Center for Pervasive Computing

Report Series

Publication

CfPC-2004-PB-1

Architecture Presentations:
Experiences from Pervasive Computing Projects at
Computer Science Department
University of Aarhus

Author(s): Henrik Bærbak Christensen, Klaus Marius Hansen,
Ulrik Pagh Schultz, Peter Ørbæk, Niels Oluf Bouvin
Version: 1.4
Status: Closed

1 Content and Version Control

1.1 Table of Content

1 Content and Version Control 2
1.1 Table of Content 2
1.2 Version Control 3
2 Introduction 4
2.1 Purpose 4
2.2 Audience 4
2.3 Overview of report 4
3 ABC: Activity Based Computing 5
3.1 Architectural Drivers 5
3.2 Architecture Description 5
3.2.1 Activities 6
3.2.2 Framework aspect 6
3.2.3 Logical view 6
3.2.4 Deployment view 7
3.3 Lessons Learned 8
3.3.1 What worked well? 8
3.3.2 What worked questionably? 8
3.4 Conclusion 9
4 EPCiR 10
4.1 Architecture Description 10
4.2 Lessons Learned 12
5 Kimura 13
5.1 Architecture Description 13
5.2 Lessons Learned 14
6 Distributed Knight 15
6.1 Architecture Description 15
6.2 Lessons Learned 17
7 POMP 18
7.1 Architecture Description 18
7.2 Lessons Learned 18
7.3 Conclusion 19
8 Topos and predecessors 20
8.1 Architecture Description 20
8.1.1 Topos Architecture 20
8.1.2 OpenLSD Framework 21

8.1.3 Servers and Agents 23
8.1.4 Push-Pull 23
8.2 Lessons Learned 24
9 HyCon and predecessors 26
9.1 Architecture Description 26
9.2 Lessons Learned 28
10 Conclusion 29

1.2 Version Control

Version	Date	Author	Summary of Changes
1.0	17-1-2004	HBC	Initial version and template
1.1	26-1-2004	Various	Most contributions collected.
1.2	8-3-2004	KMH	EPCiR, Kimura, Distributed Knight
1.3	6-4-2004	HBC	POMP short description inserted
1.4	13-4-2004	HBC+various	Final version

2 Introduction

2.1 Purpose

This report describes architectures and experiences with building these from a number of projects related to pervasive computing that have run with participants from Department of Computer Science, University of Aarhus over the last couple of years. The architectures were presented at a presentation day on 7th January 2004.

The main purpose of the presentation day and this document is to serve as referential basis for the work going into defining and researching architectures for palpable computing as described primarily by the work package 2 in the Description of Work for the PalCom EU project (http://www.palcom.dk). Software architectures are described and discussed as the primary benefits and liabilities of the architectures as well as the processes that lead to them are presented.

2.2 Audience

This technical report is primarily meant to serve as referential material for the people in the PalCom project engaged in architectural design and evaluation.

2.3 Overview of report

The report covers the following architectures (presented in the order they appeared at the workshop):

1. **Activity Based Computing prototype** (Henrik Bærbak Christensen): This prototype system has been conceptualized, designed, implemented and evaluated at various workshops in the period medio 2001 to end 2003. It is a prototype system focusing on the concept of maintaining 'activities' for users and is in its present form focused at clinical work.
2. **EPCiR** (Klaus Marius Hansen): This is a proof-of-concept prototype for a residential pervasive computing platform based on an Open Services Gateway initiative (OSGi) implementation. It has been developed throughout 2003 and shows the status and relevance of an OSGi platform.
3. **Kimura** (Klaus Marius Hansen): Kimura was developed from early 2001 onwards at Georgia Institute of Technology. It supports everyday office activities using a ubicomp environment including large interactive displays. The communication in the system is based on events and implemented using a tuple space.
4. **Distributed Knight** (Klaus Marius Hansen): Distributed Knight supports informal, creative object-oriented modeling on a variety of input and output devices. Its architecture has been designed using type-based publish/subscribe as a main communication mechanism in order to support decoupling of components.
5. **POMP** (Ulrik Pagh Schultz) : POMP is a virtual machine designed to support applications that execute in a pervasive computing environment. The key features are support for strong mobility of programs and the ability to decompose programs into independent units that can be moved independently.
6. **Topos** (Peter Ørbæk and Michael Christensen) : Topos is a spatial hypermedia application and architecture supporting 3D spatial organization of documents and working material as well as on-line collaboration.
7. **HyCon** (Niels Olof Bouvin and Frank Allan Hansen) : HyCon is an open hypermedia framework aimed at supporting context-aware hypermedia, allowing users to, e.g., create links and annotations to geographical locations while in the field.

3 ABC: Activity Based Computing

3.1 Architectural Drivers

The ABC framework has been developed in context of the *Pervasive Healthcare project.* The project was initiated mid 2001 and ended mid 2003.

The project was organized in four themes. Each theme studied pervasive computing support for a particular work situation. The themes were

1. Medicine handling. Here mainly the work of nurses was studied.
2. Prescription. Here mainly the work of physicians was studied.
3. Collaboration. Here collaborative aspects were studied.
4. Patient. Here the focus was the patient.

The ABC framework was conceptualized, designed, and implemented in a prototype version for the purpose of demonstrating pervasive support to nurses and physicians in a number of design and evaluation workshops. At these workshops, clinicians were asked to simulate normal working situations with the aid of the prototype software in order to evaluate the usefulness of ideas and the prototype.

Thus the overall driving quality attribute of the software system was *end user functionality* as the prototype had to demonstrate enough realistic functionality to enable workshop participants to role-play clinical scenarios.

A second quality was of course to study various architectures for handling *activities,* one of the major novel aspects of the architecture.

3.2 Architecture Description

The ABC framework has gone through a number of major architectural revisions, typically in order to accommodate new demands as the project progressed through the themes.

1. **Medicine theme:** Here the focus was detection of *activities* in work situations based on context sensing. Persons and artifacts were augmented with sensors (RFID) that allowed the computing system to infer location. This information combined with heuristics of recurring tasks allowed the computing interface to be tailored automatically for the task at hand. The archetypical example is the medicine tray that a nurse puts onto the bed tray on a patients bed. ABC senses the nearness of the nurse, medicine tray, and patient, and infers that the EPR system should be reconfigured to show the medicine schema for the particular patient and with the EPR logged in as the nurse. Focus was on being able to forward these activities to any computing device in the vicinity of the nurse; and that activity execution should not be automatic.
2. **Prescription theme:** Context triggers were not found that allowed prescription situations to be automatically identified with any usable accuracy. Instead we focused on the fact that physicians work is often interrupted. Thus the prescription of a particular patient is a task that is suspended and resumed many times over the day as new information, lab results, consultations, X-rays, and so forth are accumulated. The ABC was reworked to allow *activities* to be long-lived, that is a physician could reestablish a given working context (both data context and graphical user interface context) precisely and quickly on any available device.
3. **Collaboration theme**: In this theme, activities as supported by ABC were augmented to serve as sessions for collaboration. That is, several persons could join the same activity and thus work together. The activity served to keep the graphical and data contexts of all participants synchronized, and supported tele pointers and audio feedback. Asynchronous collaboration support was also added in that all computer manipulation performed on an activity could be recorded for later playback.

The patient theme did not have any implications for the ABC.

3.2.1 Activities

Activities are central to the ABC architecture. Technical, an activity can be defined as:

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

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 by a number of radically different applications as long as they provide the same service (i.e. we can 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 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.

3.2.2 Framework aspect

ABC is meant to be a framework that allows activity-enabled services to be programmed with a minimal of 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 (like discovery); and a client-side run-time middleware component that allows client-side services to communicate with the server as well as the user to manage, inspect, and execute activities.
2. **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 an un-typed object (for suspend) and parse this object back into a running service (for resume).

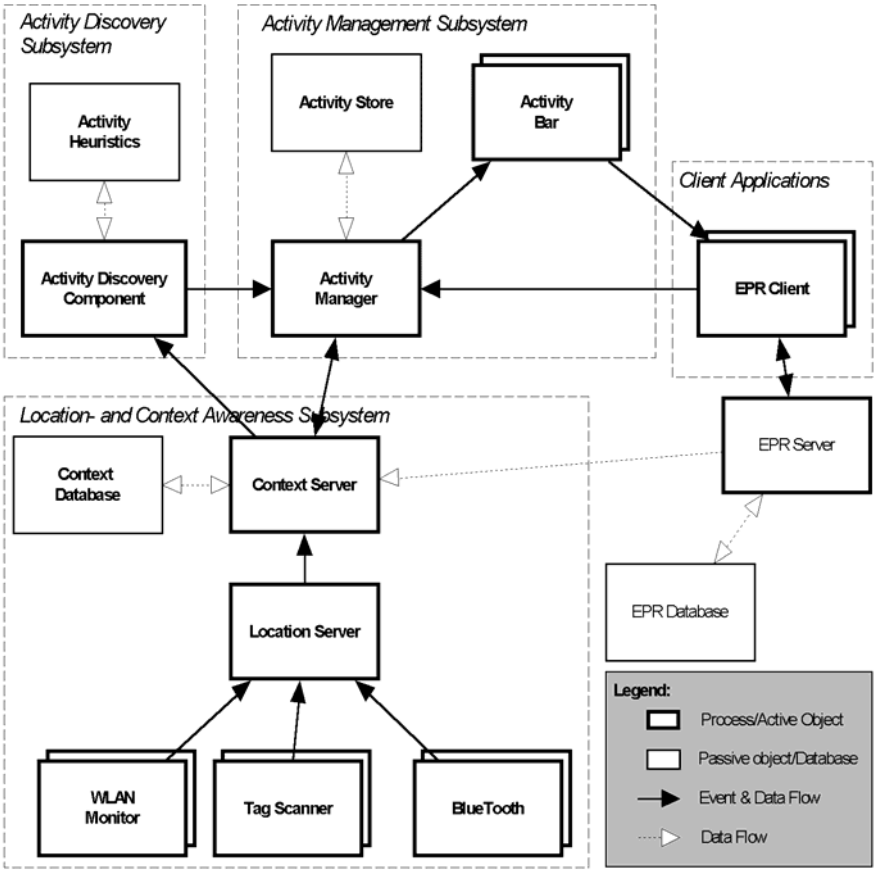
3.2.3 Logical view

The logical view focus on functional, run-time, components and their data- and control flow relations. The diagram below outlines the basic architecture. The notation is non-standard but explained in the legend. The diagram is overlaid an indication of a more coarse-grained logical division into subsystems.

The major components and their responsibilities are:

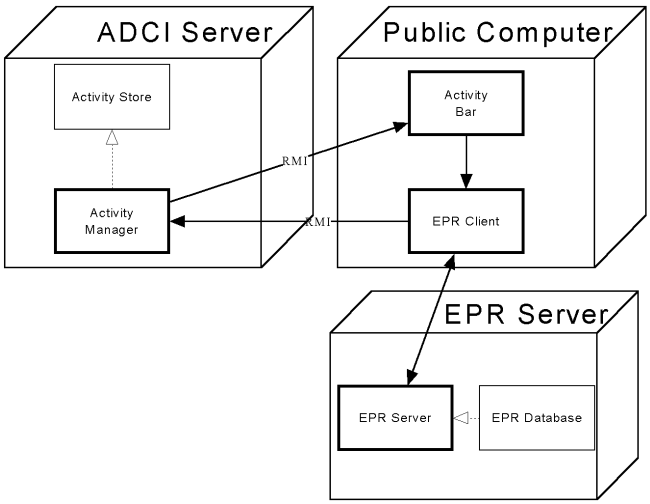
- Tag Scanner, WLAN Monitor, BlueTooth: These components handles the hardware and generate location/movement events that are sent to the location server.
- Location Server: Receives events, like tag-enter and tag-leave, from the hardware and maps hardware IDs to logical IDs. Events are sent to the context server.
- Context Server: Maps logical IDs to physical location information based on knowledge of the physical location of scanners and knowledge of what tag any given person or thing is wearing.
- Activity Discovery Component (ADC): Infer possible activities based on information from location- and context server and heuristics about recurring activities in healthcare. Once they are created, they are stored in the activity store.

- Activity Store: Detected activities, as well as activities explicitly created by healthcare staff, are stored here. Upon storing, the activity manager is notified about new activities.
- Activity Manager: Receives notifications of new activities and forwards activities to all pervasive computing equipment (more accurately, the activity bar running on the device) that is near the person that the activity relates to.
- Activity Bar: Receives activities and presents them non-intrusively to the healthcare staff. It is a separate application that resembles the task-bar known from the Windows operating system. Activities are not activated before the user explicitly selects them, typically by clicking the icon of the activity on the activity bar. Upon activation the activity is forwarded to the proper application, typically the EPR system, where it fetches relevant data and formats the user interface properly.
- Electronic Patient Record System: Third party database and application handling patient record information. Accepts activities from the activity bar and fetches data and formats the user interface according to the specifications of the activity.



3.2.4 Deployment view

At run time the diagram below shows how run-time components are divided over computers in a typical setup.



3.3 Lessons Learned

An number of issues can be raised based on the experience gained in designing, implementing, and maintaining ABC. Below we have tried to present issues that we felt worked well and issues that worked questionably.

3.3.1 What worked well?

- **Activity concept.** Based upon our experience from the workshops we find that integrating support for activities, understood as the ability to swiftly change work context and resume work context on any available device, is a important and a major benefit for nomadic work like is seen in healthcare.
- **Collaboration aspects.** Collaboration is an important issue in a Palpable Computing universe. Our experience was positive. Especially it seems that the activity concept provides a natural abstraction for collaborating people through which session management can take place. Activities provides a natural starting point for initiating collaboration because it is focused upon a specific task.
- **Ant build management.** In the developing process Ant proved to be of great value in several aspects. It streamlined of course build management which was rather complex. It stream lined execution as it provided targets for executing all the relevant applications. And finally but very importantly it served as up-to-date documentation and communication link between the four developers on the system.

3.3.2 What worked questionably?

- **End user functionality was primary architecture quality.** ABC had to be demonstrated and used in workshops at regular (rather short) intervals. This meant that functionality was the highest priority in the development process, not conceptual integrity, modifiability, testability, nor other qualities of a more software engineering related interest.
- **Framework support for service developers.** ABC provides a very slim interface for developers of activity-aware services. They have to construct the state object themselves, and parse it back again. No support is given by ABC for this task. It turned out that converting a large service's state into an untyped object is an error-prone and tedious task. Thus we must look into how to improve this.
- **RMI communication.** More or less all communication within ABC's distributed components is done using RMI. While this is often the proper choice it also means that if some of the components by any chance does not run properly the full system comes to a grinding halt. Thus ABC is rather fragile to malfunctioning components. A more louse coupling between vital and nice-to-have components is desireable.
- **Server based.** ABC is basically server based. Activities are stored centrally. While this has many benefits, it of course also introduces the single-point-of-failure problem.

- **Consistency of un-typed configuration data.** Some components communicate via un-typed data. For instance, the discovery component of the ABC architecture relies on object identities defined as string attributes: when the interpretation was changed from RFID ID's to CPR ID's for persons, discovery was disabled but this fact was not realized before several months after the change was made; and meanwhile the new interpretation was coded into several other components. Maybe there is an interesting research challenge buried here: how to ensure consistency in the face of configuration data forming the basis for highly flexible and run-time configurable software.
- **Persistence using Java object persistence.** This simply does not work in a high-paced development – class definitions change much too fast for persistent objects to keep up. The consequences is that no data is ever kept persistent.

3.4 Conclusion

If architecture is the major research topic (as it is in PalArch) we are faced with a challenge, namely to test our ideas out in practice with end users to provide feedback on ideas *and* to research and test purely architectural issues and ideas.

In our opinion, this gap cannot be bridged by end user prototypes alone. End user prototypes must prioritize end user functionality to make sense. This quality priority is contradictive to more architectural oriented qualities given the relatively low amount of man-power resources in the project. We therefore advocate **a 2-stringed strategy** in which both *end user prototypes* as well as *architectural prototypes* are produced. While the former is focused on presenting ideas for end users no matter how it is programmed, the latter has purely architectural focus and will probably not provide any functionality of relevance to end users.

The **developers' perspective** is important. How do programmers write services to run on our palpable architecture? Writing RMI based applications in Java is very simple for programmers due to automatic stub and skeleton generation and the run-time ORB. In contrast, our ABC programmers are offered next to no support in defining state. We must design an architecture that is programmable in practice.

Attention must be paid closely to the **distribution and communication mechanisms** used. When should we use synchronous calls? When use asynchronous communication? When is it ok do decouple communication in time, in space.

4 EPCiR

The Enabling Pervasive Computing in Reality (EPCiR; http://www.ooss.dk/epcir) project was concerned with investigating and experimenting with pervasive computing technologies that would be widely available within a few years, i.e., technologies there were commercially available during the project.

To ground the work in the project a number of scenarios concerning residential pervasive computing were developed and used. The scenarios centered on home care and home alarms and their actors and communications are illustrated in Figure 1.

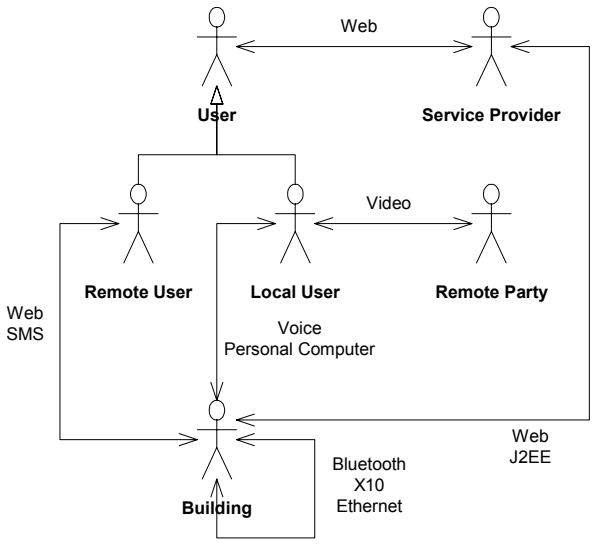


Figure 1. Actors and Communication in Basic Scenarios

Based on these scenarios, prototypes were built and their architecture evaluated.

4.1 Architecture Description

The following architectural qualities were seen as architectural drivers in the business context (which included service and content provision in residential pervasive computing):

- *Availability.* High availability is critical in residential scenarios both with respect to availability of remote and local services
- *Security.* Protection from unauthorized use and protection of privacy is essential
- *Usability.* Given a diverse set of users and the residential setting of use, high usability is seen as a primary goal

Following a technology identification phase, it was decided to build the prototypes on top of an Open Services Gateway initiative (OSGI; http://www.osgi.org) implementation. OSGi provides ”open specifications for the delivery of multiple services over wide-area networks to local networks and devices” and is supported by among others IBM, Nokia, Oracle, Philips, Siemens, and Sun. The OSGi implementations, which are in Java, run on top of “service platforms”. Examples of where service platforms may be included are set-top boxes, cable modems, residential gateway, consumer electronics, industrial computers, and cars. On these platforms, service providers may dynamically deliver services (such as residential monitoring, content delivery, or remote control) through OSGi. We used an OSGi implementation from ProSyst (http://www.prosyst.com).

The system architecture considered in EPCiR was based on OSGi and is depicted below at a very high level using UML. The description is made from an execution view showing computational nodes and components on these nodes.

The architecture is a hybrid between a generic OSGi architecture and a specific architecture in EPCiR in that it on a high level only contains generic OSGi nodes, but assumes that ProSyst-specific solutions have been used. Figure 2 shows this architecture[1].

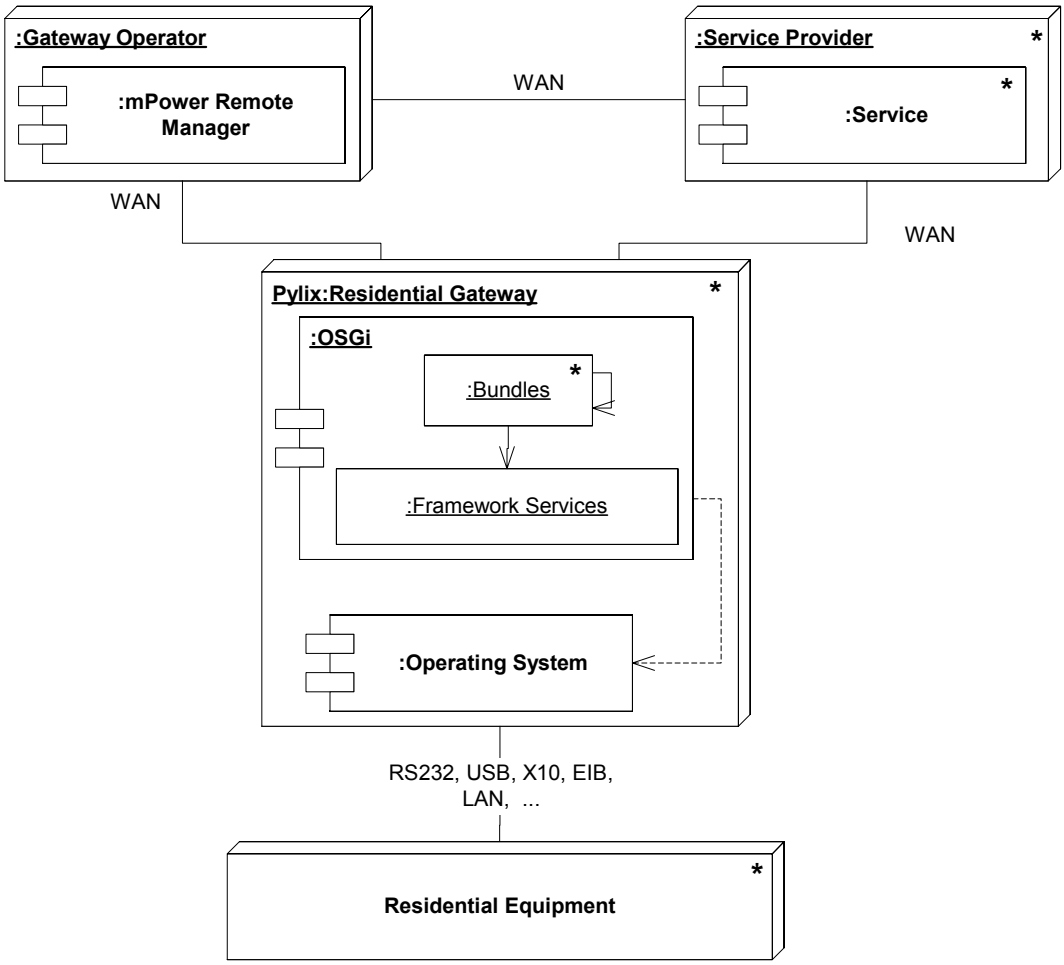


Figure 2. Execution view of architecture.

Below we briefly describe the responsibilities of each component:

- *Gateway Operator.* Monitors and maintains gateways using mPower Remote Manager (mPRM) from ProSyst (http://www.prosyst.com). Handles bundle administration also for service providers. Responsible for initial bootstrapping of installed residential gateways.
- *Service Provider.* Provides services of value to the residential user. An example from the EPCiR scenarios of a service could be the provision of MMS or SMS monitoring and controlling of a home. mPRM provides, among other, bundle inventory, service packages, and user charging functionality to service providers. The initial contact from service providers to gateways goes through the gateway operator.
- *Residential Gateway.* Contains an OSGi framework executing bundle components. Communicates with gateway operator and service provider to implement applications for residents. Gateway between the residence (including residential equipment) and the Internet.
- *Residential Equipment.* Equipment that is connected to the residential gateway in the form of sensors, actuators, alarms, or switches in the residence. Can be controlled, administered or

[1] The EPCiR project primarily implemented software running on a residential gateway and simulated the Gateway Operator and Service Provider parts.

monitored by the residential gateway. Residential equipment may also be home computers on a LAN.

The connector/connections on the diagrams use a number of very different protocols, it being one of the main ideas of OSGi to connect devices speaking heterogeneous protocols. Possibilities between the residential gateway, gateway operator, and service provider include web services (using HTTP), TCP/IP, and the ProSyst-specific management-based protocol (using UDP).

4.2 Lessons Learned

Specification at the framework level is useful for developing pervasive computing applications. The specification provides a useful level of detail for hardware and software vendors to create components that may interoperate. In principle, bundles/components from IBM may usefully coexist on a gateway with bundles/components from ProSyst since the platform standardizes among others interfaces and lifecycles of these. Moreover, the specification is currently sufficiently broad/loose so that a large array of types of pervasive computing architectures may actually be realized on top of the platform.

OSGi can provide a number of implementations of existing protocols on existing hardware. Given the number of supporters of OSGi there are a large number of implementations of the platform (on residential gateways, industrial gateways, PC's etc.) interfacing to a variety of communication protocols (Bluetooth, GPRS, LON, X11 etc.). This provides an easy entry to integrate devices in pervasive computing scenario if the devices are supported. On the other hand, integrating new devices (such as a temperature sensing device as in EPCiR) may be more involved.

Good development tools are essential. The ProSyst environment featured strong integration with the Java Eclipse platform including development, debugging, deployment, and monitoring of components on gateways. This was essential in developing for a highly dynamic platform such as OSGi is.

Reflection in OSGi is lacking. In order to be able to discover and use services automatically in OSGi-based applications the description of these need to be standardized and tied to the specific services/bundles. Currently, what is standardized is only the basic platform itself (such as deployment of services, security, and basic device management); what is needed are descriptions on a higher semantic level in order to discover, e.g., that a device connected to a gateway may be a temperature sensor capable of giving a temperature reading at a given interval and with a given precision.

Usability of self-configuration and errors in OSGi is problematic. Connected to the lack of (self-)descriptions of OSGi bundles/components, it is hard to provide useful self-configuration of devices connected to the gateway and also handling of errors at a user level is problematic since these need to be handled at a low OSGi-level Java layer.

5 Kimura

Kimura is an augmented office environment developed at the College of Computing, Georgia Institute of Technology. During 2001 we played a major role in implementing the basics of the environment and this presentation describes the system status at that point in time.

Figure 3 shows the basic setup of Kimura and a sample “activity montage” of the system. The environment integrates a desktop PC, peripheral display (here three SMART Boards) and a variety of virtual and physical context sensors. The environment is used to track and to interact with user activities. The user’s ongoing activities are represented as working contexts including documents related to the activity and interactions with people and objects related to the activity. The working contexts are visualized as *montages* on the peripheral displays and the current working context is shown on the desktop PC. Kimura automatically tracks the contents of the current working context.

Figure 3. Kimura Augmented Office Environment (Left). Kimura Activity Montage (Right)

5.1 Architecture Description

The Kimura architecture is a *blackboard architecture* in which a collection of agents communicate via a shared tuple space.

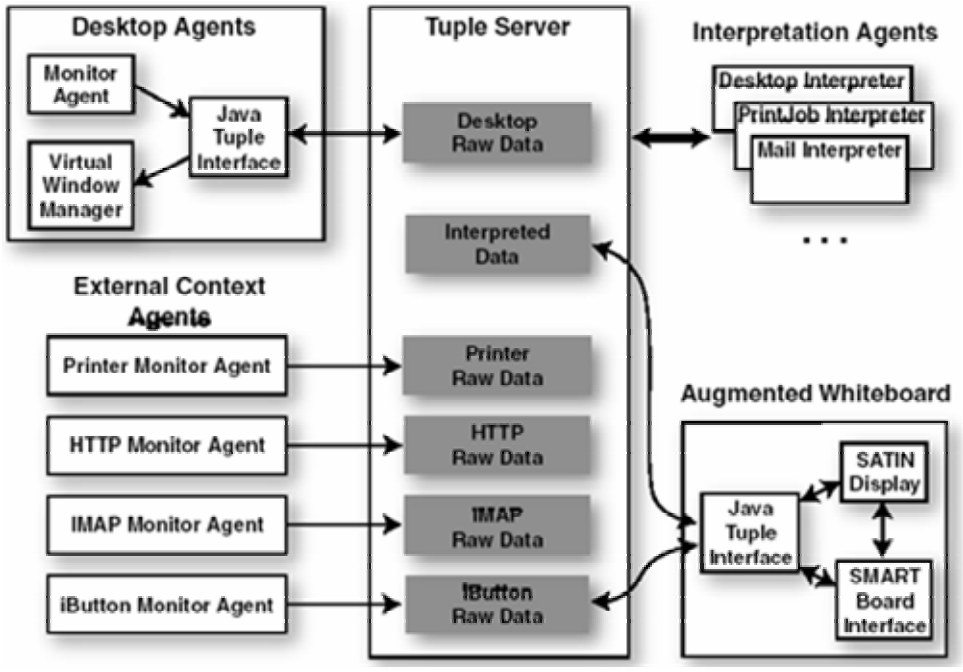


Figure 4. Kimura Architecture. Arrows indicate primary data flow

The *Tuple Server* acts as the blackboard and provides a number of tuple spaces (based on IBM TSpaces) in which data is put, subscribed to, and got by agent clients. The data storage is unstructured, persistent, and typed based on tuples of Java objects.

The *Desktop Agents* handles and monitors the interaction by the user. Users use an ordinary Windows PC with ordinary applications. Windows "hooks" let the system intercept events, package these as tuples, and put the tuples into the *Desktop Raw Data* tuple space. Examples of information captured are bitmaps of applications, how windows are moved, resized etc. Furthermore, the Desktop Agents change activity if SwitchMontage tuples are encountered both as a result of using the desktop itself as well as a result of external events such as using the peripheral displays.

The *External Context Monitoring Agents* collect information relevant to the user's activity, but external to the desktop. Examples are monitoring the printer or email queues of a user or monitoring the users location (simulated using an iButton). The collected information is published in designated tuple spaces. Work in progress is connecting these agents to a more general context-awareness framework (viz., the Context Toolkit).

The *Interpretation Agents* work on combinations of raw tuple data and interpret it. The use example above shows an example in which the combination of the location of a person and an email written to that person is combined to a tuple designating that the person is present and that we are interested in this person.

Finally, the *Augmented Whiteboard* components implement the interaction with and visualization on the peripheral display of activities. The user is able to interact with activities using gestures (e.g., moving, splitting, joining, or annotating) and the actions of the user is reflected in the tuple spaces and thus in the rest of the system.

5.2 Lessons Learned

Tuple spaces provide an easy and robust way of prototyping. The loose coupling inherent in tuple spaces lends itself to event-based communication and implicit invocation. This makes it easy, e.g., to add a new External Context Monitoring Agent to the system. Furthermore tuple spaces provide for automatic persistency through checkpoint of tuple space state and the concrete implementation used (IBM TSpaces) provided for tuple that could contain (user-defined) Java types.

Blackboards are a useful abstraction for interpretation activities. A number of pervasive computing applications lend themselves to interpretation/classification/recognition activities. Blackboards are a well-known and well-tried technique for this in that individual agents may implement different strategies for interpretation and may coordinate in flexible ways.

Applications may be hard to understand, test, and debug. Since most communication is implicit, it is hard to follow, e.g., execution traces through the applications built with this kind of architecture, making it hard to grasp (and reliably design) the runtime behavior of the applications. On the other hand, the architecture is flexible and for problems that are well-suited for being solved by blackboards, more conventional RPC-like architectures may lead to designs that are equally hard to understand.

Centralized/single point of failure. Tuple spaces/blackboards are most useful as a central repository where all information is available for (re-)interpretation. Trying to distribute data may put constraints on which data is available where, potentially leading to lesser flexibility and openness.

6 Distributed Knight

Distributed Knight is a distributed collaboration extension of the Knight tool for co-located collaborative modeling. The Knight tool was built on the premise that current electronic modeling tools are problematic in terms of

- *Usability*. Current tools (e.g., CASE tools) force users to focus on the tool instead of actual modeling.
- *Collaboration* which is essential for creative, constructive design is actually inhibited by electronic tools working on normal desktop PCs.
- *Extensibility*. Typically, the UML is used as a fixed modeling notation meaning that changes to the modeling notation cannot be made (easily) dynamically.

The Knight tool tries to solve some of these problems by combining gestural interaction and large electronic whiteboards. Figure 5 shows the use of the Knight tool using these interaction mechanisms.

Figure 5. Use of Knight on an Electronic Whiteboard (Left). Recognition of a Box Gesture as a UML Class

In effect, the Knight tool tries to combine the benefits of traditional whiteboards and electronic modeling tools while avoiding the liabilities. *Distributed Knight* tries to extend this to a situation in which

- *Users are physically distributed.* A project group may, e.g., need to work on the same model while being geographically separate.
- *Different input/output devices may be used.* One user may be using a desktop PC, another user a mobile Tablet PC and other users in a collaboration session may be using Distributed Knight on electronic whiteboards.
- *Mobility should be supported.* Users should be able to use the devices suited for the current use context when and if they need to collaborate on modeling.

In doing this, Distributed Knight tries to – much as the original Knight tool – provide a transparent user interface while providing the powerfulness of (distributed) computerized tools.

6.1 Architecture Description

Figure 6 shows a module view of the Distributed Knight architecture as a UML class diagram in which stereotyped dependencies between packages show the kinds of use relationship there exist between modules.

The left part of the figure shows the modules that are particular to Distributed Knight; the right part shows modules that are also used in stand-alone Knight. The Distribution module has been implemented with minimal changes to the original Knight architecture.

In Figure 6, the *Repository* is responsible for storing UML models and diagrams in accordance with the UML specification. Other modules/components can be Observers of the Repository, being notified when it changes, and are able to read and write from it. The data of the Repository is serialized in the standard XML Metadata Interchange (XMI) format specialized for UML, meaning that other modeling tools (such as Rational Rose) are able to read the native Knight save format.

The *Workspace* component is the actual window that the user works in. The Workspace and the *Radar* overview shown are two examples of Observers of the Repository, the Workspace being the only one which actually changes the repository.

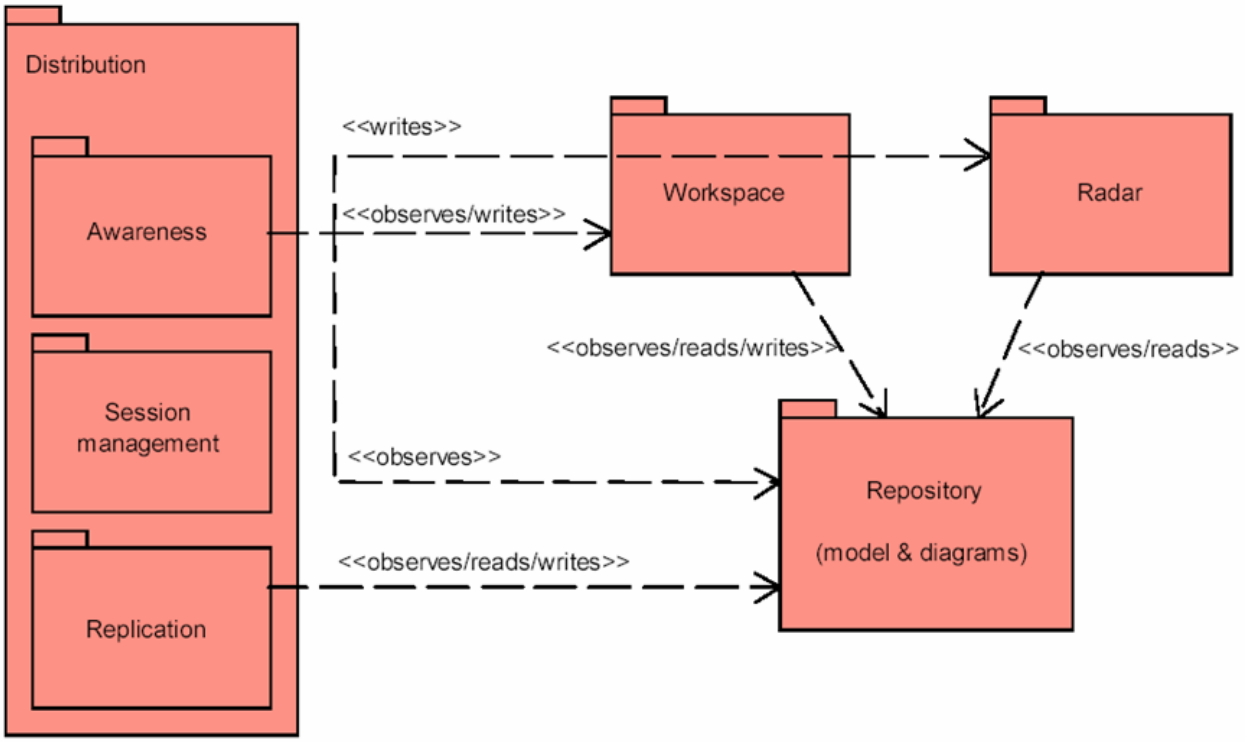


Figure 6. Module View of Distributed Knight Architecture

The *Distribution* component has the responsibility for all distributed collaboration-related functionality. The core of this functionality is publishing and subscribing to object-based events implemented using the Type-based Publish/Subscribe (TPS) distributed communication mechanism. TPS basically supports publishing events that are instances of application-defined types (i.e., objects) and subscribing to events based on the state of the events.

The *Replication* component implements a peer-to-peer data sharing scheme in collaboration based on TPS. It basically observes the Repository and the command history (used for, e.g., undo/redo) and publishes changes to other Distributed Knight clients based on the changes to Repository made by a command invocation. Furthermore it listens to changes to other peers' Repositories and act according to received events.

The *Session Management* component is responsible for location other peers, handling joining and leaving of sessions and so forth. The *Awareness* component is responsible for distributing synchronous context awareness cues such as the locations of collaborator's cursors or feedback on which gestures collaborators are drawing.

The event hierarchy of fig provides a "static model" of the communication protocol used among the Distributed Knight peers.

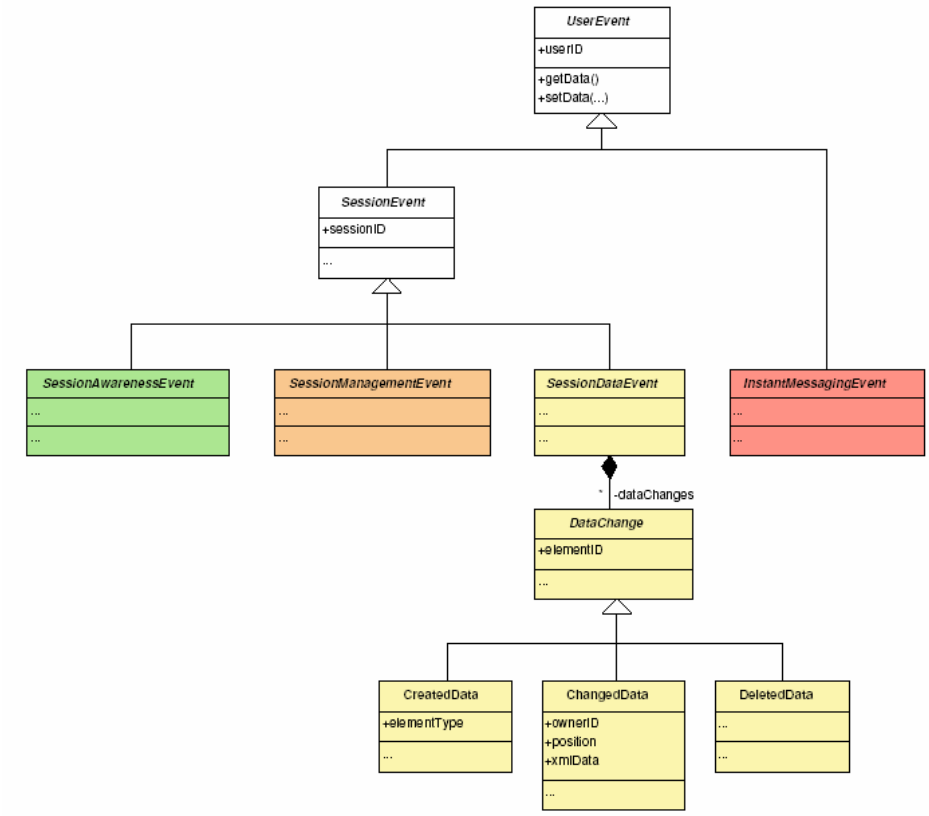


Figure 7. Event Hierarchy of Distributed Knight.

6.2 Lessons Learned

Publish/subscribe is a suitable abstraction for a class of peer-to-peer applications. The Distributed Knight architecture is essentially a peer-to-peer architecture. None of the peers are aware of the existence of central components and data is shared among peers. The only central component is a publish/subscribe server that basically just dispatches events to peers in the system. A decentralized component could be implemented using peer-to-peer techniques (cf. JXTA).

The event hierarchy of TPS supports understandability and extensibility. As with the event-based communication of Kimura, the publish/subscribe architecture is decoupled (in space, time, flow, and data) and thus supports flexibility. As an example, an instant messaging application was added to the Distributed Knight system late in the development process. This application supported instant messaging and was at the same time the main user interface for session management activities such as joining or leaving sessions. The addition of instant messaging was very much supported by the loose coupling of the system, but also by the fact that, e.g., session management was already represented as objects and explicitly modeled in the event hierarchy. The event hierarchy/event model also supports understandability as the various events are related to each other.

Understandability etc. is an issue. As in the Kimura case, the behavior of a publish/subscribe system may be hard to understand, interactions may be hard to reproduce etc.

7 POMP

7.1 Architecture Description

The Pervasive Object Model Project (POMP) is an ongoing effort to implement a language and a virtual machine for applications that execute in a pervasive computing environment consisting of heterogeneous devices. We see application mobility as a central concern in such an environment; the activity the user is currently performing can be supported by applications that migrate to devices in close physical proximity to the user or in close virtual proximity[2] to another application.

The Pervasive Object Model virtual machine (POM) supports applications split into coarse-grained, strongly mobile units that communicate using method invocations through proxies. POM is written in Java and currently supports execution of applications written in a minimal dialect of Smalltalk. We are currently investigating efficient execution of mobile applications and construction and de-construction of applications composed from strongly mobile units.

In POM, a mobile entity is referred to as a *system* (in the sense of a self-contained entity). Systems are normally employed in POM to modularize applications. A single virtual machine can host multiplesystems that communicate using proxies. A system can at any time migrate to some other virtual machine where it continues its execution (e.g., strong migration of all threads); it is thus a set of objectsthat migrate as a single unit.

POM applications are normally structured into several systems that communicate using proxy calls. Services that require access to native resources (e.g., access to GUI or to files) are typically isolated in a single system (which is then linked to the current location because of its use of native resources). The remaining application is then free to migrate to other virtual machines while maintaining an indirect reference to the services of the original virtual machine (in the form of proxies). Alternatively, the application can decide to obtain access to the services of the virtual machine that it has migrated to, for example by replacing a remote GUI with a local one.

7.2 Lessons Learned

Asynchronous messaging would have been easier to implement. POM communication is done using proxies, which implement a synchronous form of communication, since we found it easiest to work with at the application level. However, we require that migration be possible at any time, even when in the middle of performing a proxy call. For this reason, the virtual machine implements proxy communication using asynchronous messages that are automatically forwarded as systems migrate. Implementing the synchronous behavior involved making proxy calls interruptible at the thread level and enabling them to be resumed after migration, an added complexity that would have been unnecessary with asynchronous messaging. However, we still believe that synchronous messaging is the better choice since it is easier to use for the programmer.

Basic migration of applications is easy, doing it right it hard. Implementing the basic application migration in the form of serialization and deserialization of systems was a fairly easy task. It was much harder to make migration interact appropriately with proxy-based communication and primitives. For example, a system that is about to migrate must not serve any new requests, and existing requests must be migrated with the system. Along the same lines, primitives must either be fast, such as addition, or interruptible, such as synchronous proxy communication.

2Our language and virtual machine supports the notion of *virtual distance:* the distance between two computing systems measured in terms of bandwidth and latency.

Strong mobility facilitates making mobile applications. Modern applications are usually structured in terms of multiple threads, which enhances application responsiveness and facilitates programming advanced applications (at the expense of complexity). Making such applications mobile is trivial in POM, unless native resources are being accessed, in which case the application must be split into multiple systems. Without strong mobility, all threads would have to synchronize to allow the application to move. Without the ability to serialize the stack, the programmer would have to do reimplement migration for every application (as opposed to having it implemented once and for all in the virtual machine).

Strong mobility is not a silver bullet. Strong mobility easily allows a desktop application to move from one desktop to another. However, when moving between devices with different capabilities, it does not solve the problem: the application needs to adapt to the local resources. Moreover, migrating the entire application to a resource-constrained device is problematic. We believe that an appropriate structuring of applications into systems (modules) is the solution, but this hypothesis remains to be tested.

7.3 Conclusion

The POM platform supports strong mobility, which facilitates implementing applications that can move from one device to another, to match the mobility of the user in the pervasive computing system. However, the job is only half done: dynamic reconfiguration of applications must also be supported, to allow applications to adapt to local resources. We aim to support this feature by composing applications from multiple, collaborating systems. Nonetheless, a high-level means of structuring the connections between the systems is needed, as is a well-defined way of reconfiguring the way systems are combined into an application.

8 Topos and predecessors

Topos is the name of both the software infrastructure to support all the demonstrators of the WorkSPACE project, as well as the 3D interface client to the infrastructure.

The Topos client is a multiple-document-interface (MDI) windowed application presenting a 3D spatial hypermedia interface to document and material organization. A central concept in Topos is the *workspace* which is the primary means of grouping and organizing materials and documents in 3D space. The workspaces themselves can be grouped, mixed and connected in a variety of ways.

8.1 Architecture Description

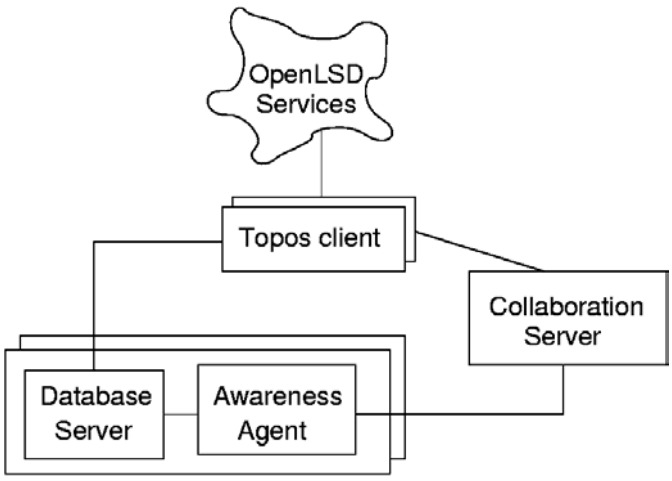
8.1.1 Topos Architecture

In the following we distinguish between the Topos *client*: the application that users run on their personal computer displaying the 3D interface, and the Topos *architecture*: made out of the clients together with servers and network services. Topos clients connect to a collaboration server to coordinate real time distributed collaboration, they connect to databases to store and retrieve the shared persistent state of their workspaces, etc. The client application also provides services to the network: for example the ability to display a workspace for another client on a large screen display.

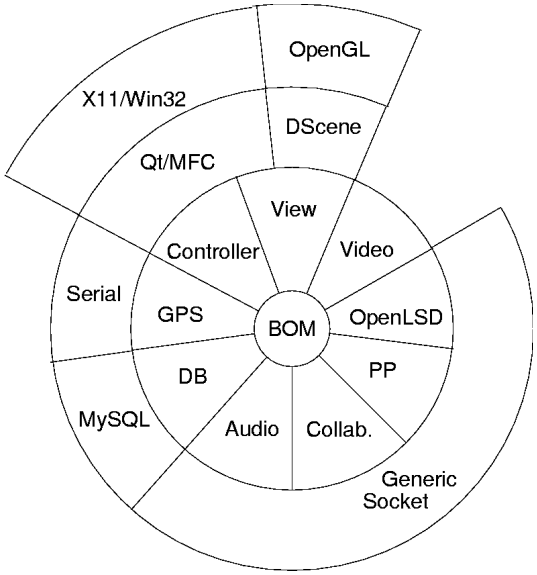
More concisely, the Topos architecture contains the following elements: SQL databases, collaboration servers, awareness agents, PDAs, location-based services (RFID tag readers, mouse services, location services, panel services, etc.), and the Topos clients.

The SQL databases keep the persistent and shared state of electronic workspaces. The collaboration server coordinates real time interaction between collaborating users. Awareness agents monitor changes to databases and enables off-line notifications of changes made since a workspace was last opened.

The circle diagram shows a rough picture of the run-time module architecture of the Topos



application. In the center is the Business Object Model (BOM) keeping the in-RAM state of the Topos objects (documents, workspaces, links, etc.) and their inter-relations. The persistence of the objects is facilitated through the DB layer interfacing to one or more MySQL databases. The 3D graphical interface is manifested through one or more Views which present a subset of the BOM via the DScene 3D scene graph class library developed in the project. DScene builds on OpenGL to implement hardware accelerated 3D graphics.



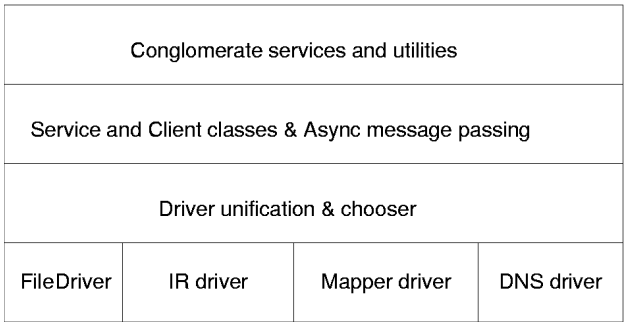
User input is primarily effected through the underlying window system and GUI library (we use Qt on Linux, MacOS X, and SGI and MFC under Windows). Input events are routed to a Controller which again delegates the events to various sub-controllers implementing different input modalities. Multiple controllers may be associated with a single view to support multiple users collaborating around a single display.

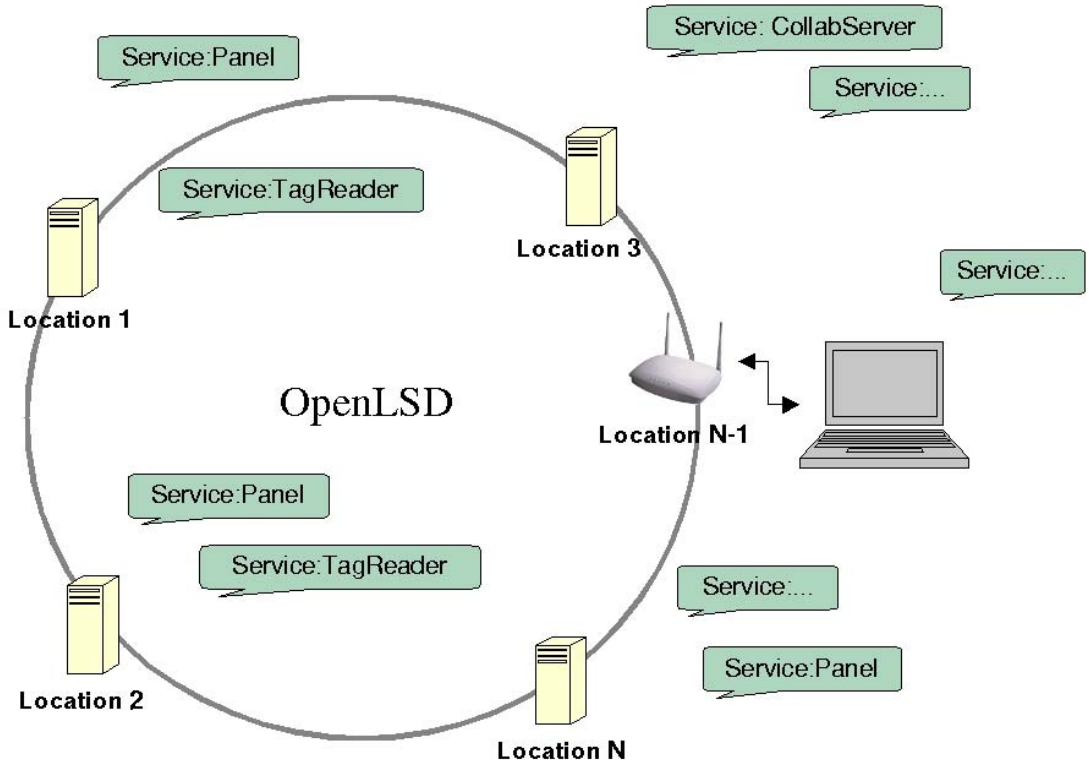
The GPS subsystem interfaces with GPS receivers through a platform independent serial library. The audio subsystem implements full-duplex network audio communication between workspaces and the video subsystem interfaces to platform specific video input methods to support augmented reality applications and vision based tracking.

The OpenLSD framework is described below and provides various network services. The push-pull (PP) subsystem implements peer-to-peer communication, and is also described below. All the network subsystems build on a platform neutral generic socket library supporting unicast and multicast UDP, TCP/IP, and IrDA TinyTP as the underlying protocols.

8.1.2 OpenLSD Framework

Our networked services are written on top of the OpenLSD C++ framework developed within WorkSPACE. OpenLSD is a location-based service discovery framework building on multicast IP, related to Universal Plug-and-Play from Microsoft, Salutation and SLP, but improving on these technologies in various aspects and emphasizing the provision of network services based on physical location. The diagram on the right depicts the layers of the OpenLSD framework.



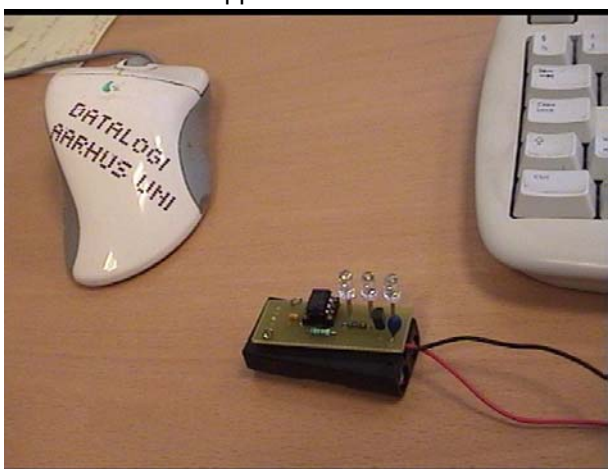


The figure above shows the cloud of services in the framework.

A basic tenet of the framework is that services are responsible for announcing themselves to the network. They do this by periodically sending multicast announcements to a "well-known" group address. An ethernet-like exponential back-off algorithm is used to avoid using too much bandwidth for these announcements.

The other basic idea of the framework is that clients and services know their own physical location, and may reveal it to others via service announcements. There is no central "big brother" knowing the location of everyone.

To provide physical location information we have implemented a location-mapper service within the framework which utilizes both an existing asset database for our stationary computers and talks SNMP (Simple Network Management Protocol) to Apple AirPort wireless LAN (IEEE 802.11b) base-stations to obtain location information for laptops and other wirelessly connected equipment. Laptop computers and PDAs may also obtain information about their physical location from small IrDA based IR beacons (based on a MicroChip PIC12F629 micro-controller, see picture) also developed within the project.



Two RFID (Radio Frequency ID) tag reader services have been implemented: One for the Philips I-CODE reader, and one for a low cost reader module from ACG (understanding the same I-CODE tags). The Philips I-CODE reader supports reading multiple tags at a time, allowing for tag interaction schemes not possible with the ACG reader which can only read one tag at a time. Both readers interface through a serial port and we use the platform-neutral serial communications library mentioned above to talk to the physical readers.

All services support any number of listening clients by virtue of the OpenLSD framework. The framework also provides prospective clients with location and network address information of running readers so as to eliminate manual configuration.

The OpenLSD framework is used by an announcement service that announces available databases to Topos clients. Users may choose among these from a drop down list. A pick-and-drop service allows sets of objects to be picked at one station and dropped at another station. A mouse service can be used to send mouse events from a tracking device on one machine to another machine hosting the display.

8.1.3 Servers and Agents

All collaboratively shared workspace states in Topos are kept in one or more SQL databases (we use the MySQL implementation).

Real-time collaboration is facilitated by a collaboration server that keeps track of connected clients and forwards update messages among them. Mostly, the data to update the databases is not sent through the collaboration server, but stored directly in the database. Instead, after the data are stored, an update message is sent to the collaboration server, and distributed, notifying collaborating clients of the partial changes that have happened. The clients then retrieve the updated parts of the objects to update their display. For interactive movements of objects in Topos, storing the new coordinates in the database before the move happens on the collaborating clients, proved too slow. Instead, during interactive movements, coordinate matrices are sent directly via the collaboration server and position information is not committed to the database until the movement has ended.

A similar mechanism is employed for doodle strokes (sketching). We found it important to support collaborative doodling to let users highlight areas of interest and for remote pointing. Collaboration messages containing doodle-line segments are sent while the doodle is being drawn, while the finished doodle stroke is committed to the database when the pen is lifted. It proved important to send line segments instead of points, in order to support several users doodling at once.

Most collaboration messages must convey the absolute ID of the updated object. This consists of a database name and host name, and a record number organized in a URL like scheme. These IDs can be quite long. We have therefore implemented a compression scheme in the protocol layer which (per-connection) remembers the ID of the last message , and transmits only those parts of a new ID that are different from the reference ID. In many cases this can halve the size of the collaboration messages sent, and thus saves network bandwidth and improves response times.

We use a central collaboration server and connect to it via TCP/IP. Another option initially considered is to avoid the central server and use multicast UDP to coordinate collaborative actions. We avoided this design as multicast IP is not widely accessible over the wide area and through firewalls. We have since conducted several collaborative sessions between Aarhus and the UK over TCP, which would have been practically impossible with the use of multicast.

The collaboration server keeps track of the presence of interactive users in a workspace. Users co-present in a workspace are made aware of each other using the small unobtrusive "presence viewer" (in the upper left corner of the screen shot) within Topos. Networked peer-to-peer audio UDP channels between the users in the same workspace are automatically created to facilitate seamless on-line collaboration.

An awareness agent watches changes made in workspaces and updates the state of closed workspace proxies to notify users who come on-line later, that changes have happened in those workspaces without users having to inspect the workspaces.

8.1.4 Push-Pull

The push-pull protocol developed for Topos is used for peer-to-peer communication between Topos and PDAs, web servers and other Topos applications. It was first implemented as a means of communication between a PDA holding bookmarks for workspaces and a Topos application running on a SpacePanel. Via an IrDA/TinyTP link and the push-pull protocol the PDA could "push" a workspace onto the panel, or alternatively "pull" a bookmark onto the PDA.

The protocol has been extended to support the "push-to-panel" feature from one Topos client to another Topos client over TCP. Here Topos clients act as OpenLSD service providers, by announcing a "panel" service. When a user selects "push-to-panel" from a menu, a list of physically nearby panels appear, based on location information carried by OpenLSD. When a panel is selected, a push-pull connection is made and the appropriate message is sent to the Topos client on the panel machine.

The push-pull protocol is also used to communicate between a Perl CGI script running under the Apache web server and a Topos instance working as a web server back-end. This setup allows limited remote access to Topos workspaces through an unaugmented web browser. Finally, the push-pull protocol is used for remote preview generation: Preview images for some document types cannot easily be generated on platforms not supporting the native application. For example, it may be nearly impossible to generate a preview for a MS Project file under Linux. However, a Windows machine may be configured to provide a document-rendering service via the OpenLSD framework. Enabling the use of this service on the Linux machine lets Topos on the Linux machine contact Topos on the Windows machine for preview generation of documents inserted into workspaces. The same service also allows the Linux machine to ask the Windows machine to open the document when the user double clicks on it.

8.2 Lessons Learned

Participatory development process. Topos is the application and the software infrastructure produced in the WorkSPACE project. It serves as the vehicle for experimentation with new technologies for spatial computing. Due to our focus on participatory design and use experience, it has been important to be able to provide our professional partners with working prototypes starting at an early stage of the project, both for use in future laboratory sessions, in our laboratory, as well as in the workplace of the architects. One of the implications of this is the necessity of focusing on the quality of the software. It must be able to "stand on it own legs", not requiring an ever-present programmer to keep it running. Another concern is that it must be easy to incorporate new technologies and embrace change as the prototype develops through the participatory development process.

Cross-platform. Topos was born from the ashes of the Manufaktur and HoloManufaktur prototypes of the previous EU Desarte project. Manufaktur pioneered the notion of 3D workspaces for organization of working materials. Where Manufaktur was written for the Windows 2000 operating system, HoloManufaktur was its sibling written for Silicon Graphics IRIX and designed for use at a HoloBench using stereo graphics and two-handed input through a Polhemus magnetic tracker. The two applications could inter-operate, access the same databases simultaneously and supported real-time collaboration. However, it was evident that maintaining two separate code bases was unsuitable for future development.

Thus Topos was born as a completely new code base written with cross-platform applicability in mind. Topos was originally written and designed to work on Windows 2000, SGI IRIX and SuSE Linux 6. However, during the project, these platforms changed underneath. Windows 2000 became Windows XP, SGI IRIX became more and more irrelevant due to the relative lack of graphics and processor performance on the hardware running IRIX: the PC platform simply developed much faster. SuSE Linux was changed to Redhat Linux 7. In the middle of the project the Linux version was ported to MacOS X as a proof of concept, however, this line of work was abandoned as supporting yet another platform was judged too expensive in terms of man-hours. At this point Topos is being actively developed under both Redhat Linux 7 and 8 and Windows XP.

Developing the application and the surrounding services for a cross-platform environment has sometimes slowed the development process but, we believe, has also helped improve the quality of the code, forced better architectural choices, and made the port to MacOS X possible. Using more than a single compiler has also made the code more standards-based and independent of particular compiler quirks. This has helped when changing development environment even on a single platform, eg. from MS Visual Studio 6 to Visual Studio.NET or from one version of GNU C++ compiler (GCC) to another.

Library versus application framework. Another experience, shared by many others, from the project is that it is easier to integrate 3rd party software in the form of libraries than software in the form of application frameworks. In our experience application frameworks in their nature have tendencies towards fairly narrow conceptions of the types of applications that can be built on top of them, often, amongst others, leading to a tight coupling of components within the framework. The consequence in some cases being a clash with the basic components and structures of an existing applications.

Model-view-controller. Much of Topos is built around a model-view-controller pattern. The automatic updates performed by the views when the model changes have been both a boon and a drawback: it has often been easy to implement new functionality at the model layer, but getting performance right by selectively switching off some of the automatic updates at the right points was not always easy.

Refactoring interface to GUI. We started out with a small interface to GUI/platform specific functionality, which grew larger as time went by. Perhaps because this is the main interface between the basic business object model and user interface this part of the code, in a rapid prototyping process, is more likely to develop in unforeseen manners with respect to structure and functionality. In our case, a refactoring of the interface has long been needed.

Compatibility between releases. We have used versioning of our preference files, our database records, as well as in our network protocols from day one. This has provided compatibility between releases and has been a large plus.

Human readable configuration files. At a late stage we changed our binary preference file format into readable XML. This has eased debugging of configuration related problems.

Platform independence and run time availability of menu interface. We defined all of our menu interface first in data structures in source code, and since in an external XML file, both for platform independence and for run time configurability. This has been a plus. The use of platform independent menu IDs has proved valuable and has been used eg. in the scripting language.

Own network protocols. We have defined a lot of our network protocols ourselves, which has given us freedom, e.g. to make connectionless protocols and asynchronous protocols where it was needed.

Unified metamodel. Topos uses a unified metamodel, the large number of externally visible object types are all internally represented by objects of a single class, and stored in a single database table. This has made database storage easy, and our own internal schema evolution mechanism has provided room for growth in a compatible manner without having to convert databases for users. The unified model is also used to support collaboration message exchange.

Zero-configuration setup of network services. The OpenLSD framework has allowed almost zero-configuration setup of network services, on the other hand latency of service announcements has some times been a problem. Also the framework does not scale easily to very large networks, although it can via manual setup.

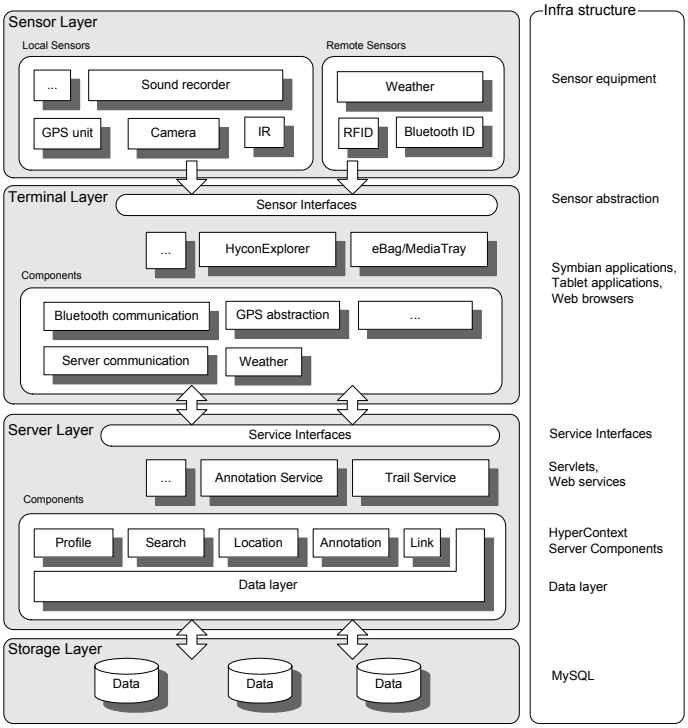
9 HyCon and predecessors

We have at Aarhus University over the years developed a number of open hypermedia frameworks and systems, such as Devise Hypermedia, the Arakne framework, and Xspect. These systems have addressed topics such as open hypermedia, collaboration, Web augmentation, and the exploration of the XLink format. However, none of these have explored mobility or context-awareness, and we have therefore designed a new framework architecture serving as a platform for the development of context-aware services.

By context-aware hypermedia, we see the need to go beyond just providing information and services depending on the user's tasks. We see a need to let the users create annotations, links, collections, guided tours and the like, automatically tagged with the captured context information (be it location or other contextual information). Thus we focus on creating more than just a context-aware browsing and navigation system, namely full context-aware hypermedia systems, where users can become producers and provide hypermedia structured materials to coworkers, school mates, or friends.

HyCon is still a framework in flux – the initial goal has been to adequately handle sensors, so that the system may be used for experiments with context-awareness.

9.1 Architecture Description



The HyCon Architecture

The HyCon service framework architecture is divided into four layers as shown in the figure above. The architectural approach is quite similar to earlier work, such as the Open Hypermedia System Working Group's Open Hypermedia Model and Construct, both heavily inspired by the seminal Dexter architecture. Key to the HyCon service framework architecture, compared to earlier work, is a extra layer dedicated to handle sensors and sensor information.

At the bottom of the figure above is the storage layer which handles persistent storage. The storage layer interface is available to components and applications in the next layer: the server layer.

The components in the server layer form the basic functionality of the framework and include a small set of reusable building blocks such as the data layer component, the location component, the subscription component, and the annotation component. The components are used to create service applications which implement interfaces to the component's functionality. Applications may use one or several components and add specialised functionality not available through the mixture of components e.g., computation on the output

from different components. This design provides a mechanism for decoupling the responsibilities of the building blocks and not creating mutual dependencies between individual components in the framework.

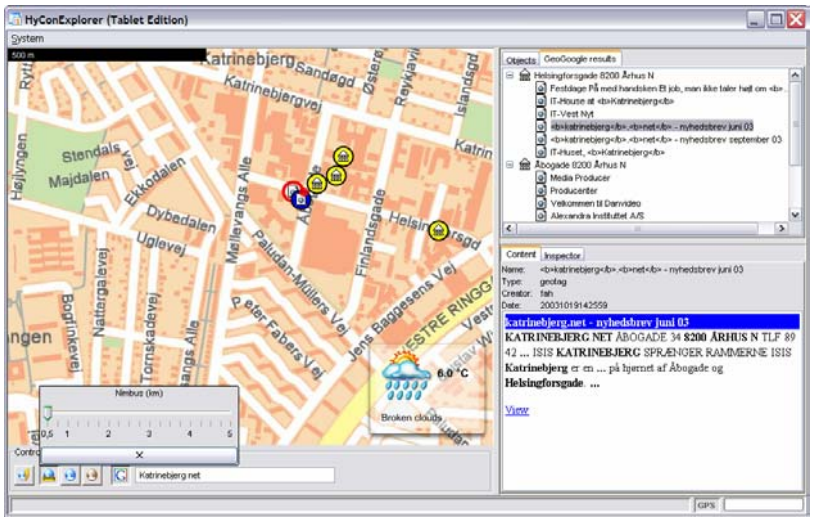
The purpose of the server layer is to provide data and functionality to the terminal layer. The division between reusable components and specialized applications is found again in the terminal layer: the components are implementing the communication to the server layer and the applications are implementing functionality and user interfaces appropriate for the terminal device in use. The hardware platform of the terminal layer may include all sorts of mobile equipment: laptops, tablet PCs, PDAs, and cell phones. These devices may have access to sensor information. Sensors can be integrated within the terminal device (e.g., a camera in a phone) or be external (e.g., a Bluetooth enabled GPS unit). Sensor information is accessed through tailored interfaces to components handling the parsing and computation of the raw sensor data.

The infrastructure and interfaces of the service framework can be realized in various ways—one way is the traditional client-server model which was chosen for our implementation. The functionality of the storage layer is handled by a MySQL server with a database having tables and relations corresponding to object and context relationships. The server layer offers a data layer component that maps between language objects and the database format. The data layer supports a basic set of methods to store, retrieve, query, and delete objects from the database. This set of methods provides the interface to the storage layer.

The runtime environment for components and applications in the server layer is an application server. The applications are implemented as Java servlets and Java Web services and thus provide their interfaces via CGI and SOAP to the terminal layer. The CGI version is internally represented as a filter chain with application servlets producing XML data output at the beginning of the chain and a XSLT servlet transforming the data to specific output formats (e.g., HTML or Scalable Vector Graphics) at the end of the chain. This has been implemented to easily support a wide range of mobile devices with different computation and display capabilities in the terminal layer.

The terminal layer may consist of many different hardware and software platforms as indicated above. We have experimented with terminal layer platforms ranging from tablet PCs to Symbian phones, and laptops. As sensors and content capturing devices we have used external Bluetooth enabled GPS units, built-in sound recorders, and digital cameras (both built-in and external USB cameras).

In order to integrate the context framework with existing online services, the components in the server and terminal layers can communicate and retrieve information from other resources than the storage layer. This enables integration with services providing online maps, weather information, and with search engines.



Left: Using the HyConExplorer for searching the neighborhood.

Right: The HyConExplorer in use by school children

9.2 Lessons Learned

The HyCon architecture is still a work in progress. As a demonstrator of the possibilities inherent in context-awareness, we have built the HyConExplorer depicted above. The HyConExplorer is based on a tablet PC, and provides users with location awareness, so that they may e.g., search for Web sites related to their current position, and create and browse hypermedia structures (such as links or guided tours) using their physical location as anchoring mechanism. This version of the HyConExplorer has been successfully evaluated in a school setting.

Other venues of exploration include further extending the range of sensors supported by the framework. The current version supports GPS, cameras, microphones (all integrated with HyConExplorer prototype), as well as RFID and Bluetooth awareness for “tagging” locations, objects, or people. We see many prospects of further development of these technologies, and the HyConExplorer will be further developed also for mobile phone devices.

At this point of investigation, we have found the notion of context-aware hypermedia worthwhile and interesting. Challenges abound, both of a hardware (e.g., GPS usually requires a separate device, which is cumbersome) and a software nature (especially the development of interactive software for mobile phones have proved challenging), but the benefits of having information systems respond according to circumstances are well worth the efforts.

10 Conclusion

The workshop ended with a short summary and discussion. The main topics that we need to carry on into the PalCom project, in particular with respect to work package 2: Open Architecture, is:

- Focus on **precision in architectural documentation and communication**. The workshop revealed a number of very different means of architectural documentation, most of which are characterized by a lack of precision. This is hardly surprising considering that the field of software architecture is relatively young; that no architectural notation is generally agreed upon; and finally that the formal training of most researchers and developers has not included architectural notation—it is still not widely accepted as part of the academic curriculum. However, that workshop showed the need for us to be precise about terminology and notation when discussing and presenting architectures.
- A close **attention to the communication mechanism used in distributed computing**. Palpable architectures are characterized by distributed communication between services running on a potentially large set of computing nodes connected in some kind of network. The presented architectures have used a number of different mechanisms like synchronous communication (Remote Method Invocation and similar techniques), asynchronous communication (tuple spaces, CORBA one-way calls, and similar), message queuing, publish-subscribe, and HTTP protocols. There is a need to provide and document a *vocabulary* of mechanisms that allows designers and developers to pick the proper mechanism for the problem at hand, instead of picking the one he/she is most used to. Examples discussed were things like *futures*, and *parbegin-parend* constructs.
- Attention to the problem of **typed versus un-typed information**. Interestingly some of the projects with a long track record reported that initial designs relied heavily on generalization hierarchies for classifying data but later designs had identified the strict classification as either too strict or unnecessary and had responded with designs where classification is handled through run-time type information. The benefit is a much more flexible design that is more easily adapted at run-time. The down-side is that dependencies between software components are much more subtle and difficult to manage when depending on run-time type information: no compiler can check consistency and there is a tendency that broken dependencies can go on unnoticed for a very long time. An example is given in the ABC section.
- Attention to **process** as part of the architectural challenge. Projects reported benefits of tools to aid process, like CVS for collaboration (and perhaps release management), and Ant for management and documentation of build processes.