralized system, only one host runs the shared application and the input/output events are distributed among all participating clients. This has been a natural choice of architecture in the groupware systems on the X Windows platform because X Windows defines a network-aware graphics protocol that separates an application's display (the X server) from its computation (the X client). This separation yields a natural approach to implementing application sharing by having one centralized X client with multiple X servers on different host machines. Examples of systems are SharedX [9] and XTV [2]. But also on the Windows platform, NetMeeting applies a centralized application sharing architecture. In a replicated architecture the same application and its execution environment are replicated at each participating host machine. The ABC framework implements a replicated architecture where the client side processes and the ABC applications are executing on the local host machines also during real-time activity sharing.

The main advantage of the centralized architecture is its simplicity. However, one of the disadvantages of a centralized architecture for application sharing is the need for a floor control mechanism because the application (running in only one copy) is unable to handle simultaneous input. Another is the lack of supporting heterogeneous computing environments because only one application is running in a specific execution environment (see however [13] for an exception). We have chosen the replicated architecture because replicated applications sharing has faster local responses, lower network bandwidth requirements, and has more potential in supporting concurrent work eliminating the need for floor control. In our design of the collaborative sharing of activities we used NetMeeting as a prototype in a large workshop held with clinicians from the hospital and one of the conclusions from this workshop was to avoid floor-control mechanisms. Furthermore, a replicated architecture is better suited to handle the contingencies in a ubiquitous and heterogeneous computing environment. For example, if the network connection disappears the consequence is that the ABC infrastructure with it state and collaboration management fall out. However, the distributed applications running on the client hosts continue to execute.

The traditional problems of maintaining consistency among replicas of the same application and of accommodating latecomers in a replicated architecture [12, 13] have proven to be relatively easy to handle within the ABC framework. This is caused by the invariant of an activity object, which ensures that the activity manager (the server) always has the shared state of the collaborative session, and latecomers just obtain this when activating an activity.

A related characteristic of synchronous collaborative applications is to what degree the application itself is made collaborative by having access to the source code [12]. Col-laboration aware application are specifically designed for simultaneous use by multiple users, whereas collaboration transparent application are shared by collaboration-aware mechanisms that are outside and unknown (i.e. the word 'transparent') to the application and its developers. Researchers following the collaboration transparent strategy have a very strong argument when they say that the assumption of creating collaboration aware version of already wellaccepted single-user application (e.g. Microsoft Office) is highly dubious. From a user point of view, who would abandon their favorite word processor to use a co-authorship application [11] and from a technical point of view, the assumption of having access to proprietary source code for modification is in practice impossible for any real-world kind of applications. The current implementation of the ABC framework supports the development of collaboration aware applications. This decision was made because the ABC framework is a vehicle for investigating various research questions within ubiquitous computing and CSCW. However, we plan to integrate existing applications running in different environments by making ABC application wrappers to e.g. programs running on a Windows platform. We are currently investigation how to do this by hooking into the Windows API and making a general ABC wrapper to be used for all Window based applications. However, this approach is basically in conflict with our cross-platform, cross-environment strategy of decoupling services from applications, because we want to make the state management work on other platforms than Windows.

Finally, as for the key idea in the ABC framework of collecting services ('what users are doing') in an activity, there are similarities with other more user-interface related research. Starting with Rooms [5], and continuing through recent system, such as the Task Gallery for Windows [15] and the Kimura system [14], research have been conducted to design ways of handle multitasking in window-based user-interfaces. These 'virtual desktop' approaches treat tasks as a cohesive collection of applications. When a user refers to a particular task, the system automatically brings up all the applications and documents associated with that task. This relieves the users from launching and arranging applications and documents individually. In our work, we extend this notion by modeling an activity as a collection of abstract services decoupled from application that can handle such services. This decoupling paired with the distributed nature of activities allows an activity to be handed over to and instantiated in different environments using different supporting applications running of different hosts, and by different users who participate in the activity.

7 Conclusion

We have in this paper outlined the basic idea of activity based computing and argued why this paradigm has potential in a pervasive computing environment, especially for work contexts characterized by mobility, frequent interruptions, ad-hoc use of multiple devices, and collaboration. The background for this research is clinical work in hospitals, which is a domain that provides a nice test-bed for research into computing infrastructures for ubiquitous computing. In this domain, collaboration is the rule rather than the exception as many highly specialized professions must meet in order to determine the proper treatment of a single patient.

So far, activity based or task level computing infrastructures have not touched upon the collaboration issue nor disussed the conceptual or architectural implications. Our main contribution is to open this debate in several aspects. First, we have experienced how the activity concept is indeed useful for users to structure collaborative behaviour. An activity or task is the natural focal point of collaboration and is a natural human level concept for structuring collaborative aspects such as defining what the collaborative effort is about and defining the persons and roles involved. As the system understands "activity" as first class object at the architectural level, many issues relating to session management is handled elegantly. Second, the computational activity becomes the natural architectural component to manage collaborative dataflows and define protocols.

Our second contribution is to describe a proposal for how activity based architectures may support collaboration; and outlined an architecture for it. We have argued how the activity based paradigm at the logical/conceptual level can support collaboration. However, as argued above additional refinement as well as archicutral components are necessary in order to achieve proper performance.

While we cannot claim to have a detailed and in-depth quantiative evaluation of the architecture, we never-the-less have important qualitative data from our workshop evaluations. Our experience from these is that the activity concept indeed feels natural to clinicians for organizing their collaborative efforts when using a computing system. The proposed architecture was buildable and performed adequately with a limited number of participants on an activity.

Acknowledgements

References

[1] The ABC Framework. http://www.pervasive.dk/abc.

[2] H. Abdel-Wahab and M. Feit. XTV: A framework for sharing X window client in remote synchronous collaboration. In *Proceedings of IEEE Tricomm'91*, pages 159–167, Chapel Hill, NC, USA, Apr. 1991.

[3] G. D. Abowd and E. D. Mynatt. Charting Past, Present, and Future Research in Ubiquitous Computing. *ACM Transactions on Computer-Human Interaction (ToCHI)*, 7(1):29–58, 2000.

[4] Project aura. http://www-2.cs.cmu.edu/~aura/.

[5] 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.

[6] J. Begole, M. B. Rosson, and C. A. Shaffer. Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 6(2):95–132, 1999.

[7] J. M. Carroll and M. B. Rosson. Deliberated evolution: Stalking the view matcher in design space articles. *Human-Computer Interaction*, 6(3,4):281–318, 1991.

[8] H. B. Christensen and J. E. Bardram. Supporting human activities – exploring activity-centered computing. In G. Borriello and L. E. Holmquist, editors, *Proceedings of Ubicomp 2002: Ubiquitous Computing*, volume 2498 of *Lecture Notes in Computer Science*, pages 107–116, Göteborg, Sweden, Sept. 2002. Springer Verlag.

[9] D. Garfinkel, B. Welti, and T. Yip. HP SharedX: A tool for real-time collaboration. *Hewlett-Packard Journal*, 45(4):23–36, Apr. 1994.

[10] D. Garlan, D. P. Siewiorek, A. Smailagic, and P. Steenkiste. Project Aura: Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31, Apr. 2002.

[11] J. Grudin. Groupware and social dynamics: eight challenges for developers. *Communications of the ACM*, 37(1):92–105, 1994.

[12] J. C. Lauwers and K. A. Lantz. Collaboration awareness in support of collaboration transparency: requirements for the next generation of shared window systems. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 303–311. ACM Press, 1990.

[13] D. Li and R. Li. Transparent Sharing and Interoperation of Heterogeneous Single-User Applications. In E. F. Churchill, J. McCarthy, C. Neuwirth, and T. Rodden, editors, *Proceedings of the 2002 ACM conference on Computer Supported Cooperative Work*, pages 246–255. ACM Press, 2002.

[14] B. MacIntyre, E. D. Mynatt, S. Vodia, K. M. Hansen, J. Tullio, and G. M. 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.

[15] 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 *Proceedings*

of the SIGCHI conference on Human factors in computing systems, pages 494–501. ACM Press, 2000.

[16] 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.