CAPEUS: AN ARCHITECTURE FOR CONTEXT-AWARE SELECTION AND EXECUTION OF SERVICES

Michael Samulowitz, Florian Michahelles, Claudia Linnhoff-Popien
University of Munich, Dept. of CS, Oettingenstr. 67, D-80538 Munich, Germany

Abstract This paper introduces a comprehensive framework that allows mobile users to access a variety of services provided by their current environment (e.g. print services). Novel to our approach is that selection and execution of services takes into account the user's current context. Instead of being harassed by useless activities as service browsing or configuration issues, environmental services get seamlessly aligned to the user's present task. Thus, the challenge is to develop a new service framework that fulfils these demands.

The paper proposes a document-based approach; so called Context-Aware Packets (CAPs) contain context constraints and data for describing an entire service request. The core framework, Context-Aware Packets Enabling Ubiquitous Services (CAPEUS), reverts to CAPs for realising context-aware selection and execution of services.

Keywords: Mobile Computing, Context-Awareness, Context Representation, Task-Driven

1. INTRODUCTION

Future environments will host a vast number of mobile and wireless devices, besides to general-purpose computers. These smart spaces [1] offer a variety of services to their visitors, which may include intelligent home or office services. Multi-service environments involve a number of research challenges for mobile computing scenarios. One challenge is how to discover services when a user moves into a new environment. This problem may be tackled by service discovery protocols, like JINI [2], IETF SLP [3], or SDP [4]; the user may access a local service access point, which returns available service interfaces. Another challenge is how to describe service interfaces; because a new discovered service interface can only be selected and used if it can be understood. This leads to abstract interfaces [5], first work on this issue can be found in [6]. A third challenge, considering multi-service environments, is

context-aware service provision. In particular, the system selects and executes services in regards to the user's task in mind.

It is the third feature that has been least addressed by previous research projects. If for example a user's present task requires printing a document, then it should simply be printed. Of course this action should take into account her current context, here primarily the location. The printer should reside in her near neighbourhood, preferably in the same room, not in another building. Or, she is on the move, the system predicts her route and when passing the whereabout of her printout she gets the directions via a head-mounted display.

The paper presents a novel architecture for context-aware service provision in *ubiquitous computing* environments. We propose that just offering services to a user is not enough. Services should be aligned to a user's task; the user shouldn't be harassed with selecting the right service for the task at hand. Anymore, execution details should not be visible to the user.

The following issues were crucial to our architecture:

- service invocation mechanisms that regard arbitrary contextual constraints, e.g. location or time,
- a sophisticated representation scheme for representing contextual information, in particular constraints,
- reduce user's required interaction

In developing the architecture, we implemented and deployed some context-aware applications. Thereby we were able to proof our concept in practical use. Apart from satisfying the main design goals our architecture revealed three distinguishing features. Firstly, it is applicable to many context-aware scenarios, which outlines its general-purpose character. Secondly, it detaches context handling from service matters. By this separation of concerns services can be added without affecting context processing. Respectively, context handling may be manipulated without affecting underlying services. Thus, our design contributes to a simplified management. Finally, our work contributed to a new understanding of context. We do not only consider context information for deducing the current situation of an entity [8], but also *context constraints.* Here, context constraints are utilised to govern service selection and execution processes. Anyway, the concept of context constraints may be applied to many other problem areas.

2. DESIGNING A CONTEXT-AWARE SERVICE ARCHITECTURE

A user's device should act as a *portal* to its surrounding computing environment; it is an access point to the service and data space [5]. In our approach we

refine this idea: The device acts as a *mediator* for expressing service needs to the environment; service needs result from the user's current context (or task). Our work mainly focuses on representing and interpreting the user's service needs.

As a solution for communicating service needs we endeavoured a uniform document format: *Context-Aware Packets* (CAPs). CAPs allow expressing service needs on a high abstraction level without knowing specifics about services available in the environment. CAPs are created by the user's device and put in the network; inside the network the CAP gets evaluated. The evaluation a CAP results in selection and execution of services fitting to the specified service needs. Service needs are expressed by *context constraints*, which describe the situation and circumstances under which the user intends to use a service.

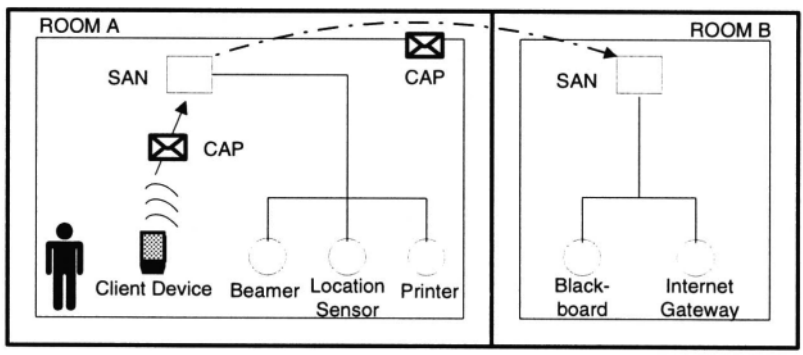

Figure 1. Testbed setup: Service Access Nodes control the services and sensors for a room. The user's device emits a CAP document via wireless link to the SAN in the room she stays. Then, system looks for a proper service and executes it.

Figure 1 depicts CAP evaluation from a networking perspective: A user injects a CAP to a local service access node (SAN). A SAN acts as a service proxy controlling the services available in its domain. For our testbed a domain always corresponds to a room and its hosted services (e.g. digital blackboard). The SAN evaluates the CAP; evaluation considers two phases: *selection* and *execution.* In the selection phase, the SAN checks whether the CAP is related to a service in its domain (room), or not. If not, the CAP is routed to a SAN meeting the needs; routing choices are governed by the CAP's embedded context constraints. In the second phase, a selected service is to be executed. Here, context constraints control the execution of a selected service.

In order to govern choices based on context constraints it is mandatory to determine the actual context, e.g. where is a user, or what services are available in the current environment. To find out, the SNA may read out sensor signals (e.g. location sensor) or check a service repository for available service interfaces.

Comprising, the architecture – using CAPs – controls the selection and execution of services based on context constraints, which reflect the user's service needs. The selection process initiated by CAP chooses a fitting service and its location (in which room); the execution process may bind services to context triggers, which control temporal issues.

2.1. The Concept of Context-Aware Packets (CAPs)

Technically, a CAP constitutes a kind of *remote procedure call* (RPC) [11] based on a document-based approach as XML-RPC [12]. CAPs feature some differences from the classical RPC concept: the receiver of the call is determined after it is issued. Specialised network nodes, SANs, route the CAP to its receiver based on its embedded context constraints. Another difference relates to the execution, calls in respect of CAP may be deferred, or even be autonomously repeated. Further on, CAP processing may require user interaction. *Interactive* CAPs (see section 2.4) ask the user to confirm the execution of a pre-selected service. Or in the case of problems the user may be notified. And finally, CAPs may be nested, see 2.3 for details. Provision of these non-RPC features requires a novel representation scheme, which is described in the following.

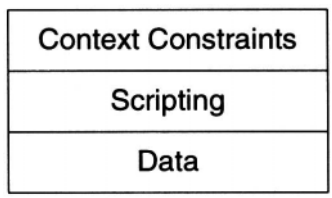

Figure 2. Context-Aware Packet (CAP)

As Figure 2 depicts a CAP document is organised into three parts: *context constraints*, *scripting*, and *data*. **Context constraints** play a major role inside CAP; as already mentioned they are used to mediate a user's service needs. The context constraints' semantics and representation will be explained in the subsequent section.

The **data** section provides data to be processed by the selected service. The data format is not restricted to one specific format, it relates to the service in mind. Hence, a beamer service might expect PowerPoint formatted data, and an Internet photo album might expect JPG. Data may be directly embedded in the document or referenced, e.g. by HTTP link. Referenced data allows implying dynamic data when executing a service, e.g. always display a user's day schedule when she enters the office.

The **scripting** section allows representing simple scripts executed on a selected service in order to embrace more complex semantics, which cannot be

captured by context constraints or data section. In our current prototype implementation we did not make any use of the scripting feature. Scripting was added for completeness, so future applications of CAPs will not require any modification to the CAP format.

2.2. Context Constraints

This section outlines the semantics and representation of context constraints. Inspired by control theory [13], context constraints can be interpreted as a *set point*. The *actual value* is referred as a *context configuration.* A context configuration reflects the actual context of a set of entities; the value of a context configuration results from sensor measurements. In order to control service selection and execution processes, the system compares the context constraints to the context configuration. If the actual context configuration meets the context constraints, the CAP's data is applied on a specific service. In this moment the actual situation (deferred by the context configuration) exactly matches the requirements expressed by the context constraints.

If the context configuration does not meet the context constraints, the system may wait for the required situation to occur. Or it detects that the exposed context constraints cannot be fulfilled at all, e.g. the CAP relates to a service in another environment (room), hence the CAP has to be re-routed.

Context constraints are phrased using the following primtives: *abstract entities*, *relations*, and *events.* The primitives are described in the following paragraphs.

Abstract Entities

Abstract entities are categorised into three types: *Actors*, *Abstract devices*, *Items.*

Any user interacting with the system is an actor. Abstract devices represent devices, which offer services to their environment. These can be all kinds of devices from an abstract perspective, which means not related to concrete devices (e.g. printer luther in room D2). Finally, items denote passive elements. In contrast to abstract devices they do not offer services but are equipped with sensors, such that the item's state can be retrieved and used for denoting context triggers.

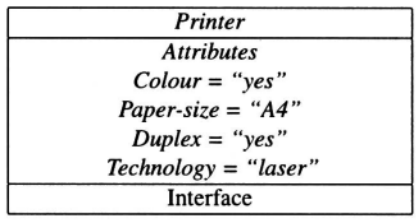

Figure 3. Printer Entity.

Entities are described by attributes. During the evaluation of context constraints, these attributes are used for mapping abstract entities to concrete units, which serve the requirements expressed by the entities' associated attributes. For example, specifying a colour laser-printer as an abstract device and "applying" it to an office returns an address to a suiting printer, if there is one. Chapter 3 shows how this functionality was implemented by inter-operating with a service discovery protocol. Figure 3 shows a sample entity, which uses common technical attributes. The concept of CAP does not restrict attributes to any types, but it has to be considered that the attributes are still applicable. Each CAP may relate to multiple entities.

Relations

Generally, a **relation** describes the dependencies of entities from each other in a spatial or temporary manner; a relation exists of a set of entities. One sample relation, *inRoom*, describes spatial information, such that the members of this relation have to be in the same room. Thus, starting with entities as atomic units, relations allow gluing those together for modelling dependencies. Relations constrain the selection of a desired service. Referring to the laser-printer example of the previous paragraph, an *inRoom* relation containing the printer entity and a user entity constraints printer selection to the same room as the user stays.

Events

Generally, events indicate actions or occurrences detected by a program. Usually, events can be user actions, such as clicking the mouse, or pressing a key. A system can subscribe to certain events and can respond to their occurrence.

An **event,** in the context of CAP, describes a trigger, which either activates (positive trigger) or aborts (negative trigger) the execution of a CAP initiated service call. Events report actions and occurrences detected by sensors, which are modelled by *item entities.*

Event conditions are represented by logical condition-expressions, which can be either ground or combined by logical operators (AND, OR). Figure 4 shows a condition in the CAP's XML format. Our representation scheme adopts to the trigger concept for policy-based management described in [15].

```
<AND>
   <condition subject = "day" object = "monday" type = "="/>
   <condition subject = "user\_location" object = "at_work" type = " ="/>
</AND>
```

Figure 4. "Triggering a reminder if the user gets to work on Monday"

Regarding semantics, the event concept steps out the service selection, it rather allows to condition execution of a selected service.

2.3. Composing Services

The latter sections outlined the structure of a single CAP; this section explains how multiple CAPs may be composed by *nesting*. In particular, this is useful for re-transferring CAPs in case of error, or for tunnelling through converters.

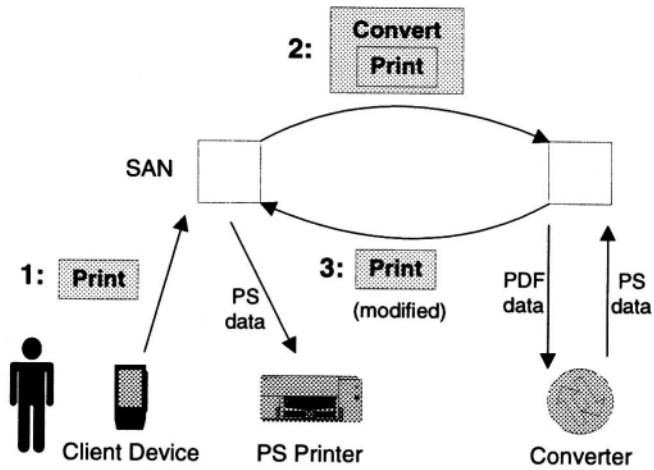

Figure 5. Tunneling a Print-Job

CAPs are nested, by putting one CAP into the data part of another one. Multiple nesting is also feasible. This recursive structure allows expressing cascaded use of multiple services. For example, a user wants to print a document in PDF format, but the printer in the current environment only accepts Postscript format. Instead of signalling the user that the demanded service request cannot be fulfilled, the system autonomously sends the document to a conversion service. There it is converted to the wanted format (Postscript) and is send back to the printer where the job is finally done.

For implementing this scenario (see Figure 5), the receiving service access node (SAN) wraps the incoming CAP, the outer CAP is directed to a conversion service. The SAN in the domain of the conversion service processes the nested CAP considering the context constraints of the outer CAP and data of the inner CAP. Hence, the data gets converted. Finally, the SAN removes the outer CAP and updates the data of the original CAP; the resulting CAP gets evaluated as usual.

2.4. Example: A CAP for local Printing

This section briefly discusses an example and explains vital CAP elements not covered by the previous sections. Figure 6 depicts a sample CAP document for printing a document locally (in the same room as the user stays). The begin of the document indicates the sender, the receiver and names the wanted action:

```
<CAP name="printing_locally" from="Florian"
                put_to="Printer"
        duration="one-shot" interactive="no">
```

Due to the fact that a CAP document may include a number of abstract entity descriptions the "put_to"-directive is used for labelling the *service entity.* The CAP's embedded data is applied to this service entity, here it is the printer entity. *Duration* signals if the CAP is only executed once or multiple times; *one-shot* denotes single execution, applying *multi-shot* a CAP is activated repeatedly.

```
...
<CAP name="printing_locally" from="Florian"
put_to="Printer" duration="one-shot">

<ContextC>

  <Entity type="abstractDevice" name="Printer">
      <attrib lextype="colors">greyscale</attrib>
      <attrib lextype="papersize">a4</attrib>
      ...
      <interface>
       <put name="lpr">
        <param lextype="lpr_filter">no</param>
        <param lextype="MIMEtype">
              stream/postscript
        </param>
        ...
       </put>
      </interface>

</Entity>

<Entity type="Person" name="Florian">
    <attrib lextype="date">02.02.2001 13:45</attrib>
    <attrib lextype="role">guest</attrib>
    ...
</Entity>

<Relation name="inRoom">
    <arg>Florian</arg>
    <arg>Printer</arg>
</Relation>

</ContextC>

<Data type="application/postscript">
  <![CDATA[ %!PS-Adobe-2.0
    ... ]]>
</Data>

<Scripting>future use</Scripting>

</CAP>
```

Figure 6. Example CAP

The *interactive* directive is a toggle for the CAP interaction facility. If the CAP is interactive, service execution has to be confirmed by the user. As can be seen in Figure 6 entities are optionally attributed by an *interface* section, which firstly describes if the entity is a data sink or source, and secondly it denotes the entities supported data types for exchange.
A data sink is indicated by a *put interface*, a data source is indicated by a *get interface.* As outlined in the previous section, a single service entity may

provide both interface types, put and get. In case of the conversion service data is first written and consequently converted data is read back.

The *relation* element embraces entities belonging to a specific relation, here *inRoom.*

The complete XML DTD can be seen in [16].

2.5. CAP Life-Cycle

As outlined in Section 2.2 processing of CAPs and their associated services is controlled by context constraints. For managing and tracking the control process of multiple CAPs floating the system, each CAP always adopts to a discrete state. Figure 7 shows a CAP's main processing states and possible transitions. During its lifetime a CAP may traverse five different states: *Find*, *Wait*, *Ready*, *Talk To* and *Terminated.*

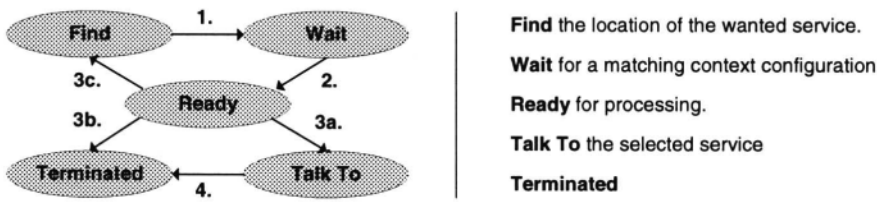

Figure 7. CAP Life-Cycle

Find is the initial state the CAP is to be routed. When the CAP reached its final destination it adopts to *wait*, the system waits for a situation that matches the CAP's embedded context constraints. If there are no outstanding obstacles regarding service execution the CAP switches to *Ready*. *Talk To* relates to actual service execution. And ultimately, CAP processing is *terminated*, after execution or abortion has been performed.

3. IMPLEMENTATION

This chapter details the architecture of CAPEUS and its constituting components. Further on it describes technical features of our implementation.

3.1. CAPEUS Architecture

The overall architecture, CAPEUS (Context-Aware Packets Enabling Ubiquitous Services), is depicted by Figure 8. The left side of the picture relates to components running on the user's device. In brief, the CAP Organiser produces

CAPs on demand of user applications; the user model [17] provides personal preferences for service use.

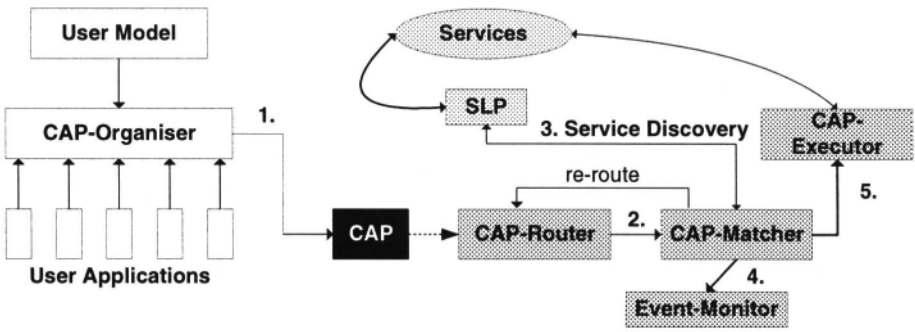

Figure 8. CAPEUS Architecture Design

The right side relates to components running on the service access node (SAN), which was the focus of our work: CAP Router, CAP Matcher, CAP Execution, Event Monitor, and SLP. The function of the SAN's components is described in the following sub-sections.

3.1.1 CAP Router. If a SAN receives a CAP via its network interface it is directed to the CAP router. The router component has mainly two tasks:

1 managing logical connections,

2 forwarding CAPs to other SANs based on symbolic attributes;

3 passing the CAP to the CAP Matcher.

We assume that network connections do only exist a short period of time, as featured by emerging networking standards, which offer ad-hoc peer-to-peer connectivity, e.g. Bluetooth [18]. Typically, a user's device establishes a wireless connection to its local SAN; then the CAP is transferred, and subsequently the network connection is closed.
But in some cases CAP execution may require interaction with the user, for signalling error, or successful service execution. Or in the case of *interactive* CAPs the user is asked if he really wants a service to be executed (when all context-constraints are evaluated valid). Hence, the router manages address data of the CAP sender, so it is possible to connect to her later. If a CAP is completely processed the user's associated data is deleted.

The second task relates to forwarding CAPs based on symbolic attributes, e.g. "Room D2, Building 10", "where the user is", or "to a proximate room". For implementing this functionality we endeavoured a hierarchy of routers as depicted in Figure 9.

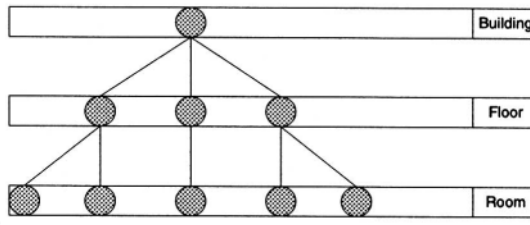

Figure 9. Hierarchy of CAP Routers

For routing a CAP to its destined environment (here room), the CAP is first routed to a floor level router. If context constrains do not allow processing the CAP in a room in this floor, the CAP is routed to the next higher level (building). Hence, the search domain is widened to the complete building. Obviously, this approach does not scale well, but it was sufficient for demonstrating the concept. Particularly, it was easy to express vicinity, e.g. room in the same floor.

Besides, a more sophisticated solution for routing messages based on attributes can be found in [19]; or a hierarchical lookup system for locating mobile objects is described in [20].

Finally, the router passes the CAP to the *CAP Matcher.* From the router's perspective, the matcher checks if the service is related to the current environment, and if it can be executed there. Otherwise, the CAP is send back to the router with additional addressing attributes.

3.1.2 CAP Matcher. Briefly, the *CAP matcher* compares the context constraints with the actual measured context configuration. If the context configuration matches the context constraints then the CAP's data is passed to the *CAP Executor,* which finally executes the service. Otherwise, the matcher calculates the *divergence*; in our context the divergence denotes *what* the matcher prevents from selecting or executing a specific service. For instance, if the matcher compares the CAP's context constraints to the actual context configuration and finds out that the CAP is related to another service domain, then the resulting divergence refers to location. Hence, the matcher may induce re-routing of the CAP. Altogether, the divergence is used to control the actions of the matcher in accordance to context constraints.

The resulting divergence may relate to *location*, *user interaction*, *wrong interface*, *trigger, equal,* or *general error.* Table 1 lists possible differences and associated actions. Location is sub-divided into two categories: *vicinity*

and *other location*. *Vicinity* instructs the router to send the CAP to proximate domains for possible execution. *Other location* the CAP is related to another domain and has to be routed there.

Table 1.

Divergence		Component/Action
Location	Vicinity	Router: try proximate domains
	Other location	Router: route to specified address
User Interaction		Router: indicate state, user confirmation (interactive CAPs)
Trigger		Event Monitor: wait for matching situation
Wrong Interface		Matcher: convert data, nested CAP (see Section 2.3)
Equal		Executor: apply data on selected service
General Error		Router: Error message to user; Matcher: abort processing

In the following it is outlined how the matcher parses CAP documents: firstly the matcher evaluates the *relations* for selecting a suiting service. The matcher applies the CAP's embedded triggers for governing the execution of the previously selected service.

Evaluating Relations

The matcher represents nested relations as tree, nodes represents relations and leaves represent abstract entities. Figure 10 shows a simple example, it is used to select the cheapest printer in the same room as the user. As already, mentioned the *inRoom* relation requires entities to reside in the same room. The *min(X)* relation selects among multiple entities the one with minimum value for attribute *X*.

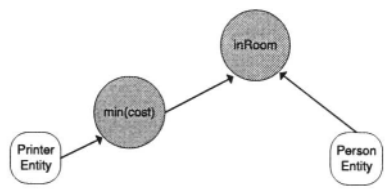

Figure 10. The cheapest printer in the same room as the user stays.

The relation tree is evaluated bottom-up; hence the matcher first evaluates the *min* relation and thereafter the *inRoom* relation. Figure 11 depicts the matching algorithm for relations. When applying the algorithm the set of possible candidates (service entities) gets stepwise reduced. Lastly, a set of suitable service entities remains; one of them is selected randomly.

For service discovery the system compiles a request based on the attributes of the abstract entity; in turn the service discovery returns all service entities which match the attributes. Sometimes it may be necessary to convert the attributes to a format compatible to the chosen service discovery protocol. Hence, we implemented a transducer component, which converts an attribute schema to a schema in question.

1 Enter next relation (bottom-up).
2 Execute service discovery for all abstract entities of the relation.
3 Evaluate relevant entity attributes by applying the relation.
4 Discard service discovery results, which do not fulfil the relation.
5 Continue with 1 until complete tree is processed.

Figure 11. Matching Algorithm for Relations

We deployed service location protocol SLP [3] for service discovery, for supporting other service discovery systems, like UPnP [22], JINI [2], or Salutation [23], we applied the strategy pattern [24]. Hence, the transducer component can be exchanged without affecting other components.

Evaluating Events

After evaluating the relations, the desired service is selected. In contrast, *Events* control the execution of this service.

For each event defined in the CAP the *matcher* spawns an individual *Event monitor*. An Event Monitor senses the environment for the event in question. If the event occurs the matcher is notified. As described in Chapter 2, events may be combined by logical operators. Special care has to be taken for evaluating conjunctive operators ("AND"), because the system has to decide whether two events are considered to be coeval or not. Hence, two events are considered coeval if they occur in a time period shorter than a defined threshold. For most applications a threshold less than a second seemed to be reasonable.

If all events evaluated true the event monitors get killed. Consequently, the matcher hands the CAP's data and the address of the selected service to the *CAP Executor*, which ultimately executes the selected service.

3.1.3 CAP Executor. The *CAP Executor* embodies the interface from CAPEUS to extern services. It is supplied with one single service location, which was determined by the *Matcher*. Further on, it accesses the CAP's embedded data. Services that do not comply to the executors expected streaming interface may be wrapped; as in CyberDesk [25] we could apply observable APIs [26]. In future projects, the executor will also process the scripting part, for facilitating more complex semantics.

3.2. Prototype

The CAPEUS prototype was implemented in JAVA [27] and is based on the principles discussed in the latter chapter. The objective is that the prototype will be suitable to implement and also validates the main concepts described above. In addition, to the features discussed and implemented within this project CAPEUS is meant to be used and refined in various future projects. CAPEUS was implemented and tested in conjunction with the DWARF project [28] on augmented reality [29]. The DWARF prototype provided several features, which could be used for demonstrating CAPEUS in practical use. On the other hand CAPEUS supplemented DWARF by adding spontaneous use of environmental services, whereby selection and execution of services is governed by pre-defined context constraints. In the current implementation DWARF offers besides of WaveLan [30] and Bluetooth [18] connectivity also location sensing. Up to now the system provides GPS for outdoor environments. It is planned to use RF tags for indoor environments.

We implemented a few context-aware applications for demonstrating our architecture in practice, two of them are briefly described in the subsequent paragraphs.

Context-Aware Notes

In analogy to the stick-e framework [31] this application mimics the post-it metaphor. If the user takes a note, it is tagged by the current available context information, particularly the location. If the user comes back to this place in future, her handheld[1] will display the note. The application also allows creating public notes, visible for anybody.

We implemented this application just by creating a repository of *multi-shot* CAPs. Each reflecting a user's note, the CAP's context constraints reflect the note's related context information. For personal notes the service access node (SAN) runs on the handheld. Anytime a note's related context occurs the associated CAP triggers a display service on the handheld.

For public notes the CAP is transferred to the location's responsible SAN. In this case the note is displayed to any user entering the location.

Task Aligned Local Service Use

We implemented an application for local service use, considering the user's task, here document viewing. In particular it is a document viewer running on a laptop computer. To any document, the viewer offers service options related to the currently displayed document type. For instance, viewing a text document the system offers the following service options: *print, blackboard, and note*. Each service option is related to a specific CAP: *print* issues a local

[1]For the prototype we used laptops with WaveLan connectivity.

print job as described see Section 2.4. *Blackboard* is similar to *print* it displays the document on a local digital blackboard. And finally *note* reverts to the context-aware note service described in the previous section.

In case of the print and blackboard service the viewer creates associated CAPs just by filling up the data part; the context constraints reflect pre-defined preferences of the user. The *note* is handled as described in the previous paragraph.

Due to the fact that the system pre-selects service options related to a specific document type, and transparently handles all interaction regarding remote service use (context-aware selection and execution), the system significantly reduces required attention by the user. Additionally, this technique allows saving *display estate* by simplifying the menu structure, what is crucial to handheld computing.

4. RELATED WORK & FUTURE DIRECTIONS

Work on service provision for mobile users as [34] or [35] are mainly based on the idea of service adapters. Using service adapters remote services appear local to applications running on the mobile device; service adapters mainly handle intermitted network connections and pass method calls to a remote service. Additionally, the mobile device can detect the service peers in its proximity, as featured by Bluetooth [18].

These approaches are similar to our architecture in the way that they enable the use of environmental services to a mobile user. But in contrast to CAPEUS, service use is restricted to services in coverage of the wireless network. Thereby, the spatial structure of environments is not reflected. Further on, CAPEUS enhances plain service use by supporting selection and execution processes taking in account arbitrary context information, not only location. Service invocations are modelled by documents, so it is not required to implement an individual service adapter for a specific service.

Our work profited from the Stick-e framework [33], which introduced the concept of virtual post-it notes; the implied context trigger mechanism influenced our design. The representation of service interfaces was inspired by the work of Hodes and Katz, see [5].

And finally, our work was fostered by work done on context-awareness [7,21,10,31].

Future work will focus on the following issues: Firstly, we will continue research on automatic creation of CAPs considering a user model, which reflects the user's preferences. Secondly, we will add support for other interface types. For the moment CAPEUS only handles streaming interfaces. Thirdly, scripting is to be supported in future versions of CAPEUS. Finally, we will consider security in order to protect the user's privacy.

References

[1] G. Abowd, J.P.G. Strebenz. Final Report on the Inter-Agency Workshop on Research Issues for Smart Environments. IEEE Personal Communications, October 2000.

[2] Jini(TM), 1998. http://java.sun.com/products/jini.

[3] C. Perkins. Service Location Protocol White Paper, May 1997.

[4] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An Architecture for a Secure Service Discovery Service. In Proceedings of MobiCom '99, Seattle, WA, August 1999.

[5] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussmann, D. Zukowski. Challenges: An Application Model for Pervasive Computing. In Proceedings of MobiCom 2000, Boston August 2000

[6] T. Hodes and R. Katz. A Document-based Framework for Internet Application Control. In 2nd USENIX Symposium on Internet Technologies and Systems, October, 1999.

[7] G. Nelson. Context-Aware and Location Systems. PhD thesis, University of Cambridge, Computer Laboratory, Cambridge, UK, January 1998.

[8] Anind Dey and G.D. Abowd. Towards a Better Understanding of Context and Contex-Awareness. Technical report, GeorgiaTech, 1998.

[9] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In CHI'99, Pittsburgh, PA, US, 1999.

[10] Christos Efstratiou. Developing a Context-aware Electronic Tourist Guide: Some Issues and Experiences. Technical report, Department of Computing, Lancaster University, Lancaster, LA14YR, U.K., 2000.

[11] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification Version 2, June 1988. RFC 1057.

[12] UserLand Software, Inc. XML-RPC. http://www.xmlrpc.com/

[13] Brogan, W.L., Modern Control Theory, (Prentice Hall, NJ, 1991).

P. Pin-Shan Chen. The Entity-Relationship Model-Toward a Unified View of Data. ACM Transactions on Database Systems, 1(1):9,1976.

[14] Morris Sloman and Emil Lupu. Policy Specications for Programmable Networks. In First International Working conference on Active Networks (IWAN'99), Berlin, June 1999.

[15] F. Michahelles. Designing an Architecture for Context-Aware Service Selection and Execution. Diploma Thesis. University of Munich, 2001.

[16] Michael Samulowitz. Designing a Hierarchy of User Models for Context-Aware Applications. Workshop on Situated Interaction in Ubiquitous Computing. CHI 2000, The Hague, April 2000.

[17] The Bluetooth SIG. WWW. http://www.bluetooth.com/v2/document.

[18] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The Design and Implementation of an Intentional Naming System. In Proceedings of the ACM Symposium on Operating Systems Principles, Charleston, SC, 1999.

[19] James Weatherall and Andy Hopper. Predator: A Distributed Location Service and Example Applications, 1999. In proceedings of Cooperative Buildings 1999, Springer-Verlag Lecture Notes in Computer Science.

[20] S. Fels, S. Sumi, T. Etani, N. Simonet, K. Kobayashi, and K. Mase K. Progress of C-MAP: A context-aware mobile assistant. In Proceeding of AAAI 1998 Spring Symposium on Intelligent Environments, March 1998.

[21] Universal Plug and Play Device Architecture. www.upnp.org

[22] Salutation, 2001. http://www.salutation.org.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[24] A. Dey, G. Abowd, M. Pinkerton, and A. Wood. CyberDesk: A Framework for Providing Self-Integrating Ubiquitous Software Services. Proc. ACM UIST'97, 1997

[25] A. Wood. CAMEO: Supporting Observable APIs. Position Paper for the WWW5 Programming the Web Workshop, April 1996.

[26] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification. Addison-Wesley, Reading, Mass., 2 edition, June 2000.

[27] P. J. Brown. The Stick-e Document: a Framework for Creating Context-aware Applications. In Proceedings of EP'96, Palo Alto, published in EP, January 1996.

[28] Martin Bauer, Asa MacWilliams, Florian Michahelles, Christian Sandor, Stefan Rib, Martin Wagner, Bernhard Zaun, Christoph Vilsmeier, Thomas Reicher, Bernd Brügge, and Gudrun Klinker. DWARF: System Design Document. Internal Report. Technical University Munich.

[29] Ronald T. Azuma. A Survey of Augmented Reality. August 1997.

[30] Airport Wireless Technology, http://www.apple.com/airport/.

[31] P. J. Brown. The Stick-e Document: a Framework for Creating Context-aware Applications. In Proceedings of EP'96, Palo Alto, published in EP, January 1996.