Design of the ABC Framework
Version 4

Jakob Bardram, Jonathan Bunde-Pedersen, Martin Mogensen

July 28, 2005
Contents

1 Introduction 4
  1.1 Goals of the Activity-Based Computing Platform . . . . . . . . . . . 5
  1.2 Environmental Assumptions . . . . . . . . . . . . . . . . . . . . 5

2 System Overview 6
  2.1 Key Concepts . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
    2.1.1 Activity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
    2.1.2 Services . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
    2.1.3 Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  2.2 Key Features . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
    2.2.1 Activity Suspend and Resume . . . . . . . . . . . . . . . . 7
    2.2.2 Activity Roaming . . . . . . . . . . . . . . . . . . . . . . . 7
    2.2.3 Activity Adaptation . . . . . . . . . . . . . . . . . . . . . . 7
    2.2.4 Activity-Based Service Discovery . . . . . . . . . . . . . . 8
    2.2.5 Activity Sharing . . . . . . . . . . . . . . . . . . . . . . . . 8
  2.3 ABC Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
    2.3.1 Software Qualities . . . . . . . . . . . . . . . . . . . . . . . 10

3 ABC Protocol and AML 11
  3.1 ABC Protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
    3.1.1 ABCP Request Interface . . . . . . . . . . . . . . . . . . . . 11
    3.1.2 ABCP Publish-Subscribe Interface . . . . . . . . . . . . . . 13
    3.1.3 ABCP Response Interface . . . . . . . . . . . . . . . . . . . . 14
  3.2 Activity Markup Language . . . . . . . . . . . . . . . . . . . . . . 14
    3.2.1 All Users . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
    3.2.2 One User . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
    3.2.3 Activity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
    3.2.4 Activity Proxy . . . . . . . . . . . . . . . . . . . . . . . . . 16
    3.2.5 Post and Append . . . . . . . . . . . . . . . . . . . . . . . . 16
  3.3 Example of ABCP + AML for ’state_changed’ event . . . . . . . . . 16

4 Infrastructure Layer 18

5 Client Layer 19
  5.1 The ActivityController . . . . . . . . . . . . . . . . . . . . . . . 19
    5.1.1 Resuming an activity . . . . . . . . . . . . . . . . . . . . . 19
    5.1.2 Suspending an activity . . . . . . . . . . . . . . . . . . . . 20
  5.2 The StateManagers . . . . . . . . . . . . . . . . . . . . . . . . . . 21
5.3 Events ......................................................... 22

6 User-Interface Layer ........................................ 23
   6.1 The graphical user interface ............................. 23
   6.2 Zoomable activities ..................................... 24
   6.3 System Hooks ............................................ 25
   6.4 Monitoring applications ................................. 25

7 Activity Sharing and Collaboration ....................... 26
   7.1 Collaboration Management ............................... 26
   7.2 State changes ........................................... 27
   7.3 Telepointers .............................................. 30
   7.4 Voice Links .............................................. 31

8 Applications ................................................. 33
   8.1 External .NET Applications .............................. 33
   8.2 Internal Wrapped Native Applications .................. 34
   8.3 Internal Wrapped Unknown Type Applications .......... 35

9 Application Programming Interface ....................... 36
   9.1 Services provided by the framework .................... 36
      9.1.1 State ............................................... 36
      9.1.2 Collaboration ...................................... 37
      9.1.3 Intra-activity communication ....................... 37
   9.2 Application requirements ............................... 37
      9.2.1 Annotations ........................................ 37
      9.2.2 Overriding the default behaviour ............... 38
   9.3 Example application ................................... 38

10 Legacy Applications ....................................... 41
   10.1 Overview of approaches ............................... 41
   10.2 Collaboration ......................................... 43

11 Related Work .............................................. 44
List of Figures

2.1 The Activity structure. .................................................. 6
2.2 The ABC architecture. .................................................. 9

5.1 The ActivityController and relations. .............................. 19
5.2 The flow of events. ...................................................... 22

6.1 The ActivityBar. (1) ActivityProxyList showing all activities. (2) Actions on activities. (3) ActivityButtons. (4) Shows status information. (5) Event-viewer. .................................................. 23
6.2 The zoomable UI. (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. .................................................. 24

7.1 The ABC hybrid architecture. ........................................... 27
7.2 The simples state event propagation situation. ...................... 28
7.3 A subtle state event propagation situation. .......................... 29
7.4 Client threading problem ............................................... 30
7.5 Client side currency control algorithm in pseudo code .......... 32

9.1 Annotating a field ....................................................... 38
9.2 Overriding default behaviors ........................................... 38
9.3 The ABCChat program. (1) The shared ‘chat’ textbox. (2) The private textbox writing messages. (3) The statusesbar showing if someone is writing a message. .................................................. 39
9.4 Overriding default behaviors ........................................... 40
# Document Management

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Init.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4/4-2005</td>
<td>JB</td>
<td>Initial version</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Chapter 1 – Introduction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Chapter 2 – System Overview</td>
</tr>
<tr>
<td>1.1</td>
<td>11/4-2005</td>
<td>JB</td>
<td>Chapter 3 – ABCP protocol and AML</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Chapter 5 – Client layer</td>
</tr>
<tr>
<td>1.2</td>
<td>12/4-2005</td>
<td>JBP</td>
<td>Chapter 6 – UI Layer</td>
</tr>
<tr>
<td>1.3</td>
<td>13/4-2005</td>
<td>JBP</td>
<td>Chapter 7 – Collaboration</td>
</tr>
<tr>
<td>1.4</td>
<td>18/4-2005</td>
<td>JBP</td>
<td>Chapter 8 – Applications</td>
</tr>
<tr>
<td>1.5</td>
<td>25/4-2005</td>
<td>MM</td>
<td>Chapter 9 – API</td>
</tr>
<tr>
<td>1.6</td>
<td>25/4-2005</td>
<td>MM</td>
<td>Chapter 10 – Collaboration</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This technical report describes version 4 of the ABC Framework. The ABC Framework is designed to support Activity-Based Computing (ABC). ABC will not be presented in this report – interested readers are referred to the various publications on ABC [? , ? , ? , ?]. The purpose of this document is to describe the technical design and implementation of the ABC Framework, version 4. For reference, below is a short resumé of the differences between the versions:

<table>
<thead>
<tr>
<th>Version</th>
<th>Impl.</th>
<th>Dist.</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1       | Java  | Java RMI | Activity Discovery  
          |        |          | Activity as an event |
| 2       | Java  | Java RMI | Activity as an object  
          |        |          | Server, management, storage  
          |        |          | Activity as a singleton moving from the server to the client when in use. |
| 3       | Java  | Java RMI | Activity Sharing  
          |        |          | Activity as session mgmt.  
          |        |          | New ABC client (bar, telepointers, voicelink)  
          |        |          | Java-based API for development of ABC applications, including distributed observers and collaborative Swing widgets. |
| 4       | Java  | ABCP    | Language and platform independent  
          | C#     | AML     | Robust vrt. network failure  
          |        |          | Runs with and without server  
          |        |          | Integration with Windows XP  
          |        |          | Wraps some native applications  
          |        |          | New ABC client (UI, Zoom, collaboration)  
          |        |          | C#-based API for development of ABC applications |

This document describes the overall goal of the ABC Framework (section 1.1);
describes the high level architecture of the ABC software system (chapter ??), the dis-
tribution mechanisms in terms of the ABC Protocol and the Activity Markup Language
(AML) (chapter 3); the Activity Server (chapter ??); the Activity Client (chapter ??);
how collaborative activities are handled (chapter 7); the various applications that we
currently support or have developed (chapter 8); and the programming interface (API)
(chapter 9). Finally, chapter 11 contains some references to related work and describes
how this work is different.

1.1 Goals of the Activity-Based Computing Platform

1.2 Environmental Assumptions
Chapter 2

System Overview

2.1 Key Concepts

Activity-Based Computing is divided into three levels of support illustrated in figure 2.1.

![Activity structure diagram](image)

Figure 2.1: The Activity structure.

2.1.1 Activity

The most important concept within the ABC architecture is that of an activity. An activity is an entity that represents human work activity and contains information about what computational services are supporting this activity. This includes:

1. The purpose or the goal of the activity. Currently this is implemented as a name and a description of the activity.

2. The participants who is engaged in this activity, including information about which of these participants own the activity.

3. The services which are involved in this activity and support the human users of the activity.

4. The state of the activity, including the state of each service in the activity. A service state may include data itself or pointers to data on e.g. a server.
2.1.2 Services

A service is a representation of a computational service which supports part of the activity. Services are abstract representations of concrete services. Concrete services may be applications, like MS Word or Mozilla Thunderbird, or it may be e.g. web services, which are accessible from a web browser. The same service may use different concrete services when activated. For example, both the Mozilla Firefox and the MS Internet Explorer are concrete services which implement the abstract ‘internet browser’ service.

2.1.3 Data

Each service may access data. The ABC framework does not specify how this is done and we rely on other data distribution mechanisms for services to access their data. Data may be stored locally on a disk and hence only local service may access this data; it may be residing on a server, thereby enabling client-server applications to access it; or it may stored as part of the abstract service state description and distributed via the ABC platform.

2.2 Key Features

In order to support the goals of being an runtime infrastructure for pervasive computing with special focus on mobility, collaboration, interruptions, adaptation, and context-awareness the ABC platform has a range of key features.

2.2.1 Activity Suspend and Resume

The main feature in activity-based computing is the support for activity suspending and resuming. This feature supports interruptions in work since a user can easily alternate between parallel activities.

2.2.2 Activity Roaming

Activities are stored and distributed via the infrastructure. An activity can therefore be suspended on one device and resumed on another. This feature supports mobility since it allows users to roam between fixed devices. This feature also supports the user to apply appropriate devices in different situations thereby allowing users to effortlessly move his or her work from one device to another, like from a desktop PC and onto a tablet PC before commencing the ward round.

2.2.3 Activity Adaptation

Activities can be suspended and resumed on heterogeneous devices. For example, an activity may be suspended on a desktop PC in the radiologist’s office, resumed on his PDA on the way to the conference room, and in the conference room resumed on the large wall-sized display. An activity adapt itself to the available concrete service on a given device. Hence, the de-coupling between an service and a concrete service (application) helps services to run locally. This feature support the use of heterogeneous devices.
2.2.4 Activity-Based Service Discovery

When an activity is resumed on a local device, it needs to look around for concrete services (application) that can help execute its abstract services. This is done by local service discovery on the device. The devices may use remote services for delegating work. For example, a radiology viewing service may actually run on another device that the one the activity is resumed on.

2.2.5 Activity Sharing

An activity is not personal but is shared amongst its participants. This feature enables participants to take turns in working on an activity by resuming it. The state of the activity is constantly changed and users will hence see the results of the other participants’ work. Furthermore, if two or more participants resume the activity simultaneously on different devices they will engage in real-time activity sharing, i.e. they will see each other work on the activity, see the state changes in the services, and online collaborative widgets like telepointers and a voice link will be established. This feature supports the user to engage in collaborative work as a natural part of using computers – not as a special application outside the basic platform.

It should be noted that the activity concepts does not entail any notion of workflow. Hence, the activity does not describe how is has to be done nor does it specify which activity is to be done before, after, or in parallel with it.

2.3 ABC Architecture

The ABC architecture is illustrated in figure 2.2. The architecture consists of an infrastructure layer, a client layer, and a User-Interface (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 Activity Server and its components are further described in chapter 4.

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 client layer and its components are further described in chapter 5.

The UI layer resides on top of the ABC framework. This layer is specially designed and implemented according to different devices, like a standard PC, a tablet PC, a wall-sized display, a PDA, or a wearable client with a headmounted display. Implemented user-interface components in version 4 includes:

- The Activity Bar which is a bar similar to the Windows Taskbar for handling activities. It provides access to a user’s list of activities, and buttons for creating, resuming, suspending, and saving activities. It also provides access to the collaborative parts of the ABC client, like the list of collaborators, a simple chat, telepointers, voice links, and the activity server.
• A speech interface that 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.

• A focus+context zoomable interface for zooming in and out in an activity. This helps the user to toggle between an overview of an activity and focusing on details within an activity. This is done by zooming the interface. If, for example, an activity is moved from a wall-sized display to a smaller laptop, the zoomable interface helps the user to get an overview of the whole activity and zoom in on specific services (i.e. running applications).

• A set of collaborative widgets used in real-time activity sharing. This including telepointers for remote gesturing, a peer-to-peer voice link between participating clients, a simple chat application, and a list of participants of the activity, including their contextual information as revealed by the context service.

Chapter 6 describes these components in the user-interface layer as integrated into the Windows XP operating system in greater details.

The communication between the Activity Server and the Activity Client (the controller) is done using the Activity-Based Computing Protocol (ABCP) using the Activity-Based Computing Markup Language (AML) described in chapter 3.
2.3.1 Software Qualities

The ABC architecture is designed to meet some basic architectural qualities, including:

1. **Loose coupling** – the ABCP protocol is stateless, like the HTTP protocol. Hence, in a mobile and pervasive computing environment where clients enters and leaves network infrastructures, the ABCP protocol enables clients to enter and leave this network. Activity-based sessions can be resumed independent of network disconnections between the suspend and resume.

2. **Scalability** – the ABC architecture is a hybrid between a client-server and a peer-to-peer architecture. This is done for scalability reasons. The client-server architecture is used for providing central access to shared resources, like activities and context information and for resolving parallel state updates from multiple clients in real time. Only minimum network communication is used between clients and the servers. Heavy network traffic for telepointers and voice over IP is done peer-to-peer between hosts participating in a real-time activity sharing session. This is done using unreliable communication (UDP) because these collaborative information is stateless.

3. **Heterogeneous devices** – the ABCP protocol and the AML specification is designed to support heterogeneous devices, including heterogeneous operating systems, programming languages, and available resources.

4. **Contingency Management** – still missing...

5.
Chapter 3

ABC Protocol and AML

Communication between the infrastructure layer (the Activity Server) and the client layer (the Activity Controller) is done using the stateless ABC Protocol – ABCP. The ABCP protocol includes both a simple request-response api like the HTTP protocol as well as a subscribe-publish-notify api for subscribing to events at the server. A special XML-based Activity Markup Language (AML) is designed to model activities and activity events. Communication using AML over ABCP makes the ABC infrastructure language and operating system independent and enables it to scale to the whole internet – similar to the HTTP protocol.

3.1 ABC Protocol

3.1.1 ABCP Request Interface

The syntax of an ABCP request is:

ABCP/<version> <method> /<argument>?{<key>=<value>&)*

Where

<version> ::= 1.0

<method> ::= GET | POST | DELETE | CREATE | REGISTER |
UNREGISTER | RESUME | SUSPEND | LOGIN |
LOGOUT | PUBLISH | SUBSCRIBE | UNSUBSCRIBE |
NOTIFY

<argument> ::= all_users | user | activity | proxy_activity |
last_activity| resource | participant |
service | component |

Examples of ABCP request for the Activity Store and the Activity Manager is shown in table ??.
Store

ABCP/1.0 GET /all_users
  returns all users
ABCP/1.0 GET /user?id=<user_id>
  returns AML data for specified user
ABCP/1.0 GET /activity?id=<activity_id>
  returns AML data for specified activity
ABCP/1.0 GET /proxy_activity?id=<activity_id>
  returns AML data for activity proxy for specified activity
ABCP/1.0 GET /proxy_activity?user=<user_id>
  returns AML data for activity proxies for specified user
ABCP/1.0 GET /last_activity?user=<user_id>
  returns AML data for last activity for specified user
ABCP/1.0 POST /activity?id=<activity_id>
  appends the AML data to the specified activity on the server
ABCP/1.0 DELETE /activity?id=<activity_id>
  deletes specified activity, returns nothing
ABCP/1.0 DELETE /participant?activity=<activity_id>&id=<person_id>
  deletes specified participant, returns nothing
ABCP/1.0 DELETE /service?activity=<activity_id>&id=<service_id>
  deletes specified service with state information, returns nothing
ABCP/1.0 CREATE /activity?user=<user_id>
  creates new activity with specified users as creator, returns the activity

Manager

ABCP/1.0 REGISTER /host?name=<host_name>&device=<device_type>
  registers an activitybar at the manager with the specified device type
ABCP/1.0 UNREGISTER /host?name=<host_name>
  un-registers an activitybar at the manager
ABCP/1.0 RESUME /activity?id=<activity_id>&user=<user_id>&host=<host_name>
  activates and returns activity
ABCP/1.0 SUSPEND /activity?id=<activity_id>&user=<user_id>&host=<host_name>
  deactivates the activity, returns nothing
ABCP/1.0 LOGIN /user?id=<user_id>&host=<host_name>&place=<place_id>
  log in user at specified host and place
ABCP/1.0 LOGOUT /user?id=<user_id>&host=<host_name>&place=<place_id>
  logs out user at specified host and place
Where,

\[<device\_type> ::= <platform>+<net>+<display>\]
\[<platform> ::= tiny | small | medium | large | huge\]
\[<net> ::= 0.. MB/s\]
\[<display> ::= none | tiny | small | medium | large | huge\]

---

**Resources**

ABCP/1.0 GET /resource?activity=<activity_id>&id=<resource_id>

returns a resource with the specified name, associated with the specified activity. Hash = name.

ABCP/1.0 POST /resource?activity=<activity_id>&id=<resource_id>

post a resource with the specified name and associate it with the specified activity.

ABCP/1.0 DELETE /resource?activity=<activity_id>&id=<resource_id>

deletes specified resource.

---

### 3.1.2 ABCP Publish-Subscribe Interface

The syntax of an ABCP Subscribe-Publish-Notify interface is:

ABCP/1.0 SUBSCRIBE /host?name=<host_name>&port=<port_number>&activity=<activity_id>&scope=<scope>

subscribes to listen to activity changes.

ABCP/1.0 UNSUBSCRIBE /host?name=<host_name>&port=<port_number>&activity=<activity_id>

unsubscribes to listen.

ABCP/1.0 PUBLISH /activity?id=<activity_id>&arg=<argument>

publish an event about changes to the specified activity. contains content=AML for the change to be published.

ABCP/1.0 NOTIFY /activity?id=<activity_id>&arg=<argument>

receives an event that the specified activity has changed.

Where,

\[<scope> ::= activity | meta | state\]
\[<argument> ::= activity\_changed | activity\_created | activity\_resumed | activity\_suspended | activity\_deleted | participant\_added |\]
All these published AML data will be stored on the ABC server, except event of type ‘event’, which are for general stateless collaboration events.

3.1.3 ABPC Response Interface

The syntax of an ABPC response is:

ABCP/1.0 <<status_code>> <<response_message>>
Content-Length: <<content_length>>
Content-Type: text/xml

<activity id="..."
  ...
</activity>

where «status_code», «response_message» and «content_length» equals the HTTP protocol. Note the two carriage return before the content (AML) of the response is listed. This is equal to the HTTP protocol as well.

3.2 Activity Markup Language

3.2.1 All Users

<?xml version="1.0" encoding="ISO-8859-1" ?>
<users>
  <person id="bardram@daimi.au.dk">
    <name>Jakob Bardram</name>
    <context>
      <location>Hopper.333</location>
      <active_activity id="1">Activity no. 1</active_activity>
    </context>
  </person>
  ...
</users>

3.2.2 One User

<?xml version="1.0" encoding="ISO-8859-1" ?>
<person id="bardram@daimi.au.dk">
  <name>Jakob Bardram</name>
  <context>
  ...
</person>
3.2.3 Activity

<?xml version="1.0" encoding="ISO-8859-1" ?>
<activity id="88" name="iouoiu" status="88" type="44" creator="uuj">
    <meta>
        <description>
            æladksfæslsdkf
        </description>
        <participants>
            <person id="23">
            ...
            </person>
        </participants>
        <resources>
            <resource activity="88" name="screen_dump" type="image/jpeg" length="1024">
                ...
            </resource>
        </resources>
    </meta>
    <state activity="88">
        <service id="345" type="browser">
            <component id="444">
                <location url="http://www.dr.dk/" bookmark="#editor" />
            </component>
            <component id="445">
                <frame x="10" y="30" h="200" w="300" />
            </component>
            ...
        </service>
        <service id="858" type="browser">
            <component id="444">
                <location url="http://www.dr.dk/" bookmark="#programs" />
            </component>
            <component id="445">
                <frame x="10" y="30" h="200" w="300" />
            </component>
            ...
        </service>
        <service id="235" type="editor">
            <component id="434">
                <document location="http://www.daimi.au.dk/hi.doc" bookmark="#programs" />
            </component>
            <component id="435">
                <frame x="10" y="30" h="200" w="300" />
            </component>
        </service>
    </state>
</activity>
3.2.4 Activity Proxy

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<activity_proxies>
  <activity_proxy id="88" name="iouoiu" status="88" type="44" creator="uuj">
    <description>
      æladksfæsldkf
    </description>
    <resources>
      <resource activity="88" name="screen_dump" type="image/jpeg" length="1024"/>
    </resources>
  </activity_proxy>
  ...
</activity_proxies>
```

3.2.5 Post and Append

When posting or adding information to an activity, the `Content-Length` and the `Content-Type` header lines are required. For example:

```
ABCP/1.0 POST /activity?id=12
Content-Length: 126
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1" ?>
<activity id="12" name="iouoiu" status="88" type="44" creator="uuj">
  <description>
    æladksfæsldkf
  </description>
</activity>
```

3.3 Example of ABCP + AML for ‘state_changed’ event

```
ABCP/1.0 PUBLISH /activity?id=88&arg=state_changed
Content-Length: 126
Content-Type: text/xml

<?xml version="1.0" encoding="ISO-8859-1" ?>
<activity id="88" senderid="10.11.34.102">
  <state activity="88"/>
</activity>
```
<service id="345">
  <component id="url">
    http://www.dr.dk/
  </component>
</service>
</state>
</activity>
Chapter 4

Infrastructure Layer

Server by JB
Chapter 5

Client Layer

The client layer provides the user with an interface. It provides access to the storage- and event-handling facilities of the backend server. Furthermore it integrates itself with the OS and provides methods for controlling and monitoring services which are part of activities.

This chapter describes these capabilities and gives an overview of the interplay that occurs between them when using the system.

5.1 The ActivityController

The activity controller [AC] is the hub of the client. It ties three other main components of the system together. The first component is the Server (or LocalCache) which handle storing/retrieval of activities. The second is the collection of StateManagers which each manage the finer details such as monitoring and controlling of applications in a single activity. The last component is the GUI which can receive events and control the current state of the system.

![Figure 5.1: The ActivityController and relations.](image)
The **Server** is reached via a connection which is initiated by the AM. This connection is used for storing and retrieving AML-data. The client also maintains threads which listens for events originating from the server and posts events (for instance when an application changes state) to the server as well. If the connection at some point fails the LocalCache steps in and intervenes the data flow, and the AC itself continues functioning without any changes.\(^1\)

The **GUI** has access to events from the system and handles for controlling activities through the AC. The events which is available are as follows:

- **ActivityEvent** constitutes an event like ‘Activity x has been resumed’.
- **GenericEvent** which can be anything (just a string encapsulator) and thrown at anytime (even before the AC itself is initialized). An example could be a missing .config file or if the system is unable to connect to a server initially.
- **ConnectionEvent** is an event thrown whenever the LocalCache takes over (connection lost) or a connection is reestablished.
- **InternalError** is an exceptional event, which allows the AC (and UI) to respond asynchronously to exceptions occurring anywhere in the system (including another thread).

The AC also provides functions for controlling the state of any current activity. This is done via the public methods of the AC such as `resumeActivity(string activity_id)` and `suspendActivity(string activity_id)`.\(^2\)

The **StateManagers** are initialized and controlled by the AC. While the AC handle the overall control of an activity, the details of monitoring the individual services (applications) which comprise the activity is handled by a StateManager. These details are things like knowing which applications correspond to which services and basically maintaining the state of an actual application such that it mirrors the information in the matching AML-data. Originally there was a one-to-one relationship between the AC and a StateManager but by allowing several StateManagers to be active at once we can both optimize the switching of activities and also run several activities simultaneously (on a wall-sized display for instance).

In the following two sections we will briefly describe what happens throughout the system when; firstly an activity is resumed, secondly when it is once more suspended.

### 5.1.1 Resuming an activity

The GUI component of the system must be tailored to either run a single activity or to run multiple activities. The AC itself does not make any assumptions about this and does not have a notion of a ‘current activity’. Therefore the GUI itself must maintain this information, and in the following we assume that there is only one concurrent activity (as far as the GUI knows) which is the case with the current version of our system\(^3\). So, the first thing that happens whenever an activity is resumed is that the current activity (if any) is suspended (see section 5.1.2). Then a number of actions takes place:

---

\(^1\) An event is thrown through the AC when the client goes offline/online  
\(^2\) For a complete list see the ABC Client API TODO  
\(^3\) ActivityBar project
1. The AML-data describing the activity is loaded, either from a file or from the Server (or LocalCache if the system is currently offline).

2. If an inactive (see section 5.1.2) statemanager for the activity already exists, it is just activated. Otherwise a StateManager is created (or an old one is reused) and is given the activity (data) to resume.

3. The StateManager restores the activity (see section 5.2).

4. The Server is told that the activity has been resumed and the AC registers listeners for the Server-events. An ActivityEvent is generated.

5.1.2 Suspending an activity

The process of suspending an activity exists in two variants. The first simply closes all applications, and the second keeps them running in the background. There are advantages and disadvantages to each approach. We have in the ActivityBar implemented a hybrid approach, which means that we use the optimized version up with to 10 activities and then via a fifo-queue begin actually shutting down the activities. The basic steps are as follows:

1. The AC asks the StateManager for the activity to update the AML with the current state of the activity. The StateManager queries all applications for their state and returns it to the AC.

2. A Request to store the AML for the activity is assembled and sent to the Server by the AC.

3. Prior to the actual ‘closing’ of the applications the ApplicationMonitor is disabled. Then one of two things happen; either the StateManager is then asked to suspend the activity which means it must end all applications and reset itself, or it is asked to mark itself as InActive. If the latter is done then the StateManager simply hides all its applications – but keeps running them. This is done as an optimization because starting and application is extremely slow compared to hiding it. The main disadvantage is of course that the resources used by the applications cannot be freed.

4. The Server is told that the activity has been suspended, and an ActivityEvent is generated.

5.2 The StateManagers

A StateManager handles the individual applications which are part of an activity. The StateManager will update the state when an update-event arrives from the server and will return updated state-info when requested by the ServerEventPublisher. It also handles starting and shutting down of applications.

The resuming of an activity seen from a StateManager consists of determining which applications to start (by querying the registry) and then starting a thread for each. When all threads have finished (or timed out) the StateManager will register itself as a handler for events concerning any of the applications with the ApplicationMonitor. It will then be informed about changes in the application, e.g. if the application quits.

---

*The ServerEventPublisher will detect state-changes and publish them to the Server for distribution*
5.3 Events

The AC and the CollaborationController are from the point of view of the GUI the only source of events. But they themselves are not the main event-generators, instead they are pipes for the events coming from all parts of the system. This makes it conceptually simpler to find and react to events in the GUI, and also provides a single place for controlling the flow of events. The different types of events are listed in section 5.1 and figure 5.2 is an illustration of the event-flow.

![Event Flow Diagram]

The applications themselves (.NET based) can also generate events (GenericEvents), these are routed through theStateManager to the AC. Another type of event is the IntraActivityEvent which is described in chapter 9.
Chapter 6

User-Interface Layer

The UI layer has two main responsibilities. Firstly, to provide the user with handles to perform actions on activities (ie. the rest of the system). Secondly, to provide the user with feedback indicating the status of the system and activities. The UI can be made to handle several activities concurrently or just a single one. It can be made very simple providing only the basic create, deleted, resume and suspend actions on activities or can provide speech recognition and zoomable interfaces. These extra functionalities are provided by extra modules which can be ‘plugged’ into any UI.

6.1 The graphical user interface

The UI layer is relative independent of the rest the the ABC architecture. The AC and the CollaborationController [CC] are the two points of contact. All UIs must inherit from the ABCFrontend which provides delegate-methods (which can be overridden) for events and furthermore fields to access the AC and CC. We have built UIs for different devices, e.g. wearable computers and wall-sized screens, but have included a desktop-version with version 4.2 of the framework. This UI or ActivityBar is illustrated in figure 6.1 and is looks and works as a ‘TaskBar’ not for applications but for activities.

![ActivityBar](image)

Figure 6.1: The ActivityBar. (1) ActivityProxyList showing all activities. (2) Actions on activities. (3) ActivityButtons. (4) Shows status information. (5) Event-viewer.

A dropdown menu (figure 6.1(1)) shows all your activities with an associated screenshot, title of activity and a description. The dropdown can among other things be used
to ‘resume’ and ‘suspend’ activities via a right-click menu. The icons in (figure 6.1(2)) illustrate some actions which can be performed on activities. These include creating, suspending, zooming and inviting participants to activities. The bar itself shows a number of ‘buttons’ ((figure 6.1(3))) which denote current, recently resumed and ‘pinned’ activities. You can pin your favourite activities onto the bar. The status indicators (figure 6.1(4)) on the far right on the bar shows whether you are alone in your activity or not, whether you are publishing/receiving telepointers or voicelinks and whether or not you are connected to the server. Figure 6.1(5) is small form for showing events, e.g. another participant has joined your activity.

### 6.2 Zoomable activities

One of the main characteristics of activities is their ability to be between heterogenous devices. The Zoom-module is a way of providing a uniform view of an activity across screen-sizes. If you suspend your activity on a wall-sized display on later resume it on a small laptop for instance, there is bound to be an application which you cannot see. By zooming out on an activity, all applications are shrunked and moved towards the center of the screen to give the illusion of a camera zooming out. This allows you to quickly get locate the application you need and you can then zoom in on the application. This is illustrated in figure 6.2.

![Figure 6.2: The zoomable UI. (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.](image-url)
6.3 System Hooks

System hooks are a mechanism by which you can tap into the event-flow at an operating system level. You can intercept applications-specific events as well as system-wide events. We have used hooking to get global mouse-events as well as keyboard-events. They are used respectively to send telepointer-events and to register so-called “hot-keys” (e.g. Ctrl+Tab is used to display a list of activities). There is also an option to use hooking to monitor application-startups and -shutdowns. The current implementation relies on OS-calls to enumerate windows (see section 6.4) but this approach is costly and hooking should be used to replace it.

6.4 Monitoring applications

Perhaps move this section to client-layer?

The ApplicationMonitor is a threaded module which enumerates windows on the local system. It also keeps track on which windows it has already seen and publishes events whenever a new window is seen or an existing is lost, corresponding to an application startup and shutdown respectively. An activeStateManager registers itself and a number of applications which it owns with the ApplicationMonitor and receives events about these exclusively. The masterStateManager receives events about newly started applications as well and may choose to incorporate these into its activity.
Chapter 7

Activity Sharing and Collaboration

When multiple users are using the same activity server, they have the opportunity to engage in a collaborative scenario. This enables them to interact with the same services simultaneously while being aware of each other by means of any combination of telepointers, voice links and message chats.

This chapter will describe how collaboration is managed on the client side including the technical details of the implementation of telepointers and voice links.

7.1 Collaboration Management

When a client is working in single user mode without being connected to a server, all collaboration is disabled. When the client connects to a server, a collaboration controller will be created to enable the user to start a collaborative session with other users. To be able to engage in a collaborative session, it is obviously a demand that more than one user is participating in the same activity. This can be achieved by one user creating an activity and inviting one or more users to join the activity. The server will now make it possible for the invited users to resume the new activity.

When collaborating there are two distinct modes by which this can occur: synchronous and asynchronous collaboration. In synchronous collaboration more than one user has resumed the same activity on the same time. To support this kind of collaboration, the framework offers support for keeping the state synchronized on the participating clients, opportunity to use telepointers, voice links and message chat. In asynchronous collaboration more than one user resumes the same activity, but this is done in non overlapping time intervals. The support for asynchronous collaboration boils down to making sure that the state is synchronized between the participating clients. This is the job of the activity server.

To support collaboration, we have created a hybrid architecture combining the best from the client-server and peer to peer architecture design patterns. The client-server pattern is used to support and synchronize state changes on the participating devices. The peer to peer pattern is used to support collaborative widgets, which does not interfere with the state of an activity.
7.2 State changes

State changes are changes in the state of an actual service. E.g., an URL change in IE, a scroll in MS Word or similar. A change in a state is observed by the framework by pulling the services for their state at reasonable time intervals. If there is a difference between the last pulled state and the newly pulled state, a state change has occurred and the change should be propagated to the all participating clients.

A state change is propagated to participating clients, by submitting a publish message to the server. This message contains in excess of sender and activity information, a service id, a state component and a state value. For more details about the format see the AML chapter. When the server receives a publish message it propagates the message to the relevant clients by parsing the activity details in the message and propagating to the clients it knows have resumed the corresponding activity.

It is important to notice that due to the fact that all state changes in services are published through the server, it is possible to create a complete ordering of the events by having the server time stamp all events when the server receives them. As long as the client can be guaranteed that all relevant events will be received, the time stamp makes it possible for the clients to have a synchronized state. This is even true if the events are swapped in order by the server due to threading policies. The guarantee that events will be received is obtained by using a reliable transport protocol; in this implementation the TCP protocol has been used.

Although a complete ordering of the events exists, it is still important to think an extra time before implementing the event handling methods at the client. Two things must be pointed out before explaining the required implementation of state event propagation in the system. The first is that all state events concern a specific state component. These state components must be mutually independent, e.g. a change in the state of one state component must not influence the state of another state component. Also state
changes must be published with the new absolute value of the state component, deltas are not allowed. The second thing is that, due to the restrictions given in the first pin, an event arriving at a client with a time stamp pointing back in history can just be ignored. There is thus not a need for rollback or time warp mechanisms. The following implementation will ensure a synchronized state on all participating clients and will eliminate variables being flipped from one state to another and back due to the order of which the events arrive.

The following figure shows the simplest case of a state event propagation including two clients and a central server.

![Figure 7.2: The simples state event propagation situation.](image)

A client publishes a state event to the server. The server timestamps the event and propagates the event to all necessary clients, including the originating client. In this case the originating client can just forget everything about the time stamped event, as the client has already committed the change before publishing the event and no other events concerning this state component has arrived in the time between sending and receiving the state event.

A more subtle situation arises in the case where more events concerning the same state component arrives in the time span between publishing an event and receiving the time stamped version.

The situation is that client 1 has committed a state change resulting in the publication of state event A. The event is received by the server, time stamped and propagated to all necessary clients. In the mean time client 2 does something similar, resulting in a publication of a state event B concerning the same state component as state event A. In
this situation client 1 is forced to relay state event B until it received the time stamped version of state event A. After receiving the time stamped version of state event A client 1 can compare the timestamps on state event A and B and deduct, that state event B should be committed. For client 2 the situation is that at the time of receiving state event A the client has already committed an event with newer time stamp namely state event B and can thus ignore state event A immediately.

While the above sketched algorithm will work as intended when implemented correctly, great care has to be taken during implementation of the client side algorithm. Due to client side threading, the correctness of the algorithm can fail in certain situations. Consider the situation in figure 7.4 involving two clients and a server.

As seen in figure 7.4 the algorithm fails due to client threading policies. There are several ways to fix this problem. One way is to encapsulate the state change observation and the publication of the event in one critical region, making sure that no other threads tangle with the state during this period of time. Another way is simply to make the sending client commit its own event upon receiving. This will bring the participating clients in a synchronized state. We have chosen to implement this solution due to simplicity and the fact, that this solution does not demand any locks or like on the client, which could result in race conditions and deadlocks.

In figure 7.5 we present a precise pseudo code version of the concurrency algorithm applied at any participating client. The algorithm utilizes a hashtable called tracker and a class ServiceStateComponentTracker. We assume that all variables have been properly initialized and that
Ideally the server would parse all messages going through the server, extract relevant state information and merge this information into its internal representation of the activity state. This would keep the server up to date on the actual activity state and would eliminate any latecomer problems. This has not yet been implemented.

7.3 Telepointers

The telepointers are implemented using IP multicast. This is done to minimize the communication overhead when publishing a telepointer to multiple devices. Further more multicasts are well suited for telepointers, as this type of communication does not have to be reliable. If some multicast datagrams are swapped or lost in the network, this will not have any severe impact on the receiver. The receiver might experience a brief jump in the position of the telepointer or a brief pause. More important for telepointers are, that the datagrams are received as fast as possible and at a short interval, to make movement as instant and smooth as possible.

To be able to use IP multicast for all participating users of an activity, they all have to agree on an IP multicast address. This address is chosen by the creator of the activity and stored on the activity server as a resource coupled to the activity. This enables joining users to retrieve the resource from the server and start receiving and publishing telepointer positions on the agreed multicast address.

A user participating in an activity, which wishes to publish its pointer position as a telepointer, must use the following format:

\[ tp,<\text{IPAddress}>,<\text{User}>,<\text{MouseEventType}>,<X>,<Y> \]

This enables the receiver to deduce that this is a telepointer event from <User> at <IPAddress>. The event was of type <MouseEventType> and the new position is <X>,<Y>. Where <MouseEventType> can have one of the following values: "Move", "LeftButtonDown", "LeftButtonUp", "MouseWheel", "RightButtonDown", "RightButtonUP".
All these technical details are handled by the Collaboration Controller. If the client wishes to publish its pointer position as a telepointer, the client must invoke the function broadcastEvents on the collaboration controller with <ActivityID> and EventType.TELEPOINTER as arguments. This will cause the framework to lookup the correct multicast address and start multicasting the pointer position. A client wishing to receive telepointer positions on a corresponding activity must subscribe to the public event TelepointerEvent on the collaboration controller and be propagate an event every time a telepointer message is received on the corresponding multicast address. It is then up the subscriber to decide what to do with this event, eg. drawing of the telepointer etc. To stop publishing the function stopBroadcasting must be invoked.

7.4 Voice Links

The implementation of voice links also build on IP multicast. This is due to the same reasons as mentioned for telepointers. The same methods are used to agree a suitable IP multicast address.

A user wishing to publish a sound stream should multicast binary wave fragments at size 16384 bits prepended with a 160 bits sender id constructed by invoking SHA1 on the sender IP address. The wave fragments must have been encoded with a sound quality of rate 44100, 16 bits and 2 channels:

\[ \text{SHA1(<IP>) [160 bits]} <\text{Wave fragment}> [16384 bits] \]

This enables the receiver to remove wave fragments coming from itself and only playback fragments coming from other participants.

These technical details are all handled by the collaboration controller in cooperation with the voice link module. To start multicasting a voice stream, the client must invoke the function broadcastEvents on the collaboration controller with <ActivityID> and EventType.VOICELINK as arguments. This will cause the framework to lookup the correct multicast address and starts multicasting. Sound will be taken from the default sound input device normally the microphone. When multicasting sound, all received sound fragments will be played back by the voice link module. Eg. it is not possible to publish without listening or to listen without publishing. To stop the voice link the function stopBroadcasting must be invoked.

As a footnote it should be mentioned that the voice link module would greatly benefit from further development. The module handles voice link between two participants well, but with an increasing number of participants, the sound quality will be severely degraded. This is due to the ability to mix two or more sound fragments have not yet been implemented.
trackToServer(Event e)
{
    ssct = tracker[<unique_component_id>];
    ssct.flyingEventsCounter ++;
}

receiveFromServer(Event new_event)
{
    ssct = tracker[<unique_component_id>];
    if (<I sent this event>)
    {
        ssct.flyingEventsCounter --;
        if (ssct.flyingEventsCounter == 0)
        {
            if (ssct.saved_event.timestamp > new_event.timestamp)
            {
                ssct.last_committed = ssct.saved_event;
                <Commit saved_event>
                Return
            }
        }
    }
    else
    {
        ssct.last_committed = new_event;
        <Commit new_event>
        Return
    }
}

class serviceStateComponentTracker
{
    public ServerEvent last_committed = null;
    public ServerEvent saved_event = null;
    public int flyingEventsCounter = 0;
    }
    public serviceStateComponentTracker()
    {
    }
}
Chapter 8

Applications

To have any use of the framework, the framework must have a possibility to read and set the state of an application. This implies that the used applications must be ABC enabled. There are several ways to ABC enable an application, depending on the type and context of the application.

This chapter will present the different ways to make it possible to the framework to interact with an application. The three possibilities are to create a new application from scratch using the frameworks application classes, to wrap an existing native win32 application and to utilize the generic win32 application wrapper supplied in the framework.

8.1 External .NET Applications

When building a new application from scratch, it is possible to ABC enable it, by using the classes supplied by the framework. This implies writing the application in the .NET framework in a language of your own choosing. To do this, three things must be done:

1. Extend the class ABCSharp.abc.state.application.ABCApplication
2. Call the function initialize(int hWnd) from the constructor
3. Override the virtual functions in the super class

Extending ABCApplication and calling initialize(int hWnd) in the constructor, will cause the application to register with the framework via remoting. Overriding the virtual function will enable the framework to get and set states in the application. Further more the framework provides the possibility to override the lifecycle function beforeClose() in the super class, giving the application an opportunity to clean up things before being closed by the framework.

For more information on how to implement applications using the supplied API, see the chapter about Application Programming Interface.

Applications created this way, will communicate with the framework via .NET remoting. The client application will look for the client server on localhost port 888. This configuration is hard coded into the ABCApplication class. Due to this fact, it is only possible to have .NET applications register to a client server residing on the same machine as the application. When an application registers to the framework, a
new instance of the class ABCApplicationWrapper is created. This is the framework side representation of the .NET application. After registering to the framework, the framework can communicate with the application using the newly created ABCApplicationWrapper instance.

With the framework come the following applications which have been implemented in this way:

<table>
<thead>
<tr>
<th>Service Type</th>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcwrapper</td>
<td>ABCChat</td>
<td>a message chat</td>
</tr>
<tr>
<td>abcwrapper</td>
<td>ABCClientData</td>
<td>prototype client journal system</td>
</tr>
<tr>
<td>abcwrapper</td>
<td>ABCEasyViz</td>
<td>a wrapper to the EasyViz image viewer application</td>
</tr>
<tr>
<td>abcwrapper</td>
<td>ABCInjuredList</td>
<td>a prototype list of injured persons</td>
</tr>
<tr>
<td>abcwrapper</td>
<td>ABCInjuredChart</td>
<td>a prototype of a chart for marking injuries on a person</td>
</tr>
<tr>
<td>abcwrapper</td>
<td>ABCXrayViewer</td>
<td>a prototype of an x-ray viewer</td>
</tr>
</tbody>
</table>

Table 8.1: External .NET Applications

This way of creating ABC enabled applications is to be preferred over the following way. The reason is that creating or modifying a single application will not imply a recompilation of the whole framework, as the following method will.

### 8.2 Internal Wrapped Native Applications

When wrapping an existing native application as for instance Internet Explorer, Notepad or the like, this can be done in two ways. The first way is to create an external wrapper using the method presented above, as for instance the ABCEasyViz application, the second way is to create an internal wrapper. The second method will be presented below. As mentioned before the first method is to be preferred.

To create an internal wrapper of an existing application, four things must be done:

1. Extend the class ABCSharp.abc.state.application.Application
2. Create two different constructors
3. Override the virtual and abstract functions in the super class
4. Register the new class type as a service type in ABCSharp.abc.state.manager.ApplicationMonitor

Point two will enable the framework to correctly create instances of the wrapper. The class must implement two constructors. One taking a windowhandle as argument. One taking a servicestate as argument. Some existing wrappers also implements one taking a process, but this is not demanded and will not be used by the framework. Both constructors must invoke :base(<argument>)

Point three points will enable the framework to get and set state in the application. Also lifecycle functions as start() and end() must be overridden and implemented. The fourth point will cause the framework to discover when ever an application of the wrapped type is started and taking control of the application by creating an instance of the registered wrapper class.

After the creation of an instance of an internal native application wrapper, the framework communicates with the application via direct function calls on the wrapper class. How the communication between the wrapper and the actual application is
carried out, is up to the wrapper to figure out. This could be done via any permutation of the following techniques: Remoting, DOM objects, window messages, bit manipulation etc. It all depends on the interface of the application being wrapped.

In the framework the following applications have been wrapped via an internal wrapper:

<table>
<thead>
<tr>
<th>Service Type</th>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>browser</td>
<td>InternetExplorerApplication</td>
<td>InternetExplorer</td>
</tr>
<tr>
<td>doceditor</td>
<td>WordApp</td>
<td>Word</td>
</tr>
<tr>
<td>txteditor</td>
<td>NotePadApplication</td>
<td>Notepad</td>
</tr>
<tr>
<td>pdffreader</td>
<td>AcrobatApplication</td>
<td>Acrobat</td>
</tr>
<tr>
<td>imagesviewer</td>
<td>PictureManagerApplication</td>
<td>Microsoft Office Image Viewer</td>
</tr>
</tbody>
</table>

Table 8.2: Internal Wrapped Native Applications

Furthermore the following special wrappers have been implemented:

<table>
<thead>
<tr>
<th>Service Type</th>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcwrapper</td>
<td>ABCApplicationWrapper</td>
<td>.NET ABC Applications</td>
</tr>
<tr>
<td>unknown</td>
<td>UnknownTypeApplication</td>
<td>Generic wrapper for all Applications</td>
</tr>
</tbody>
</table>

Table 8.3: Internal Special Wrappeers

The ABCApplicationWrapper is used by the framework to wrap applications created in the above mentioned .NET way. The UnknownTypeApplication can wrap any application with a window, but can only find external visible state eg. frame size, frame position and executable path. It is not possible to retrieve internal state of the program. See next section for more info on the UnknownTypeApplication.

8.3 Internal Wrapped Unknown Type Applications

To improve support for clients working in single user mode without an activity server the UnknownTypeApplication wrapper has been implemented. This wrapper can wrap any application with a visible window, but is only able to retrieve external observable states. Eg. frame size, frame position and executable path.

This wrapper highly improves the usability of the framework in single user mode, as it makes it possible to enroll any application in an activity. On the other side this wrapper creates huge problems in collaborative mode. This is due to the fact, that no service type can be derived from the present information. Eg. it is not possible to migrate a service state to another device if the architecture or file system on the target device is different from the originating device.

Furthermore the UnknownTypeApplication wrapper may have problems finding the correct window handle for an application on any device. Many applications contains hidden windows and sub windows, this makes it to a guessing game to find the correct window handle. The current used heuristic is to look for visible windows with many child windows, as graphical user interface windows often have these properties. This works quite well for most applications, but might fail.
Chapter 9

Application Programming Interface

The API is as the name suggest the handles the programmer has to interact with the system. This section describes both what the system provides but also what is required of the programmer to build applications that interact with the ABC framework.

9.1 Services provided by the framework

The ABC framework provides some services beyond that of the operating system. The most important of these being the ability for programs to specify state which should be shared as well as persisted. The state of a program is defined exclusively by the programmer while handling, storing and manipulating the state is done by the system whenever the program is part of an activity. The application is automatically integrated into an activity, but can still run as a stand-alone. The client also provides a mechanism with which applications can communicate internally within the same activity.

9.1.1 State

The data describing an activity (see also AML description in section 3.2) is divided into three layers. The Activity-, Service- and Component-states. An ActivityState consists of a number of ServiceStates which are again composed of a number of ComponentStates. The state of each application is therefore described by a number of ComponentStates which are basically key-value pairs. Both the keys and values are strings but the interpretation of the values are left completely to the application.

When the application is part of an activity which are shared among several participants, the state of the application is pulled by the system every few seconds and any changes are sent to the server and forwarded to the other participants as updates. Similarly, the state is updated by the system when a server-update is received. The application should therefore be able to react to changes in its state in some form or way.

The system also handles storing of the state between as part of an activity. This means that the program itself does not need to save information which have been defined as part of its state. When the activity is resumed once more the application will be
asked to resume the state as it was when the activity was last suspended. This is done by the system by simply sending a number of state-updates to the application which it is already able to handle.

9.1.2 Collaboration

Apart from the shared state of an application the system also provides some extra collaboration features (see section 7 for details) like telepointers and voicelinks. This means that the application programmer does not need to worry about out-of-band communication and can program his/her application on the assumption that such matters are already taken care of.

9.1.3 Intra-activity communication

The system provides an asynchronous event mechanism for services to publish events to all or other specific services in the same activity. This is especially useful if you are programming more than one application which are in some way dependent on each other. The case could for example be a patient-switching utility which would send events to an EPR or an XRayViewer or some other application which shows patient-dependent information. The mechanism can help in assuring that the activity in some way is ‘consistent’ across all services, but it does not strictly impose any consistency.

9.2 Application requirements

To build an (.NET form-based) ABC-enabled application you only need to let your form inherit the class called \textit{ABCApplication} which is found in the ABCSharp dll. This will result in an application which can actively part of an activity and is capable of reporting state to the system.\footnote{It will only report minimal state which consist of position, size of window and executing-file of the program} It is also possible to take an existing application and make ABC-enable it. We have done this with a couple of programs including Internet Explorer and Notepad. The basic procedure is to find some way to read the state of the application (in IE’s case this is URL and position of scrollbars). This task is made considerably easier if the application already has some API to get at. Then the recommended approach is to write a .NET application (possible without a visible form) and ‘control’ the application from this external application.

The following sections describe what can be done to include more application-specific state-information.

9.2.1 Annotations

The simplest way to specify what information should be part of the state of your application is to annotate the fields of the form. This is done with the ‘StatefulComponent’ annotation seen below. The \texttt{<type>} could be any of the currently supported types (see table 9.1). The default behaviour of these are generally to extract a string from the field and set the value again on updates.

\begin{verbatim}
language=Java,fancyvrb=true

Using the annotations alone thereby allow you to pick field-variables which are made stateful. The system will take care of the formatting and interpretation of the
\end{verbatim}
9.2.2 Overriding the default behaviour

The default behaviors listed in Table 9.1 can be overridden by overriding the get- and set-methods of the ABCApplication-class as seen below. This allows you to impose your own formatting and interpretation to the state-values and more complex state-information is easily achieved. This could for example be the case if the state-value was a value which it was inconvenient or hard to express as a field variable.

```java
public override string getStateComponent(string id) { ... }
public override void setStateComponent(string id, string value) { ... }
```

The use of the above methods effectively works as a filter for state-values.

9.3 Example application

In the following we will give a simple example of how to build an ABC-enabled application. The program in question will be a Chat application in which users participating in a shared application will be able to write small text messages to each other.

The basic idea is that a shared textbox will be used to display the text entered by any/all of the participants. A second textbox is used to write the text-messages and the state of this textbox is used to communicate who are currently writing a message.

A RichTextBox (as seen in Figure 9.3, (1)) which is simply annotated in Figure 9.4, lines 10-11 and has the system-defined default behavior. This means that the state-values. This default behaviour can be overridden and the procedure will be described in the next section.

<table>
<thead>
<tr>
<th>Type</th>
<th>Default get</th>
<th>Default set</th>
</tr>
</thead>
<tbody>
<tr>
<td>All primitive types</td>
<td>get value</td>
<td>set value</td>
</tr>
<tr>
<td>TextBox</td>
<td>get .Text value</td>
<td>set .Text value</td>
</tr>
<tr>
<td>RichTextBox</td>
<td>get .Rtf value</td>
<td>set .Rtf value</td>
</tr>
<tr>
<td>... some more</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 9.1: The default behaviors of supported annotation types
Figure 9.3: The ABCChat program. (1) The shared ‘chat’ textbox. (2) The private textbox writing messages. (3) The statusbar showing if someone is writing a message.

RTF content of the field is made part of the state of the application with the id talk_all.

A TextBox (as seen in figure 9.3, (2)) which have a application-defined behavior. The code for the behavior is implemented in the overridden functions. This means that if the user is writing a message the value of the state-component with id talk_you will be ‘<user> is writing a message’. This value is set as the text of a status-field in line ... which means that all participants will be able to see who are currently writing a message.
public class ABCChat : ABCApplication
{
    [StatefulComponent("talkyou")]
    public System.Windows.Forms.TextBox tbTalkYou;


    [StatefulComponent("talkall")]
    public System.Windows.Forms.RichTextBox rtbTalkAll;

    ...

    public override string getStateComponent(string id)
    {
        // write status string
        if (id.Equals("talkyou"))
        {
            if (this.tbTalkYou.Text !="")
                return user.Name +"|writing";
            else
                return user.Name +"|not_writing";
        }
        else
        {
            return base.getStateComponent (id);
        }
    }

    public override void setStateComponent(string id, string val)
    {
        if (id.Equals("talkyou"))
        {
            setStatus(val);
        }
        else
        {
            base.setStateComponent (id, val);
        }
    }

    ...

Chapter 10

Legacy Applications

The present ABC framework adapts two points of view on services: either they are made from the ground up for ABC or they are basically ABC unaware. The former type will typically be programmed in C# using the API described in the previous section. The latter is typically legacy applications like MS Word, Explorer, etc., and approached by extracting the minimum of information that is available.

A third category is legacy applications where source code is available. We here outline some approaches for integrating these applications as ABC services.

10.1 Overview of approaches

Here we start a list of potential approaches and discuss them in more detail below. We stress that it is not a comprehensive list, but only represent a first brain-storm.

The fundamental problem is communication of state information between the ABC Controller (or more correctly, a state manager) and the individual service. The state manager must be informed that state has changed as well as the exact nature of the change itself from the service. The service must be informed when a new state must be enforced in the application.

As a starting point, we will consider state information related to activity resuming and suspension first, and postpone discussion of collaboration information.

We envision the following connectors between StateManager and legacy service:

- **Socket connection communicating AML.**

  In this approach a special ABC adapter service is written that merely forwards (a subset of the) raw AML to a well defined socket (localhost, portnumber). This require the legacy application to be rewritten such that:

  - It will communicate over the defined socket.
  - That it will parse AML commands and perform suspend/resume behaviour upon request.
  - That it is able to parse AML documents and set its state accordingly.
  - That it able to represent its state in AML format.

  The advantage of this approach is that the ABC adapter does not contain any service specific behaviour and may thus be used for any legacy application.
• **Socket connection communicating proprietary format.**

This proposal is more or less the same as above except that if the legacy application already contains suspend/resume kind of behaviour it may do so in some proprietary format. This format may then be the more natural format of communication between the legacy application and the ABC adapter service.

It embody the same requirements as above except that it is now the ABC adapter service that must do the translation between AML and proprietary format.

This approach is also interesting if no good XML handling components are available for the legacy application programming language.

The liability is that a novel ABC adapter service must be written for each type of legacy application.

• **Remoting using COM or another remote communication paradigm.**

A remote method invocation protocol is from a programming point of view more easy to handle than low level protocols like socket communication. Thus if (D)COM, CORBA, or similar is available on the legacy side then this can be used to communication state information between the two.

The benefit is in the improved programming model that, all other things equal, may lower the effort and number of bugs.

The liability may be in steeper learning curve if remote method invocation programming is not applied in the legacy application domain.

A first technical scouting of the topic seems to indicate that though the programming model is appealing a range of technical problems arise. Unlike the Java environment that readily may work with any CORBA component, .NET is a much more closed world. There is an effort to integrate .NET and CORBA but it is in its initial phase. Several techniques are listed to make .NET and COM inter-operate using P/Invoke and Runtime Callable Wrapper (RCW) technique. The problems seems similar to those exposed in the Java Native Interface—semantic mismatches between the computation models (managed code versus unmanaged, garbage collection, exception handling, type marshalling, etc.) introduce a lot of pitfalls.

• **Linking directly to ABC API.**

If the source code is written in some of the languages directly supported by .NET one possible venue is to rewrite the application to run on the .NET framework. Thereby the integration mechanism in the previous section can be used directly.

All of the above points require substantial programming effort and the somewhat tedious nature of the tasks offer risk of introducing defects into the code that are difficult to track.

Based upon concrete experience with the above in terms of effort required, nature of defects, and defect ratio, one may consider issues of automatic generation of the boiler-plate code for any of these proposals. One may envision some kind of declarative description of the state variables that form the state space to be handled by ABC and some tool that process this description to generate code templates for both the ABC side as well as the legacy application side.
10.2 Collaboration

Tele pointers and voice. Discussion pending.
Chapter 11

Related Work

Related Work – pending
Bibliography
