

Personis: A Server for User Models

Judy Kay, Bob Kummerfeld, and Piers Lauder

Department of Computer Science
University of Sydney, Australia, 2006
{judy, bob, piers}@cs.usyd.edu.au

Abstract. A core element of an adaptive hypertext systems is the user model. This paper describes Personis, a user model server. We describe the architecture, design and implementation. We also describe the way that it is intended to operate in conjunction with the rest of an adaptive hypertext system.

A distinctive aspect of the Personis user model server follows from our concern for making adaptive systems scrutible: these enable users to see the details of the information held about them, the processes used to gather it and the way that it is used to personalise an adaptive hypertext. We describe how the architecture supports this.

The paper describes our evaluations of the current server. These indicate that the approach and implementation provide a workable server for small to medium sized user collections of information needed to adapt the hypertext.

Keywords: Server for user profile/model, Security and Privacy of User Models, User Modelling, Personalisation, User Control

1 Introduction

Adaptive hypertext relies on the availability of information about the user as a foundation for its adaptivity. This paper describes our approach to building a server for such user models. The motivation for such a server follows from the nature of the information in a user model, the difficulties associated with building good user models and the way that such models might be used in a range of applications.

First consider the nature of user modelling information. Because it constitutes personal data, it needs to be treated rather differently from other parts of an adaptive hypertext system: it is subject to far tighter requirements for security of the information. For systems to move out of the laboratory, it will have to meet legal requirements such as the European Community Directive on Data Protection² It is in the spirit of such legislation that users be able to access and control their own data. A server makes sense for the provision of the required security at the same time as ensuring user access and control.

² <http://www.doc.gov/ecommerce/eudir.htm> (visited Jan 2002)

Another important problem for user modelling is that it takes considerable time and effort to build up a detailed user model. When users first comes to an adaptive hypertext system, they either have to accept a generic interface initially or they have to provide information about themselves. A server should enable the reuse of the user model across applications. In particular, suppose the user explores one adaptive hypertext to do some substantial activity such as learning how to program in C [1]. When they move to another adaptive hypertext system that teaches Java, it would be useful for that system to be primed with the user model that has already been built up.

For the most part, work on personalisation has placed the user model within an application. For example, there have been several user modelling shells: GUMS [2,3] and its successor, GUMAC [3]; BGP-MS [4] UMT [5] and in the area of student modelling, TAGUS [6] These systems explored many issues in building generic tools for managing user models. They did not provide a server for reuse of user model information across applications. A recent review of generalised support for user modelling [7,8] concluded that we have yet to see a user model server that addresses the needs for ensuring the user's privacy, control and ability to scrutinise their user model and the processes for personalisation.

There has been some work on user model servers. Orwant built a system [9] with user models in a Lisp-like language. Orwant took care to encrypt the user model during transmission. Paiva also built a server [10] which could support multiple teaching agents. A rudimentary form of user model server is provided by Hailstorm³. This initiative is an indication of the recognition of the value of a user model.

Interestingly, Hailstorm is described with a focus on user control: 'It puts users in control of their own data and information, protecting personal information and providing a new level of ease of use and personalisation.' It allows a registered user to store personal data in a standard form with standard access methods on a central server provided by Microsoft. Access to the information is provided through a standard set of services using the Simple Object Access Protocol (SOAP) and is based on the earlier 'Passport' system for storing user names and passwords. It seems to be intended for broad tasks: address book, email, diary, documents, device settings. It is claimed to give users control over the data while retaining privacy. However, the architecture presents both a single point of failure and a single point of security vulnerability. It is described as a simple data store, without inferencing ability.

Several of these systems, including BGP-MS, UMT and TAGUS, had interfaces for the use of the developer to scrutinise the models as they built and debugged them. However, the representation and reasoning mechanisms were not designed for the user to scrutinise them.

The goal of the Personis project is to explore ways to support powerful and flexible user modelling and at the same time to design it, from its foundations, to be able to support user scrutiny and control. Our underlying representation of the user model [11,12] collects *evidence* for each component of the user model.

³ <http://www.microsoft.com/net/hailstorm.asp> (visited Jan 2002)

In large scale field testing [12] we have demonstrated that many users can and do scrutinise their user models. In a relatively short eight-week study, some users scrutinised the full range of details of the user model, including the meanings of the components of the model, the details of evidence about each one, the details of the evidence sources and the reasoning used to infer component values. In this paper, we describe the way that the Personis server builds upon this foundation to provide a user model server that can support scrutability for adaptive hypertext systems.

In Sect. 2, we describe the architecture of the Personis server and the way that the server fits into the architecture of an adaptive hypertext system. Section 3 gives an overview of its application in a simple recommender application. We report results of some evaluations of the current Personis server in Sect. 4 and Sect. 5 has discussion and conclusions.

2 Architecture of a Personis-Based Adaptive Hypertext System

2.1 High Level Architecture of Several Adaptive Hypertext Systems

Figure 1 shows the way that a Personis server can support reuse of the user models over a series of adaptive hypertext systems. This illustrates four main ideas that we now describe: the server itself; generic scrutiny tools that enable the user to see and control their own user model; a collection of adaptive hypertext applications; and the *views* which are the conceptual, high level elements shared between the server and each application.

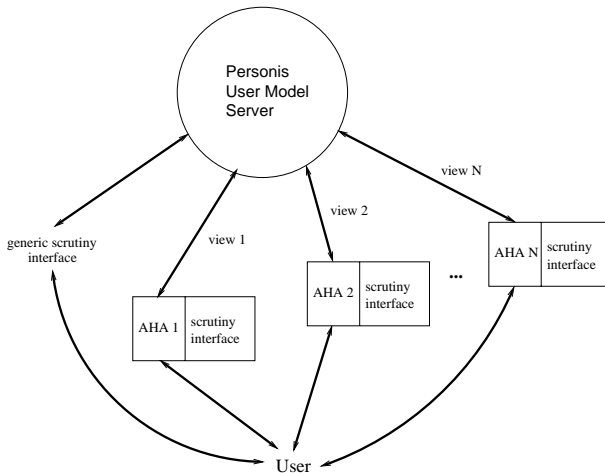


Fig. 1. Personis Server Architecture

In keeping with our previous work, our architecture includes provision for tools for the user to scrutinise their user model. This might include an interface to explore the details of each component of the model, as in the um toolkit [12] and we envisage additional tools for exploring large user models [13] We do not focus on the range of these tools in this paper. We do, however, note that such tools are important to our approach. The figure shows the user interacting with the *generic scrutiny interface* and this, in turn, interacting with the server. In fact, as the figure suggests, interfaces that support user scrutiny of the user model operate as a special type of adaptive interface. (These tools may well be adaptive hypertext applications in their own right.)

Figure 1 shows each adaptive hypertext application with two parts: the core of the adaptive hypertext which enables the user to do some task such as learn to program; and, in addition, we show a scrutiny interface associated with that adaptive hypertext application. This structure is important. If a user model server were to be in practical use, we would expect that the user model for each individual would steadily grow to be quite substantial. Although we are committed to supporting the user's scrutiny of that model, we expect that users will generally want to explore their model in the context of their interaction with a particular application. So, for example, the user might be using an adaptive hypertext that teaches the programming language, C. As they do so, they might wonder why it presented information in a particular way. They might also see a friend using the same system and if its adaptation for that friend is different, our user might want to explore why. In this type of user-scrutiny, the answers to their questions will typically involve the interaction of the adaptive hypertext application and the user model. So, it makes sense to provide support for the user to scrutinise the adaptivity within the context of the adaptive hypertext application.

Issues of scale and comprehensibility give another reason for supporting scrutiny of the user model within the adaptive hypertext application. For the case of the C hypertext, the user would probably be primarily interested in those parts of the user model that are used by that application. Since this will be a small part of a full user model, it is a more manageable and relevant aspect to explore. Our architecture requires that scrutability be supported in the application.

The last element of Fig. 1 is the *views* of the user model available to each adaptive hypertext application. For example, the leftmost application in the figure might need just a few components of the user model. Our architecture allows the definition of a view that defines just these components. Another application will typically use a different view. The application writer would define those parts of the user model needed by their application and these would be defined in views established for that application.

Importantly, these views have an interaction with the design of the access control for the server. Personis allows the user to define just which applications are allowed to see each part of the user model. The user can also control the information sources that should be made available to each applications. So, for

example, the user model may contain evidence from several sources about the user's knowledge of programming. That user could decide to make only the information from certain sources available to an application. Another user might make a different decision. Access control information is stored with the user model in the object database.

In particular, suppose an application like AHA1 in the figure teaches about C and it collects data from the user's answers to quiz questions. It provides this to the user model as evidence about the user's knowledge. Further, suppose that AHA2 teaches about a somewhat related subject, Java programming. The user can control whether AHA2 is allowed to access user model evidence that was provided to AHA1. The user can also control just which components of the model are available to AHA2. This means that if AHA2 requests information about the user's knowledge of Java, this will only be provided if the user has made it available to AHA2.

2.2 Internal Architecture of Server

The internal architecture of a Personis server is depicted in Fig. 2. The user model information is held as an object database. This stores the user model in a representation that has the same conceptual foundation as the um toolkit [12]. The basic element is the *component* and each has an associated list of *evidence*. Each piece of evidence is tagged with information about its source. The object database also manages the structuring of user models into *contexts* which give a hierarchical structuring of the component namespace. It also holds objects which define the views of Fig. 1. Any context can define a view which includes components from any part of the user model context hierarchy.

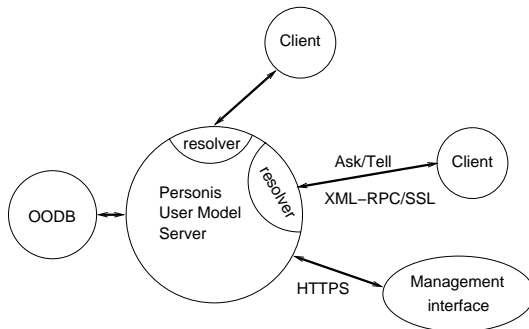


Fig. 2. Personis Server Architecture

The object database also holds the access control information. The establishes access rights to for applications and users. Each aspect can be controlled at the common levels of readable and/or writable.

The server needs to handle large numbers of clients simultaneously. To achieve this, the current implementation is an object layer over a relational database. Internally, objects in the database are referenced using an Object Identifier that is resolved to a server address. This allows parts of user models to reside on different servers. It is envisaged that users would want to keep most of their personal model on a local system under their control. They can then choose to allow selected information to be stored on servers outside their direct control.

As shown in Fig. 1, a view defines a collection of components and an application can ask for the values of these components or send evidence about these to the user model. As shown in Fig. 2, this communication between server and client operates with remote method calls using the XML-RPC⁴ protocol over SSL (secure socket layer).

In addition, the server provides a management interface implemented with HTML, HTTP and SSL protocols and accessible from any web browser. The management interface for the user model server has several functions. It provides a status display for the running server and allows reconfiguration. It also allows a suitably privileged user to create and manipulate user model definitions as well as create new user models for individuals according to a previously created definition. It also provides interfaces for setting and altering access control.

The user model server is designed to be scalable to a large number of users, each with large models. Access to model objects by an application is extremely simple with only one line of program code typically required to acquire a complete set of component values corresponding to a view.

The remaining element of Fig. 2 is the *resolvers*. These follow the approach of the um toolkit where the generic user model simply holds the uninterpreted collection of evidence for each component. It is only at runtime that the application uses a resolver to interpret the evidence available to it and conclude the value of a component. A default resolver is available but specialised resolvers can be associated with an application. So, as shown in Fig. 2, one client adaptive hypertext system might use one resolver. Another client might use another and so interpret the same component differently. To take a simple example, Resolver A might treat the user's self-assessment of their knowledge as highly reliable. Resolver B might give higher reliability to the user's performance on quizzes. Then, suppose Rebecca is the user who is quite knowledgeable about C control structures as evidenced by quiz performance. Suppose that she lacks confidence and rates her knowledge as low in this area. An application which uses Resolver A will treat her as not knowing C control structures. One that uses Resolver B will treat her as knowing them. Note that, Rebecca could decide that evidence derived from her quiz results was not to be made available to either of these applications: in that case, the server would not present evidence to the resolvers and associated applications would operate as if Rebecca's user model had no quiz results.

⁴ www.xmlrpc.org

2.3 Application View of Personis

The basic API for the user model server is simple and elegant. Remote method calls using the XML-RPC protocol (and possible future use of SOAP) provides user model access to any application with a client side XML-RPC implementation. The basic client API consists of three calls:

```
um = access(odbname, user, password)
```

The `user`, `password` and `odbname` are strings. The system maps the `odbname` to a server address.

```
components = um.ask(context, view, resolverident)
```

The `context` is a list of context names giving a path to the required context. The `view` is either a simple string indicating a view name, or a list of names of components. The resolver ident is the name of a resolver located at the server to resolve the values. If the resolver ident is omitted a default is used.

```
um.tell(context, component, evidence)
```

The `evidence` is a list containing the type of the evidence, an optional expiry time, and the value of the component.

An application can collect resolved values for a complete set of components, as defined by a view, using an `ask` statement. Combined with a statement to connect to the server and one to close the connection the entire interaction with the server is three lines of code.

3 Overview of User Model Server

We now illustrate the architecture in an example application: a “Personal Jazz Channel” that provides users with personalised jazz programmes.

Like many such systems, it asks the user to prime its user model by answering a small set of carefully chosen question. This screen is shown in Fig. 3.

From this, the system sends a collection of evidence to Personis. That evidence is tagged as being given by the user, via the Personal Jazz Channel query interface.

Then, the PJC application makes inferences about other CDs, styles and artists. Each of these generates a piece of evidence which is tagged as inferred and added to the user model by the Personal Jazz Channel application. This very simple process enables the system to have a rich set of preference data from a small set of questions.

At this point, a personal streaming audio channel is created with a mix of tracks conforming to the user’s modelled preferences. During normal interactions, the Personal Jazz Channel adds new pieces of evidence about tracks the user allows to play, those that are skipped and so on. Each time the user comes to the Personal Jazz Channel, they log in and the system sets up a connection with

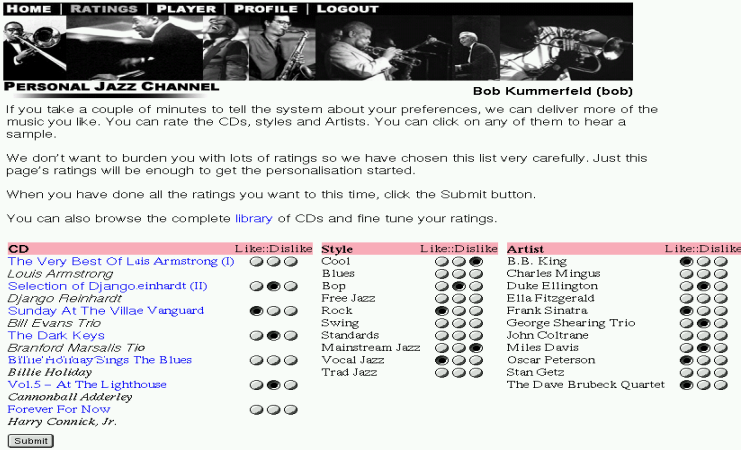


Fig. 3. Example rating screen for a personalised jazz channel

the Personis server and *asks* for the user model view it needs to perform its personalisation.

The Personal Jazz Channel application interacts with the Personis server to gather relevant model information. For example, to gather all the resolved values for jazz styles the following lines of code connect to the server, retrieve the values and close the connection:

```
um = access(odbname, user, password)
styles = um.ask(context=["music","jazz","styles"])
um.close()
```

In this case the context is ‘music->jazz->styles’ and the default *view* for the context is all the components in that context. The `um.ask` call will return a dictionary of resolved values. Each element of the dictionary is a tuple containing the resolved value, the name of the resolver used and the time it was resolved.

The Personal Jazz Channel operates in the way illustrated in Fig. 1. In addition to its job as a typical customised application, it also has an interface to support scrutability. At any time, the user is able to examine and change the personal information held by the system by selecting the *Profile* button at the top of the screen. This brings up a screen like that shown in Fig. 4.

From the point of view of our architecture, there are some important aspects to point out. Firstly, we note that it is an essential aspect of the architecture that the Personis server and each of its associated adaptive applications is loosely related. The authors of the application are responsible for it. Different applications will be created by different people and will work differently. The user needs to explicitly allow an application to access relevant views in the user model. The user may decide that the security of some systems is effective enough that it is acceptable to allow them access to substantial amounts of the user model. On the other hand, a user may be less happy with the security protection in another

HOME | RATINGS | PLAYER | PROFILE | LOGOUT

PERSONAL JAZZ CHANNEL Bob Kummerfeld (bob)

For a more detailed view click [here](#).

personal details	Style	Rating
<input type="checkbox"/> first name	Cool	<input type="text" value="Don't like"/> (?)
<input type="checkbox"/> last name	Rock	<input type="text" value="Like"/> (?)
<input type="checkbox"/> birth date	Vocal Jazz	<input type="text" value="Like"/> (?)
<input type="checkbox"/> gender	Mainstream Jazz	<input type="text" value="Don't like"/> (?)
jazz preferences	Bop	<input type="text" value="----"/> (?)
<input type="checkbox"/> styles		<input type="button" value="Submit new ratings"/>
<input type="checkbox"/> artists		
<input type="checkbox"/> CDs		

Fig. 4. Profile information: the user has selected *styles* from the list at the left and sees the user model components for styles of music.

system: that application might be authorised to access only a limited part of the user model and only information derived from a few of the evidence sources.

4 Evaluation

The current server has had very little work to optimise performance but still performs at an acceptable level. A test application performing 10000 accesses achieved the following number of transactions per second:

tell (new value each time)	26/sec
ask (single component)	17/sec
ask (view with 2 components)	13/sec

Each of the ask operations used a default resolver (most recent given evidence value).

The server was running on an 850Mhz Duron processor with 512MBytes memory. The user model being accessed had approximately 50 components arranged in 15 (sub)contexts. The database would have cached the complete model after the first access and so the figures quoted show performance for the client/server protocol and the internal data structure search and overhead.

Investigation has shown that the XML-RPC protocol is a major overhead. This is mainly due to the poor performance of the general purpose XML parser used in the implementation. We feel that significant performance gains can be made with a hand crafted parser. We feel strongly that the use of a standard protocol such as XML-RPC is warranted since it provides access to the Personis server from a wide range of programming languages.

From the application programmers point of view the Personis server is very easy to use. Only a handful of lines of code are required to retrieve the resolved

values of a sets of components. In a similar way, it is very straightforward to add evidence to existing components.

5 Conclusion

The underlying design of the Personis user model server is based upon the primary requirement that users have access to their user model and control over it. In addition, it has been designed to provide support for user modelling with an elegant but powerful programmer interface. It is novel in its design being explicitly focussed on user control and scrutability.

References

1. Kay, J., and Kummerfeld, R.J.: An individualised course for the C programming language. Online Proceedings: <http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/Educ/kummerfeld/kummerfeld.html>, Elsevier (1994)
2. Finin, T. W.: GUMS - a general user modeling shell. In: Kobsa, A., Wahlster, W. (eds.): *User models in dialog systems*. Springer-Verlag, Berlin (1989) 411–431
3. Kass, R.: Building a user model implicitly from a cooperative advisory dialog. *User Modeling and User-Adapted Interaction* **1** (1991) 203–258
4. Kobsa, A., and Pohl, W.: The user modeling shell system BGP-MS. *User Modeling and User-Adapted Interaction* **4** (1995) 59–106
5. Brajnik, G., and C Tasso, C.: A shell for developing non-monotonic user modeling systems. *International Journal of Human-Computer Studies* **40** (1994) 36–62
6. Paiva, A., and Self, J.: TAGUS - a user and learner modeling workbench. *User Modeling and User-Adapted Interaction* **4** (1995) 197–228.
7. Kobsa, A.: *Generic User Modeling Systems*. *User Modeling and User-Adapted Interaction - Ten Year Anniversary Issue* **11** (2001) 49–63
8. Fink J., Kobsa, A.: A Review and Analysis of Commercial User Modeling Servers for Personalization on the World Wide Web. *User Modeling and User-Adapted Interaction - Special Issue on Deployed User Modeling* **10** (2000) 209–249
9. Orwant, J.: Heterogenous learning in the Doppelganger user modeling system. *User Modeling and User-Adapted Interaction* **4** (1995) 59–106
10. Machado, I., Martins, A., Paiva, A.: One for all and all for one: a learner modelling server in a multi-agent platform In: Kay, J (ed): *User Modeling: Proceedings of the Seventh International Conference, UM99*. Springer Wien, New York (1999)
11. Kay, J.: Accretion representation for scrutable student modelling. In: Gauthier, G., Frasson, C., VanLehn, K. (eds.) *Intelligent Tutoring Systems* (2000) 514–523
12. Kay, J., The um toolkit for cooperative user modelling. *User Modeling and User-Adapted Interaction* **4** (1995) 149–196
13. Uther, J., On the visualisation of large user models in web based systems. Phd Thesis, Department of Computer Science, University of Sydney (2001)