RZ 3407 (# 93497) 03/04/02
Computer Science 13 pages

Research Report

Composite Profile Information

Carl Binding, Reto Hermann, Andreas Schade

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

Composite Profile Information

Carl Binding, Reto Hermann, Andreas Schade

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

This paper reviews existing techniques and standards to capture and transport capability and preference information, so-called *profiles* for end-user devices with special emphasis on limited footprint, mobile, devices. It proceeds to present a model for composing profile information based on default values and profile differences applied to end-user device originating capability and preference profiles. An account on the implementation of a middleware software component to handle and manage composite profile information highlights some of the implementation issues and provides initial performance indications.

1 Introduction

The most popular information technology architecture currently is based on the *browsing model* of the internet application architecture. An end-user requests information by using his user agent, the so-called *browser*, which acts as a rendering engine for mark-up documents containing layout directives interspersed with the information content. Mark-up documents are generated by application servers, typically accessed via the HTTP protocol [7].

The markup content generation process is controlled through various parameters: the name of the information resource being addressed (encoded in its unique resource identifier (URI) [1]), the URI parameters, the application logic associated with the URI[2], and possibly additional HTTP headers transmitted with the request.

Essentially though, the output generated by the application is a function of parameters passed with the request and some application state.

Figure 1 illustrates the application architecture. The end-user device's user-agent issues a request for information (i.e. an HTTP GET request) which is received and processed at the origin-server. The generated content is returned to the client as an HTTP response body and rendered by the client device user agent.

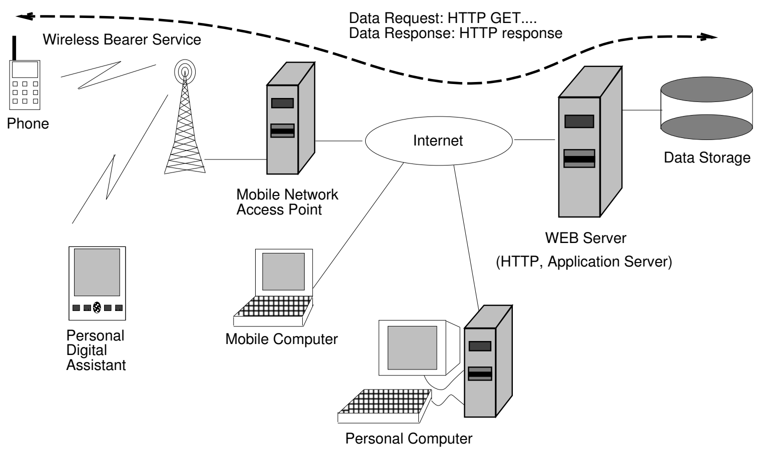

Fig. 1. Internet application architecture

In the past, most internet applications were accessed through PC or computer workstation based browsers which typically operate on a large screen attached to a computer of considerable computing power (in terms of memory, disk, CPU, etc.). With the growing capabilities of wireless wide-area networks used for voice based telephony and data traffic to smaller, personal, and portable devices, the *mobile internet* has become of increasing commercial interest as it

[2] Such logic can be encoded as a scripting language such as PERL, a conventional programming language such as C or Java [16] or a mix of mark-up and scripting languages such as the Java Server Pages (JSP) formalism [15].

allows to support an ever increasing number of end-user terminals. The success of the Japanese *i-mode* system [6], launched by NTT DoCoMo in the late nineties, has been spectacular. Given the appropriate wireless infrastructure, business models, and end-user devices, we thus can expect similar services to appear in other geographies with comparable success.

Whilst the overall application architecture for an internet and a mobile internet application does not differ fundamentally - indeed both using the same lower layer transport protocols and similar mark-up languages - the presentation to the end-user will have to differ because mobile end-user devices will remain smaller and thus limited in terms of input-output capabilities compared with traditional personal computers. (The size is driven by two factors: portability and power consumption.) Hence, applications will benefit from the ability to adapt the generated content to optimally fit the capabilities of the end-user device and the preferences of the end-user himself.

As discussed above, the content generation process is driven by parameters associated with the request for information. Hence, if we augment such requests with additional information to describe the capabilities of the device and the end-user's preferences, we enable the application to exploit these and generate better suited mark-up output to be rendered on the end-user device.

The requirements for handling of such capability and preference profile information can be summarized as follows:

1. *Expressiveness*: an end-user compute entity must be enabled to express its capabilities and the user's preferences in a concise and non-ambiguous way.
2. *Transport syntax*: the profile information must be transmitted to the content generating origin-server in a space efficient and tamper proof way.
3. *Handling*: the origin-server must be enabled to handle the information contained in a preference profile through an appropriate mechanism, i.e. through a programmatic interface to query the profile's values.
4. *Efficiency*: the transport and handling of profiles must be efficient as each information request can be associated with a preference profile.
5. *Aggregation*: various processing nodes in the network between end-user device and origin-server may augment a request. For example, if a content transforming node on the path supports additional transformations which are beyond the capabilities of either the end-user client device or the origin-server, this information can be of use to the origin-server. For example, the origin server may include content which is eventually transformed by some network node on the path between origin server and client device.
6. *Manageability*: the entire profile handling system must be manageable.

The purpose of this paper is to review the state-of-the-art in composite preference information (CPI) handling and to introduce a formally sound framework for profile aggregation. Section 2 reviews existing work on capabilities and preference profile handling. A mathematical formulation of profile processing for a specific data model is given in section 3, including a model for profile schema unification. Our implementation, inclusive some performance indications, of a standardized profile environment is described in section 4. A summary and conclusion of our work closes the paper with section 5.

2 Client Capabilities and Preference Profiles: a Survey

Current internet technology does not use preference information on a widespread scale. In part, this is due to the limited possibilities of forwarding preference profile information with an HTTP request. The HTTP/1.1 standard [7] supports a limited set of header fields of which only the header-field *User-Agent* allows to identify the end-user's browsing environment[3]. Additional preference information can be conveyed via the various *Accept* header fields, but the range of capabilities which can be described through these means remains limited.

Common practice thus is to statically associate a device profile with the device's *User-Agent* information. This profile is then accessed by the origin-server based on the HTTP request's header value.

A more dynamic approach is advocated in the model of the *client capabilities and preference profile* (CC/PP) [10]. It proposes composite preference information grouping device capabilities and user preferences into a set of *components*. For each component, a possible set of *default* values is indicated through the inclusion of a URL *reference* to such default values.

Each component furthermore contains one or more *properties*. Every property defines its name, its type, and possibly a set of values (for enumerations, for example).

A profile's components and their properties are specified in the profile *schema* or *vocabulary*. Thus, for different applications distinct CC/PP vocabularies can be defined.

An XML based syntax, the *resource description framework* (RDF) [11] is used to externalise a CC/PP compliant profile and associated default profiles. Proposals to augment HTTP headers to convey profile information have been published and - with some modifications - have been adopted by at least one standardization body (the WAP Forum) [13, 19, 22]. These transport syntaxes propose a split of profile information between the *profile* and so-called *profile differences*. The profile header field contains a reference to the actual profile value, expressed as an URL. In addition, difference values to the base profile conveyed with the HTTP request inside some *profile-diff* header are referenced via indexing. A checksum over the profile difference is included in the profile header to guarantee the integrity of the profile differences.

From an implementation point of view, the CC/PP proposal suffers from various shortcomings:

1. RDF syntax [11] is ambiguous and thus hard to parse properly.
2. There is no formalism to specify profile schemata. The proposal lacks expressiveness regarding the definition of comparison operations applied to properties and the introduced property types and their formats are not mandatory. The notation also does not clearly state which properties are mandatory, which can be omitted etc.
3. The CC/PP model leaves the determination of a property's type, possible relations between property values, their syntax, and their meaning to the application. For example, there are no rules for spelling of enumeration of

[3] The values of this header field are however *not* standardized.

literal values (e.g. capitalization, white-space) or the format for numeric values. Similarly, defaulting for omitted property values is not specified.

The WAP Forum has based its *User Agent Profile* (UAProf) [22] work on the CC/PP and RDF framework. These have been extended with a specific transfer syntax for profile and profile differences, a schema definition, and resolution rules for property default and difference values. In addition to the Wireless Application Protocol itself, the proposed *m-services* initiative also requires adoption of the UAProf standard [8].

Six components are defined in the UAProf vocabulary to describe the capabilities and preferences of a mobile end-user device:

1. *hardware*: These properties describe the hardware features of the WAP device, such as screen size, pixel dimension, keyboard features etc.
2. *software*: Indicates which operating environment is present on the mobile station. For example, the version of a Java Virtual Machine (if any), the set of accepted character sets and languages, etc.
3. *user agent*: Describes the mobile station's browser environment, i.e. are frames supported? which version of HTML is supported? are Java scripts or applets supporte? etc.
4. *network*: Describes the mobile network environment, for example which bearers are supported respectively currently enabled.
5. *WAP characteristics*: Reflects the WAP environment specific features of the device. Which version of the standard? Which WMLScript [23] libraries? Which WAP pictograms can be rendered on the device? etc.
6. *push environment*: Identifies WAP Push [21] related characteristics of the device, such as the number of push messages the device can store, which push applications it supports, what character sets and content-types for push content the device can handle etc.

The property types comprise numerics (integer), booleans, dimensions[4], and (string) literals which are also used for enumerated values.

In addition to these base types, multi-valued types are also provided. *Bag*s indicate an unordered, multi-valued set, *alternatives* describe a choice from multiple values, and *sequences* provide an ordered set of values.

The UAProf standard includes a transport syntax for transporting profiles and profile differences. Profiles references, i.e. their URLs, are contained in an *x-wap-profile* HTTP header and profile differences as RDF/XML data in *x-wap-profile-diff* headers. The *x-wap-profile* header refers to the profile differences through an index and includes a checksum (MD5) of the difference value to guarantee its integrity.

Other transfer syntaxes are based on RDF/XML documents embedded in MIME multi-part messages [20] or on a distinct set of HTTP extension headers [21].

The UAProf property resolution rules allow to prescribe following behaviour:

1. *Locked*: a property value can be modified once at most, subsequent difference values have no effect.

[4] A two-dimensional metric.

2. *Override*: difference values override previously established property values in the order of differences application.
3. *Append*: applies only to lists (i.e. bags, alternatives, or sequences) and implies concatenation of property values to extend a list of such values.

Note that the UAProf schema – in the absence of a CC/PP defined formalism – uses an ad-hoc, XML based formalism in which some of the schema definitions are simply embedded in XML comments! Further shortcomings of the CC/PP framework and the UAProf schema in particular are:

1. No syntax rules for literal enumerations. E.g. white-space, capitalization, syntax for numbers, hyphenation etc. are not clearly formulated.
2. No fully machine readable schema definition to allow automatic verification of profiles against their schema.
3. Missing component type compatibility rules for profile aggregation.
4. Relation functions defining an equivalence relationship between property values to indicate whether a given property value p_1 is equivalent, inferior, or superior to some other property value p_2. For example, does a Java Virtual Machine property value of *SUN_JVM_1* indicate similar, superior, or inferior capabilities of a Java Virtual Machine *MS_VM_J13*?

The literature on usage or foundations of UAProf is still nascent [3]. Butler reports on a Java implementation of the UAProf environment [4]. Its functionality is similar to our implementation, however no formal definition of the profile resolution process is provided. There are no performance figures and potential extensibility of the Jena framework is not described.

Potential applications to content transcoding based on usage of profiling information is discussed in [2]. The paper also describes profile aggregation based on a rules-based formalism [9] to yield a unified preference value from a set of multiple profiles. However, no formal profile composition model is given, only static preference profiles are processed, and no performance measurements are reported.

3 Aggregate Profile Model

In this section we formalize the handling of CC/PP based capabilities and preferences information. We use concise mathematical notation to describe the process of profile resolution. A further formalization allows to describe profile schemata aggregation.

3.1 Formal Model Definition

We propose a formal model to capture composite preference information, which we call the *schema* of a profile. Such schema must be machine readable for validation of profiles.

1. A schema S is defined as a set of *component* types:

$$S = \{C_j\}, 1 \leq j \leq D(S)$$

where C_j are the component types and $D(S)$ is the *dimension* of the schema and denotes the number of component types present in the schema.
2. A component type C is defined as a set of *property* types:

$$C_j = \{P_i\}, 1 \leq i \leq D(C_j)$$

where P_i are the property types and $D(C_j)$ is the *dimension* of the component and denotes the number of property types present in the component. The component type contains a special property type by which an instance of a component type (see below) can refer to its default settings.
3. A property type defines a *name*, a *resolution rule*, and a *value type* for which an ordering relationship must be specified. Property resolution rules are defined to be one of $\{locked, override, append\}$. We write $policy(P^C)$ to denote the resolution policy associated with property type P.
We shall define property type compatibility if the value type of P_i can be mapped onto the values of P_j.

Based on this type system, we define profile, component, and property instances as follows:

1. A property p is an instance of a property type P. The property has a value according to the value type as defined by the property type.
2. A component c is an instance of a component type C. A component contains a non-empty set of properties:

$$c = \{p_i\}, 1 \leq i \leq D(C)$$

Optionally, a component instance c may refer to a default component instance of the same type. This instance, which we denote by $\bar{c}$, provides the default properties for the referring component.
3. A profile s is an instance of a schema S. A profile contains a non-empty set of components:

$$s = \{c_j\}, 1 \leq j \leq D(S)$$

Using this formal schema definition, we introduce the following notation to identify the value p of a property of type P_i inside a component c_j (of component type C_j) in a profile s (of schema S):

$$p_{i,j} = P_i(c_j) = P_i(C_j(s)), 1 \leq j \leq D(S), 1 \leq i \leq D(C_j)$$

i.e. the i-th property of the j-th component of s.

3.2 Profile Operations

Default Resolution Default resolution is the aggregation of all properties of some component c and its default component $\bar{c}$. As a result of this, a new component is created in which a particular property keeps its value where provided. Alternatively it takes the value of the default component, if present, or will not be included in the resulting component.

Formally, default resolution can be denoted as follows:

$$\tilde{p_i} = P_i(\tilde{c}) = \begin{cases} P_i(c) \text{ if } p_i \in c \\ P_i(\bar{c}) \text{ if } p_i \notin c \wedge p_i \in \tilde{c} \\ \phi \qquad \text{if } p_i \notin c \wedge p_i \notin \tilde{c} \end{cases}$$

As such, default resolution is an operation over components. For a particular profile s it can be applied to all member components c_j. The resulting profile is referred to as the defaulted profile, denoted by $\tilde{s_0}$.

Default resolution is applied to the base profile s to yield $\tilde{s_0}$ as well as all *difference profiles* $s_k, 1 \leq k \leq K$ which modify the property values of $\tilde{s_0}$ in the subsequent processing step, called *difference application.* The defaulted profile differences are denoted $\tilde{s_k}$.

Note that difference profiles are optional for a given base profile s, i.e. K can be 0. In that case, profile resolution is complete after default resolution and the below step of difference application is not necessary.

Difference Application Profiles can be modified further by applying profile differences, if any. A profile difference, or profile diff for short, structurally also is an instance of the base profile's schema S. Default resolution, formalized above, is also applied to profile diffs. Hence, we now have the defaulted base profile $\tilde{s_0}$ and a non-empty set of defaulted profile diffs $\tilde{s_k}, 1 \leq k \leq K$. We regroup these profiles into a set $\tilde{S} = \{\tilde{s_k}\}, 0 \leq k \leq K$.

We denote a fully resolved property value $\hat{p}$, a fully resolved compoment $\hat{c}$, and a fully resolved profile $\hat{s}$ respectively. Computation of the property value $\hat{p}_j$ of component $\hat{c}_i$ of $\hat{s}$ can then be formalized as follows:

$$\hat{p}_{i,j} = P_i(\hat{c}_j) = P_i(C_j(\hat{s})) = \begin{cases} P_i(C_j(\tilde{s}_{k_{min}})) & \text{if } policy(P_i) = \text{locked} \\ P_i(C_j(\tilde{s}_{k_{max}})) & \text{if } policy(P_i) = \text{override} \\ \bigcup_{k=0}^{K} P_i(C_j(\tilde{s}_k)) & \text{if } policy(P_i) = \text{append} \end{cases}$$

For the *locked* case, $0 \leq k_{min} \leq K$ selects the first component of $\tilde{S}$ in which property P_i appears; the property value then becomes $P_i(C_j(\tilde{s}_{k_{min}}))$. Conversely for a resolution policy of *override*, $0 \leq k_{max} \leq K$ selects the last component of $\tilde{S}$ for which property P_j is evaluated and the property value becomes $P_i(C_j(\tilde{s}_{k_{max}}))$. *Append* resolution policy indicates concatenation of multi-valued poperty type values across all profiles in $\tilde{S}$

Schema Aggregation The formalisms for default resolution and difference application introduced rely on the assumption that the operations are performed on profiles adhering to a single schema.

In practice,however, it is the case that profile information belonging to diverse schemata S_k needs to be merged into a single composite preference information profile s_{agg} with new schema S_{agg}. Such a need can arise, for instance, because existing schema definitions evolve over time, i.e., new properties are added to a component, or new components are added to a profile. Another case is the situation, where profile schemata that have been defined by different

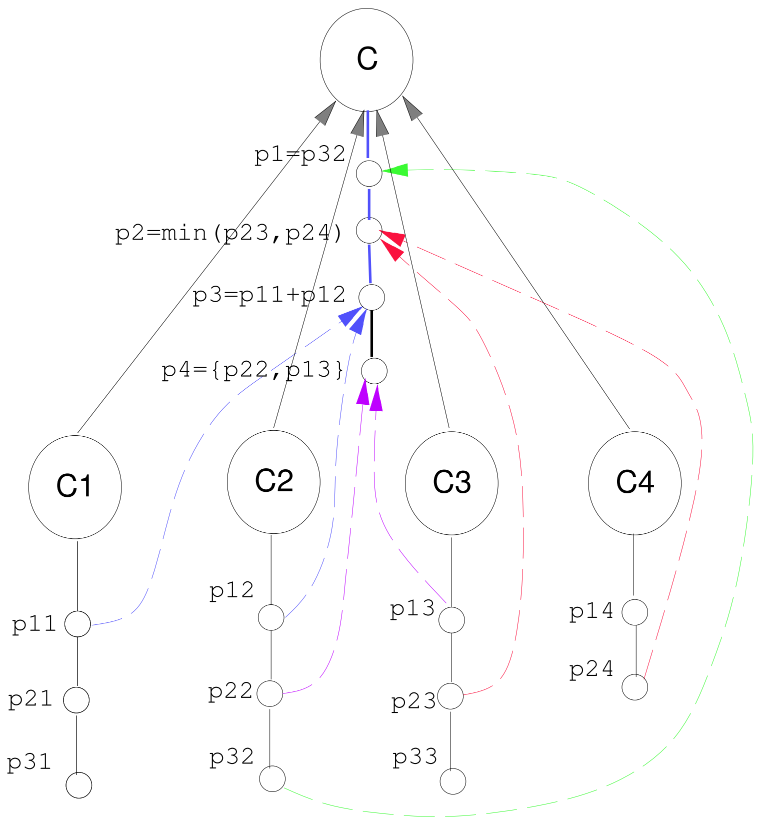

Fig. 2. Component property aggregation

organizations (e.g., standard bodies) exhibit semantic overlap and this overlap needs to be resolved into a unique set of components and properties.

Figure 2 illustrates this profile aggregation for four resolved component instances c_1, c_2, c_3 and c_4 with component types C_1, C_2, C_3, and C_4. We assume that these component types are defined by some schemata (not given here). A new component type C is created with four property types P_1, P_2, P_3, P_4 and is included in the new schema S_{agg}. The resulting profile s_{agg} contains an instance c of the (new) component type C.

The value of the properties are based on the values of the composing component instances and, for the example of figure 2, are defined as follows:

$$\begin{aligned} p_1 &= P_3(\hat{c}_2) \\ p_2 &= min(P_2(\hat{c}_3), P_2(\hat{c}_4)) \\ p_3 &= P_1(\hat{c}_1) + P_1(\hat{c}_2) \\ p_4 &= P_2(\hat{c}_2) \cup P_1(\hat{c}_3) \end{aligned}$$

(Recall that we write $p_{i,j}$ for the i-th property of the j-th component, i.e. $p_{i,j} = P_i(c_j)$.)

The above example illustrates some possibilities to combine existing properties types into new property types for S_{agg}. The combination functions can be arbitrary; one condition is type compatibility between property types. Examples of such combination functions can be:

– *identity*: the new property type is identical to an existing property of some of the components. This is the case for p_1.
– *selection*: the new property type is selected from one of the existing properties based on their values. Examples are the minimum, the maximum, the first or last value from a set of property values, etc. In above example, p_2 is such a selection.
– *computed values*: the new property type's value is a function of values of existing properties. The range of arithmetic operators depends on the types of properties. For example, numeric numeric operations *add, subtract, multiply* or *divide* can be applied to numeric property types. For string typed properties, string operators such as *concatenation, sub-string, pattern match*, etc. are possible candidates. For multi-valued property types, set-operations such as *union, intersection*, and *difference* can be defined.
  p_3 in our above example is the arithmetic sum of two properties, whereas p_4 is the union of property values $p_{2,2}$ and $p_{1,3}$.

As illustrated by above discussion, the set of possible operators is unbound. We either introduce a set of known property types with associated operators – thereby defining a type specific property combination arithmetic – or support an open ended schema in which new combining functions can be defined, akin to data manipulation languages such as SQL.

Formally, we can capture the process of property definition for S_{agg} as follows.

We take the set of original component types $\{C_k^{orig}\}$ and extract their property types which we classify into sets of properties depending on their type compatibility. (We assume a limited set of base types from which we can derive the notion of type compatibility.) Call these sets $P_{T_j}^{orig}$ where T_j denotes a certain property type. We can then define a new property P with type T_j in some component of S_{agg} as:

$$P = f(P_{T_j}^{orig})$$

with f an arbitrary function over the values in $P_{T_j}^{orig}$.

Application Interfaces Above we have introduced procedures to evaluate the property values of a given profile and associated set of profile differnces. Once these operations have been performed, an application may use two operations to access and utilize preference information:

1. *property query*: the value of a property can be queried given the name of the property and component. We denote this $P_i(C_j(\hat{s}))$ i.e. the value of property type P_i after complete resolution of component type C_j in the original profile $\hat{s}$.
2. *property value relation*: two property values can be compared to assert if their values are identical or if one property is superior respectively inferior to the other value.

The comparison operation can be extended to range over components and profiles containing components.

4 Experiences with UAProf

The model defined in section 3 is a formalization of the WAP Forum defined *UAProf* framework. UAProf however only requires *default resolution* and *profile difference* resolution; aggregation of diverse profile schemas is not required[5]. At the IBM Zurich Research Laboratory we have implemented middleware software which performs default resolution and profile difference evaluation for UAProf compliant profiles. An API exposes operations such as:

- *profile resolution*: input to this function is a profile and a possible set of profile differences. The function applies profile defaults and differences to yield an API accessible data structure of profile components and properties.
- *component and property accessors*: once a profile has been internalized, diverse accessor routines allow to traverse the profile's set of components and their properties to query property values and types. These values are represented through appropriate data types in the chosen programming language (i.e. C or Java).
- *profile comparison*: this functionality is intended to support profile matching as warranted by the WAP Push Access Protocol (PAP) [20] standard. It compares a given device's profile with constraint profile: only if the device's capabilities exceed the required constraints will a push message be forwarded to the corresponding device.
- *profile externalization*: the API supports various external syntaxes to represent profiles and profile differences such as HTTP headers and MIME components in RDF/XML syntax.

The implementation has been done in the C language for performance reasons. On top of our base CPI library, we have implemented a Java wrapper using Java Native Interface (JNI) [17]. Additional software was written to use the CPI library in an Apache module [12, 14] or a Java Servlet context [16].

Several functional units can be distinguished in our UAProf/CPI implementation. An XML parser produces a DOM-like [18] representation of RDF/XML data on which RDF syntax rules are verified. Once the syntactical verifications for RDF/XML have succeeded, the profile data is verified against the UAProf schema.

Since no machine readable formalism has been defined for the UAProf schema, we have hand-crafted an extended database model to capture CC/PP compatible schemata and applied it to the UAProf schema[6]. Thus, our CPI libray can draw on machine accessible UAProf schema information. It is used in validating profile and profile difference data. Our database model for schemata captures a schema's components, their types, and the property types. For property types, we store its name, its type, and its resolution policy.

[5] With the exception of potential backward compatibility issues.

[6] In particular, the resolution policies for UAProf properties are not present in CC/PP.

After verification of the profile's compliance with the schema and the RDF/-XML syntax, default resolution is performed. For performance reasons, we have implemented a two level cache to hold default component values. The first level is an in-core cache; the secondary level uses a relational database. Hence, our CPI library retrieves default components – referenced via their URLs – only once across the internet to load the default component caches. Depending on the lifetime of the CPI library instantiation (e.g. per HTTP request or across multiple HTTP requests), access to a default compoment is serviced via the database cache or the in-core cache and becomes indpendent of network latency and throughput.

The default component cache can also be pre-loaded during initialization of the CPI library to avoid penalizing the very first profile resolution with retrieval of default component values.

A similar two-level cache approach is used for resolved profiles. Indeed, there is a high likelihood that clients present identical profiles with a series of subsequent HTTP requests since the device's capabilities and the user's prefernences are unlikely to change between subsequent HTTP requests. We detect this by checksumming the HTTP headers related to the profile information and using the checksum value as a look-up key into a profile cache. Thus, if subsequent requests carry identical profile information, profile resolution is only performed once; profiles for ulterior requests are serviced out of the profile cache.

Another implementation issue has been the handling of profile matching. It requires evaluation of an ordering function on profile value properties as described in section 3. For simple scalar types, such as integers or even dimensions, the ordering function is simple[7]. However, for literal enumeration values no simple relationship can be defined: lexicographical ordering, for example, is not meaningful.

Our implementation therefore associates an explicit enumeration of the relations between literal enumeration values. Evidently this externalized encoding of the order relationship requires $O(n^2)$ space in the schema database, but supports convenient extensibility of the order relationship for a given enumeration type without rebuilding the CPI library when new property values are introduced.

Whilst our implementation has not yet been tuned for performance, we have performed indicative performance measurements. Our sample profile is modified with two profile differences. All our data is resident on local disk and accessed via HTTP (i.e. using a TCP/IP loop-back connection). (Our figures were measured on an IBM RISC/6000 43P Model 150 running IBM's AIX version 4.3.)

- profile resolution without cached default: In this case, the profile's default components are retrieved over HTTP and resolved before the profile resolution itself takes place. Profile resolution in this set-up takes approximatively 0.2 seconds.

[7] We consider dimension $d_1 \leq d_2$ if the width and height of d_1 are less than the respective values of d_2.

– profile resolution with in-core cached defaults: Here, we ensure that the needed component defauls are loaded into the in-core default cache. Profile resolution time is reduced to 60 milli-seconds.
– profile resolution degenerating into profile lookup: Repeated resolution of identical profiles is avoided using an in-core profile cache as described above. The elapsed time in this case becomes 70 micro-seconds per profile resolution, which in this case degenerates into a lookup operation.

5 Conclusion

Based on the recommendations for CC/PP and the WAP UAProf standard, we have introduced a formal notation to describe the profile resolution process required by the underlying information model. Our notation precisely captures default handling and profile differences application to preferences and capabilities profiles.

We have also introduced a formal model for schema unification. It is open-ended since the mapping of existing to new profile property types can be expressed using any computable function. We have, however, not created a concrete expression syntax for this mapping, but an SQL [5] like formalism appears a possible candidate.

Our experiences with the implementation of the UAProf standard are described in Section 4. We illustrate the usage of database technology to hold CC/PP schema information, including relations on property values. Our approach enables extensions to existing vocabularies as well as the introduction of new ones.

The necessity of profile default information and profile caching is highlighted by our initial performance measurements. The measured performances indicate that judicious usage of caching reduces the overhead associated with CPI processing to a tolerable level when taking into account network and processing latencies and delays in an HTTP request-reply based application protocol.

We have shown that CC/PP technology can be implemented efficiently, despite some of its shortcomings. Future success of the technology will depend on the rate of adoption of the technology by mobile device manufacturers as well as the enabling of a wider range of middleware systems to handle capability and profile information and take advantage of this information in generating optimally adapted content.

6 Acknowledgements

This paper has benefitted from discussions with our colleauges François Dolivo and Stefan G. Hild on the topic of capabilities and profile information technologies.

References

1. T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax.* IETF, August 1998. RFC 2396.

2. K.H. Britton, R. Chase, A. Citron, R. Floyd, Y. Li, C. Seekamp, B.Topol, and K.Tracey. Transcoding: Extending e-business to new environments. *IBM Systems Journal*, 40(1):153–177, 2001.
3. Mark H. Butler. Current technologies for device independence. Technical Report HPL-2001-83, Hewlett Packard Laboratories Bristol, March 2001.
4. Mark H. Butler. Implementing content negotiation using CC/PP and WAP UAProf. Technical Report HPL-2001-190, Hewlett Packard Laboratories Bristol, August 2001.
5. C.J. Date and H. Darwen. *A Guide to the SQL Standard*. Addison-Wesley Publishing Company, third edition edition, 1993.
6. Keiichi Enoko. Concept of i-mode service. *NTT DoCoMo Technical Journal*, 1(1):4–9, October 1999.
7. R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. IETF, June 1999. RFC 2616.
8. GSM Association. *M-Services Guideline*, May 2001. PRD AA.35.
9. IBM Corporation. *The NetRexx Language*. http://www2.hursley.ibm.com/nextrexx.
10. G. Klyne, F. Reynolds, C. Woodrow, and H. Ohto. *Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies*. W3C, June 1999. http://www.w3c.org/TR/NOTE-CCPPexchange.
11. Ora Lassila and Ralph R. Swick. *Resource Description Framework (RDF): Model and Syntax Specification*. W3C, 1999. http://www.w3c.org/TR/REC-rdf-syntax.
12. Ben Laurie and Peter Laurie. *Apache: The Definitive Guide*. O'Reilly & Associates, 1999. Second Edition.
13. H. Nielsen, P. Leach, and S. Lawrence. *An HTTP Extension Framework*. IETF, February 2000. RFC 2774.
14. Lincoln Stein and Doug MacEachern. *Writing Apache Modules with Perl and C*. O'Reilly & Associates, 1999.
15. Sun Microsystem, Inc. *Java Server Pages, Version 1.2*, September 2001. http://java.sun.com.
16. Sun Microsystem, Inc. *Java Servlet Specification, Version 2.3*, September 2001. http://java.sun.com.
17. Sun Microsystems, Inc. *Java Native Interface*, May 1997. http://java.sun.com.
18. W3C. *Document Object Model (DOM) Level 1 Specification*, October 1998. http://www.w3.org/TR/REC-DOM-Level-1.
19. W3C. *CC/PP exchange protocol based on HTTP Extension Framework*, June 1999. http://www.w3c.org/TR/NOTE-CCPPexchange.
20. WAP Forum. *Wireless Application Protocol: Push Access Protocol (PAP)*, August 2001. WAP-247-PAP.
21. WAP Forum. *Wireless Application Protocol: Push OTA Protocol*, August 2001. WAP-235-PushOTA.
22. WAP Forum. *Wireless Application Protocol: User Agent Profile Specification*, August 2001. WAP-248-UAPROF.
23. WAP Forum. *Wireless Application Protocol: WMLScript Specification*, August 2001. WAP-193-WMLS.