LAPPEENRANTA UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF INFORMATION TECHNOLOGY

The synchronization of personal information between mobile devices and online services by using SyncML

The subject of thesis was approved by the council of the Department of the Information Technology on 3rd of December 2003.

Supervisors: Professor Jari Porras, MSc Pekka Jäppinen

Lappeenranta, March 24th 2004

Mika Yrjölä

Teknologiapuistonkatu 4 C 9

53 850 Lappeenranta

ABSTRACT

Author: Mika Yrjölä

Subject: The synchronization of personal information between mobile devices and online services by using SyncML

Department: Department of Information Technology

Year: 2004

Place: Lappeenranta

Master's thesis. Lappeenranta University of Technology. 56 pages, 9 figures and 2 algorithms.

Supervisors: Professor Jari Porras, MSc Pekka Jäppinen

Keywords: Bluetooth, browser plug-in, personal information, SyncML, synchronization

A significant number of current mobile devices such as phones and PDAs provide support for storing personal information as well as short-range wireless connectivity. At the same time, the amount of online services that use personal information is increasing quickly. The information stored on personal mobile devices could potentially be used to eliminate the need for manual entry of the same information to online services.

This thesis presents a solution that can transfer and synchronize the personal information between the mobile device and online services. The solution is implemented as a browser plug-in. Existing solutions with related functionality are presented and evaluated for their success in the elimination of manual (re)entry of personal information. Introduction to the standards and technologies, especially SyncML and Bluetooth, that are used by the browser plug-in is given. After introducing the high-level architecture of the plug-in, the implementation details are presented. The result of the project is a theoretically working concept, although the current personal mobile devices make the implementation more difficult than it could be.

TIIVISTELMÄ

Tekijä: Yrjölä, Mika

Nimi: Henkilötietojen synkronointi mobiililaitteiden ja verkkopalveluiden välillä SyncML:n avulla

Osasto: Tietotekniikan osasto

Vuosi: 2004

Paikka: Lappeenranta

Diplomityö. Lappeenrannan teknillinen yliopisto. 56 sivua, 9 kuvaa ja 2 algoritmia.

Tarkastajat: Professori Jari Porras, diplomi-insinööri Pekka Jäppinen

Hakusanat: Bluetooth, henkilötiedot, selainlaajennos, SyncML, synkronointi

Monet henkilökohtaiset mobiililaitteet tarjoavat mahdollisuuden tallentaa henkilötietoja ja mahdollisuuden lyhyen kantaman radiotekniikoiden hyödyntämiseen. Vastaavasti henkilötietoja käyttävien tai vaativien verkkopalveluiden määrä on kasvussa. Mobiililaitteisiin tallennetut henkilötiedot tarjoavat potentiaalisen keinon välttää samojen henkilötietojen toistuva käsinsyöttö erilaisiin verkkopalveluihin ja keskitettyyn ajantasallapitoon.

Tässä työssä käydään läpi ratkaisumalli henkilökohtaisen mobiililaitteen ja verkkopalveluiden välillä tapahtuvaan henkilötietojen siirtoon ja synkronointiin. Malli pohjautuu selainlaajennukseen, joka voi pyytää sekä selaimessa auki olevalta verkkopalvelun sivulta että mobiililta päätelaitteelta senhetkiset henkilötiedot ja synkronoida ne. Jo olemassaolevia henkilötietojen hallintaa helpottavia ratkaisuja käydään läpi arvioiden käyttökelpoisuutta tämänkaltaisiin tarpeisiin. Ratkaisumallin kannalta olennaiset tekniikat ja standardit, erityisesti Bluetooth ja SyncML, esitellään. Ratkaisumallin arkkitehtuuri käydään korkealla tasolla läpi ja esitellään toteutuksen yksityiskohtia. Tuloksena on periaatteeltaan kelvollinen henkilökohtaisten tietojen synkronointijärjestelmä, jonka toteutusta nykyisten mobiilien päätelaitteiden toiminnallisuus jossain määrin hankaloittaa.

Preface and Acknowledgments

Special thanks to: Anne for bearing my absentmindedness and late work hours, my parents for not asking too often when I graduate and giving general support, #plop folks, other friends and co-workers in Comlab.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

ACL Asynchronous Connection-Less

AFH Adaptive Frequency Hopping

API Application Programming Interface

DOM Document Object Model

DTD Document Type Definition

ETSI European Telecommunications Standards Institute

FHSS Frequency Hopping, Spread Spectrum

GAP Generic Access Profile

GCC GNU C Compiler

GNU GNU's Not Unix

GSM Global System for Mobile Communications

GPRS General Packet Radio Service

GUI Graphical User Interface

GUID Globally Unique Identifier

HCI Host Controller Interface

HTTP Hypertext Transfer Protocol

IDL Interface Definition Language

IEEE Institute of Electrical and Electronics Engineers

IrDA Infrared Data Association

ISM Industrial Scientific Medical

L2CAP Logical Link Control and Adaption Protocol

LAN Local Area Network

LDAP Lightweight Directory Access Protocol

LMP Link Manager Protocol

LUID Locally Unique Identifier

MIME Multipurpose Internet Mail Extensions

NFS Network File System

OBEX Object Exchange

OSI Open Systems Interconnection

PC Personal Computer

PDU Protocol Data Unit

PPP Point-to-Point Protocol

PTD Personal Trusted Device

PUID Passport User ID

QoS Quality of Service

RFCOMM RF Communications

SAA Service Accessing Application

SAD Service Accessing Device

SCO Synchronous Connection-Oriented

SDP Service Discovery Protocol

SLP Service Location Protocol

SSO Single Sign-On

TAF Target Address Filtering

TTY Teletype Terminal

USB Universal Serial Bus

URI Uniform Resource Identifier

URL Uniform Resource Locator

URN Uniform Resource Name

UID Unique Identifier

UUID Universal Unique Identifier

XML Extensible Markup Language

XPCOM Cross Platform Component Object Model

XPIDL Cross Platform Interface Definition Language

WBXML Wap Binary XML Content Format

WSP Wireless Session Protocol

1 Introduction

During the last few years, the amount of online services that require personal information to provide personalized services has been steadily increasing. Some examples of this kind include online stores like Amazon.com and various online communities. In the context of this document, personalization means that the content or appearance of something, e.g. a web page, is altered automatically or semi-automatically to (hopefully) better suit the tastes and interests of the user. For example, the previously mentioned Amazon.com personalizes the list of books, music records and other product categories it sells to include material that has better than average probability of being interesting to the user. Personalization by its very definition needs information about the user to work. Entering the same personal information repeatedly for new services can be cumbersome and unpleasant.

During the same time, the mobile devices have become increasingly common especially in Europe. A significant amount of these devices offer features like vCard or vCalendar in addition to other functionality. Information stored in these containers includes data items such as name, address and other details that are somewhat commonly required by online services. It is also quite safe to assume that in future the storage capabilities of mobile devices will keep increasing, as well as have more diverse and flexible areas of use than the existing ones. Capability to communicate with nearby devices using short range communication technologies such as IrDA (Infrared Data Association) and Bluetooth in addition to traditional GSM (Global System for Mobile Communications) network based long-range communication is also increasingly common. These capabilities allow the information stored on the mobile devices be exchanged with the outside world, provided that the other party has similar communication hardware. However, mobile devices do have some limitations. One of the most significant limitations is that most mobile devices do not have a real keyboard. This makes information entry and modification somewhat slow and clumsy, although solutions such as on-screen keyboards, predictive text input (e.g. T9 [1]) and handwriting recognition (e.g. CIC Jot [2]) do somewhat help.

While considering both of these facts, a question about the possibility of using the personal information stored on the mobile device to solve the various problems that were described earlier naturally arises. Additionally, it should also be possible to transfer information bidirectionally; not just from the mobile device to the service, but also from the service to mobile device. Bidirectional communication capability would open several other opportunities. First of these is the possibility of performing synchronization to get the information on both sides quickly up to date with minimal amount of duplicated work. The second potential gain would be that the synchronization could also offer a way

to escape the aforementioned less than ideal information input and modification methods of mobile devices. If significant amounts of data would have to be modified or inserted, it could be done on a PC (Personal Computer) while using online services, with the considerably more convenient real keyboard. The subsequent synchronization would largely eliminate the need of using the inferior input methods of the mobile device itself.

1.1 The subdivision of the problem

Now that the original problem (the entry of personal information to online services is cumbersome) and the goals of the solution (the personal information stored on a mobile device should be made available for online services in bidirectional and easy fashion) are defined, they can be divided into several largely separate subproblems. First of these problems is the task of gaining access to the personal information stored on the mobile device (and vice versa, how to access the personal information entered to an online service). The second problem concerns the task of transferring this information between the mobile device and the online service. The third problem concerns the granularity of the synchronization; in many cases it is simple to import or export all data at once, but in most synchronization situations a more fine-grained control over the operation is required. As a final problem, it is important to consider the problem on different levels and seek related existing problems. When the relations to existing similar problems are known, it is easier to to minimize the amount of duplicate work while also maximizing the usefulness of solutions for the related problems.

For the first problem (accessing the information on the mobile device/online service), the following two questions need to be answered:

- What are the common standards and protocols to store and access personal information on the mobile devices?
- Which of those are reasonably widely supported, have good future prospects and a generally good reputation in terms of performance and ease of use?

The same questions about available standards and protocols apply for the second problem (actual transfer of information) as well. However, additional questions arise:

- What kind of communication links between the mobile devices and outside world are common? Which is the best for this particular task?

- How different communication links support or are supported by the protocols and standards that are used to access the stored information?
- What kind of security (if any) is required? How it will be handled?

For the synchronization itself, many of the questions concerning previous subproblems are important here as well. An additional important point to consider in the context of synchronization exists, however:

- What kind of limitations (if any) the synchronization solutions and standards have regarding the types of information that can be synchronized?

Finally, the last problem presented turns out to have much in common with the first. Because many aspects of the problem such as transferring the data between devices and parsing it are not particularly unique to this case, the solution of this problem has strong ties to existing protocols and solutions that are used to access data stored on e.g. mobile phones and PDAs.

Now that the general objectives and desired characteristics of the solution have been described, a rough outline of the proposed solution can be presented. The solution consists of a web browser plug-in that handles the information transfer and synchronization between the currently open web page and the mobile device, as illustrated by figure 1. The plug-in can be embedded into any web page that might benefit from it and does not affect the ordinary use of the page in any way. Thus the proposed solution degrades gracefully, which can also included in the list of important objectives. After all, the solution should not prevent the normal use of service, if the user does not wish to use the additional functionality provided by the plug-in.

The main reason of implementing the solution as a plug-in is that a plug-in can implement easy access all necessary parties that may have personal information available. Because the online service is accessed by using a browser, a browser plug-in can trivially access the contents of the current web page, including the personal information. Additionally, the plug-in can implement access to any short-range communication link that the computer has in order to communicate with the mobile device carried by the user.

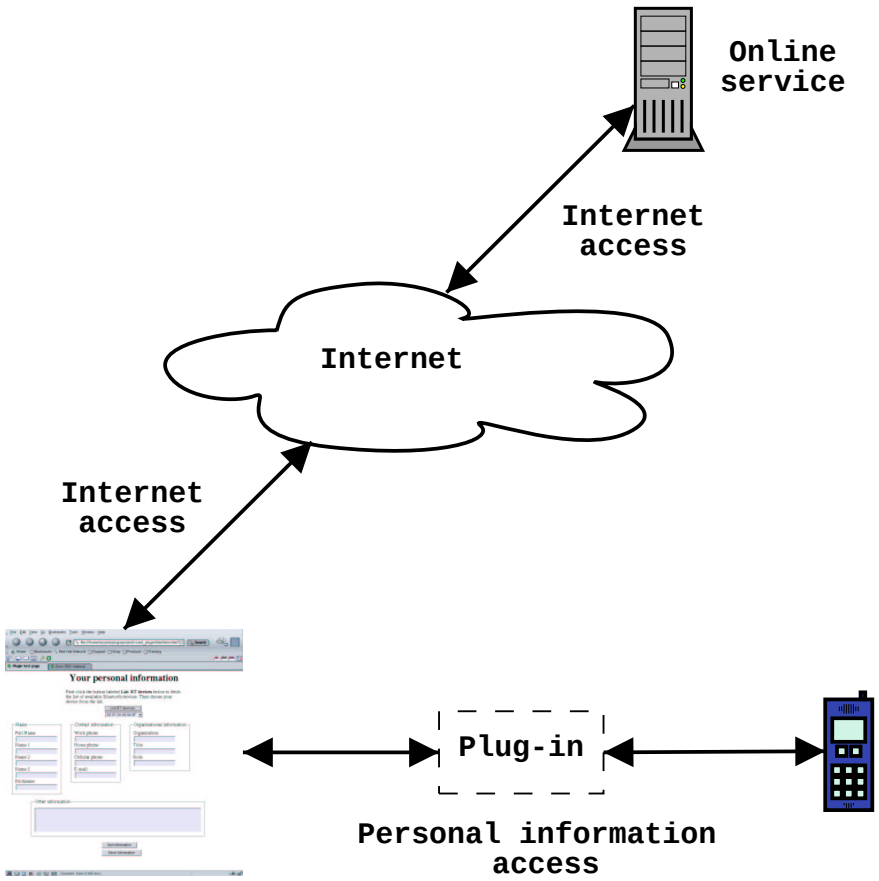

Figure 1: The structure of the proposed solution

The plug-in can be asked to fetch personal information from the chosen mobile device at any time by using a suitable short-range wireless communication link that fulfills the above criteria. Wireless link is preferred because of the increased convenience over wired connectivity such as USB and serial ports cables. After the information from the mobile device is available to the plug-in, it performs a search for conflicts between the retrieved information and the possibly available personal information on the web page. If conflicts are found, they are resolved. The possible changes are then propagated back to relevant destinations to bring both the storage on mobile device and the contents of the web page up to date. The result is consistent personal information on both mobile device and the web page of an online service. The actual transfer of the personal information from the web browser to the online service is not affected by the use of the plug-in in any way. Ideally, the information retrieval and synchronization performed via the plug-in should also be quicker than entering large amounts of personal information manually. The research presented on this thesis is focused around the goal of successfully designing and implementing a plug-in that follows the guidelines introduced in this chapter.

2 Potential standards and technologies to solve the synchronization problem

Although the general idea of proposed solution has now been presented, it is useful to give some attention to existing solutions first before explaining the proposal in further detail. As noted on the chapter 1, the currently existing related solutions and potentially useful protocols and standards related to handling and synchronization of personal information should be looked at carefully in order to find out how other parties have solved their related problems, as well as their possible pros and cons. This avoids pitfalls such as the infamous reinvention of the wheel and performing the same mistakes someone else has already done.

2.1 Existing solutions

Some solutions for minimizing the annoyance of repetitive data entry do already exist. These include browser *form filling features*, *roaming profiles*, *cookies* and *single sign-on*. Each of these has a slightly different approach and target area, so they are only partially overlapping. These are described in the following chapters.

2.1.1 Form filling functionality and roaming profiles

Many current browsers, like Mozilla and Internet Explorer offer functionality for storing information entered into web forms. This data can then later be retrieved and placed into form fields without typing it again. The form filling functionality of Mozilla is briefly covered as an example. Mozilla offers user the option of storing form contents either when submitting or manually at any time. Later, when the user returns to the same page, it is possible to ask browser to restore some or all of the stored field contents. The stored values can be edited later, as shown in figure 2, which is a screenshot of Mozilla 1.0 Form Manager GUI (Graphical User Interface). As can be seen, in addition to the URL(Uniform Resource Locator)-dependent storage of form field data, the Form Manager also allows storage of URL-independent information for name, address, phone numbers and other personal attributes.

The recognition of the form fields relies on their name attributes in order to work, e.g. the first name of person is set into a form field that has name attribute value of “Name.First”.

Because these exact names are unlikely to be used widely on the Web, the Form Manager includes functionality to map between these and some of their common variants. For example, form fields with name “fname” or “firstname” are recognized as potential equivalents of “Name.First”. This and some additional heuristics allow the once entered personal information to be inserted quickly into other locations even if the naming convention is dissimilar. In addition to the Form Manager, Mozilla also has a separate Password Manager that handles the common combination of login and password - style forms.

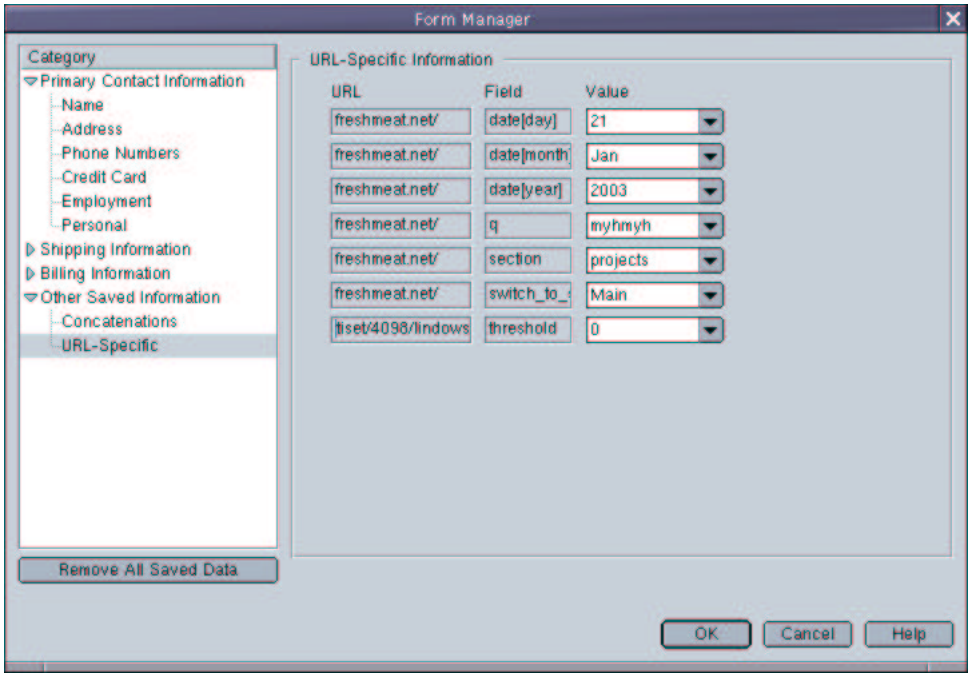

Figure 2: Mozilla Form Manager GUI

Functionality, such as the one described above and its counterparts on other browsers can significantly enhance user experience, but it does still have a major drawback: the information is stored into the browser data directory. If the browser is not able to access this directory, the existing information can not be used by the form fill functionality. So, in most cases this functionality is limited either to a single computer, or in case of the directory residing on a networked filesystem such as NFS (Network File System), to a local network. When user moves outside the boundary of access to the stored information and uses a browser on strange machine, the information is not accessible anymore.

This limitation can be partially worked around by using so-called *roaming profiles* that are supported by some browsers. The term has several different meanings, but in this context, it means that the browser can access data like preferences, bookmarks and the stored form fill data from a remote computer by using LDAP [3](Lightweight Directory

Access Protocol) or HTTP (Hypertext Transfer Protocol). However, this approach does still have some problems similar to the use of networked file systems as well as some problems of its own.

If the computer that is used to actually store the profile is not available literally 24 hours per each day, it may be difficult to trust that the profile is available when needed. Additionally, the installation, configuration and maintenance of LDAP service, or in case of HTTP, web server, may feel too daunting for average user. If implemented or configured carelessly, these solutions may also cause security risks. The browser support for roaming profiles with suitable features also causes problems; for example, Netscape 4.x does have support for roaming profiles but not any kind of form fill functionality. Mozilla and Netscape since version 6 support form fill functionality but roaming profiles are not yet implemented [4, 5]. The current versions of the most commonly used browser of the moment, Internet Explorer, support form fill functionality and use the roaming profiles of Windows operating system, which limits interoperability with non-Windows - based computers. When the incompatibility of various roaming profile implementations is added to the list of problems, it becomes clear that despite begin useful, this solution has quite a few flaws.

2.1.2 Cookies

Another well-known method to restrict the need for repeated entering of personal information are cookies [6]. The original HTTP protocol is stateless, which means that each transaction between the web browser and a web site is unrelated to previous transactions. Obviously, the lack of any persistence makes personalization and any other use of personal information very hard. While the problem can be worked around in many ways (e.g. databases on server side), in case of the cookies this is solved by implementing a simple state management mechanism for the HTTP with some additional headers. The exact syntax and feature set of cookies is not described here, instead only the key features and information specifically relevant to the topic of this document is briefly walked through to avoid digressing from the subject at hand. Due to the added state mechanism a server can send information it wants to be stored on the client and retrieve it back later when it is needed again. The cookie specification supports the specification of an URL space that the cookie is valid for, so the cookie can be used e.g. by all web servers inside same domain instead of only original host. The length of time for which the cookie is valid can also be specified.

Support for cookies in browsers is very widespread. Unlike form filling functionality,

cookie implementations on various browsers are compatible. The cookie information is useful when user returns to a previously used service. If the service uses cookies, they are a reasonable candidate for storing non-sensitive personal information. However, cookies are not useful if user would like to utilize information that he has already entered on another service, unless the domains and the format of stored information in cookies match. It should be noted that cookies have a bad reputation as far as security is concerned [7, 8], which may cause some security-conscious users to decide not to take advantage of their benefits. Due to the security concerns, the domain matching support also has a few additional restrictions in specification, which makes even legitimate use of this kind difficult to utilize. Cookies also have a common problem with the form fill implementations. The information is stored locally and thus faces the same locality problems which were described in 2.1.1: stored information may be unavailable due to network boundaries other and similar reasons.

2.1.3 Single sign-on

In case of single sign-on (SSO), the personal information is stored by an external, known third party. Personal information as well as all future additions and changes to it are submitted to one or more third parties, which the services can then query for the desired personal information. SSO can also facilitate administration and reduce direct expenses, because e.g. authentication can be moved to SSO service provider. Currently two major competing single sign-on architectures do exist: .NET Passport by Microsoft and Liberty Alliance.

.NET Passport [9], initially released in 1999, is built around a suite of Web-based services. When an user with Passport account wants to log into a service that supports it, initially the user is redirected to Passport.com domain with additional information appended to the URL by the referring service. The information is processed by Passport.com and user redirected again, this time to Passport.net domain. This two-step redirection is due to security reasons. Upon arrival at the Passport.net, the user is asked to present the necessary credentials for the sign-on. After the user has submitted the credentials the browser is redirected to Passport.com, where the information is verified. If the verification is successful, an encrypted cookie containing information about the sign-on is created and issued back to the browser. Now user is redirected back to the original service with some additional encrypted information appended to the redirection URL. This information is used by the service to create two cookies: the first contains authentication ticket information and the second contains any additional profile information that the user has accepted

to share, as well as 64-bit PUID (Passport User ID) that can be used to identify the user without any personal details if necessary. These cookies are then issued to the browser in addition to the cookie sent by Passport service itself. These steps are necessary due to the the security-related limitations of accessing cookies originating from different domains, as described in 2.1.2. The service can now access the cookies to get necessary user information. If the cookie issued by Passport service is already present when user wishes to perform sign-on, the timestamp of sign-on can be compared to the limit the service has set. If the cookie is fresh enough, user will be redirected to the original service as above, otherwise the sign-on to Passport must be done again. A single sign-out is also possible and provided by Passport service. It is implemented simply by erasing all stored cookies originating from participating sites.

Customers are free to use the Passport without any charge, but service providers have to pay an annual fee. This may discourage smaller service providers and limit the adoption of Passport to somewhat larger service providers [10]. However, a freely available development kit which allows the development and testing of services before purchasing the actual service access does exist. Passport also provides so-called Kids Passport Service, where parents can control what personal information their offspring is able to share with the services. Originally Passport had some limitations concerning browsers that were accepted to utilize its all services, but Microsoft has later relaxed the set of limitations considerably. Passport has predefined commonly used information items such as name and address. Some of these can also be defined to be shared with other services using Passport if user explicitly chooses to do so. Other information is stored by the services themselves. Passport has gained some negative reputation due to some serious security concerns [11, 12] in the past, which may somewhat discourage more security-conscious users to embrace it. As described in previous paragraph, Passport uses encrypted cookies as temporary storage of personal information and authorization credentials on the computer that the user is currently using. Thus, Passport does not work if user has disallowed the use of cookies or the browser does not support them. Microsoft has announced [13] that future Passport versions will include support for *Kerberos-based authentication* as well as for the current proprietary solution.

Liberty Alliance [14] project was founded in 2001, with purpose to provide open SSO architecture framework. In general, Liberty Alliance architecture is a more open-ended and relies more on existing open standards than the current version of Passport. It has three types of participants: *users, identity providers* and *service providers*. In this respect it is similar to Passport. A provider may also acts both as a service and identity provider if necessary. Amongst other things it defines a concept known as *federated network iden-*

tity, which forms a combination of the various identities of the user on the Internet in different services. This means that the personal information of the user is not stored centrally to the same extent it is done in case of Passport. Instead, personal information can be exchanged between service providers as well as between identity provider and service providers when required. Obviously, some restrictions are needed in order to control what information gets shared. Different service providers can form *circles of trust* to allow sharing of personal information between them. Users themselves have also possibility to control the extent of information sharing as they feel approriate. The Liberty architecture specifies transfer of information both directly between service/identity providers (Web Services architectural component) as well as using the user agent as a stepping stone (Web Redirection architectural component). These two components as well as the third component (Metadata & Schemas architectural component) are illustrated in figure 3. The Metadata & Schemas component contains the various subclasses of information that are passed between providers as well as the formats used for that. For the Web Redirection component, several different methods of handling and transferring the authentication information and other data are specified, such as cookies and web redirection. The Liberty architecture specification specifically warns about the pitfalls of using cookies and web redirection without any encryption.

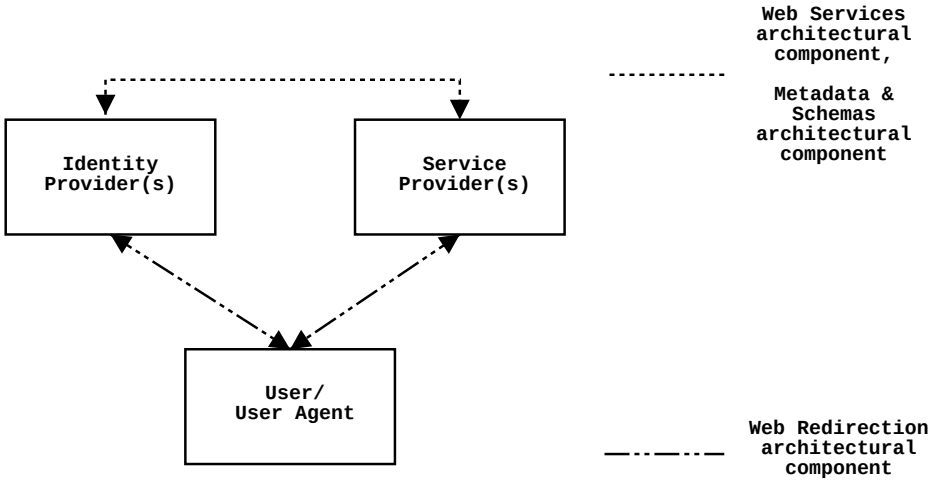

Figure 3: Liberty Architecture

Liberty Alliance has attempted to take in account the existence of some other SSO-related architectures in order to make some level of interoperability between itself and other architectures theoretically possible [15]. As a curiosity, limited interoperability with Passport is mentioned as a possibility. Like Passport, Liberty Alliance also allows the concept of single sign-out: user can perform a sign-out that causes all service providers in the

circle of trust to be informed about the user no longer being signed on.

In case of Liberty Alliance, a somewhat typical (but not the only possible; details can vary concerning details such as whether to use redirection or not) case of the sign-on process of existing user might proceed in the following way: User enters the service with a web browser. If the service provider supports supports several *identity providers*, the user selects the correct identity provider e.g. from a menu. The browser is redirected to the identity provider website, where the user can perform the actual sign-in. After the successful authentication by the identity provider, the user is redirected back to the service provider, which gets informed about the successful authentication. Now the user can use the service normally. Also, if the user has allowed it, the identity federation can be used to share some details of personal information between various services that belong to this circle of trust. The other service providers on the same circle of trust are also aware about the successful sign-on to the identity provider and can allow the user to enter without additional sign-ons. As is apparent, the procedure is not very different compared to the Passport sign-on on the high level. The differences lie on the lower levels, hidden from the viewpoint of the user.

Single sign-on allows a potentially limitless mobility; if the user can access the services that require personal data, the SSO provider is almost certainly accessible too. Network boundaries are largely a non-issue. Operating system and browser coverage is not defined by SSO concept itself, but both major SSO architectures described above should work with most browsers. The limitations on the operating system side are also mainly a potential issue on the server side; because currently most of the SSO operations are done via browser, as long as the operating system allows running of suitable browser, the SSO should work. The downside of SSO is that because it is a relatively new concept, many services are likely to be designed to use more traditional and proven solutions, if any. The possible additional expenses (for example, the annual fee required by Passport) can also be unattractive to service providers. Also, if personal information of several users is stored on a single place, it can become a very attractive target for malicious attacks. Thus, the security is very important for SSO architectures and their implementations.

2.1.4 Summary of solutions

Although all presented existing solutions are at least somewhat useful, all of them clearly have weaknesses (as summarized in table 1) , that reduce their effectiveness in some situations and none of them can be declared so superior to others that all others can be declared to be obsolete. So, a cross-platform and -browser personal secure personal information

management solution that has little or no limitations with the network topologies and is easily used with existing services would be a welcome addition. The success of proposed solution in these as well as other respects is summarized at the end of this document.

Solution	Platform - independent?	Browser support needed?	Immune to network boundaries?
Cookies	Yes	Yes 1)	No
Form fill functionality	Yes	Yes	No
Form fill / cookies and roaming profiles	Partially	Yes	Partially
Single Sign-On	Implementation dependent	Implementation dependent	Yes
Ideal solution	Yes	No	Yes

1) Supported by practically all browsers, so this is a non-issue (unless turned explicitly off by user)

Table 1: Summary of various existing solutions

3 Information about the protocols and standards used

The project can be roughly divided into the following two levels. First of these is the more abstract level that contains the used protocols and standards mostly related about the transfer of the data. Additionally, some reasons why these particular solutions were selected for a certain area of the proposed synchronization solution are explained. The second level includes the more concrete parts of the project; actual implementation tools and platform as well as the structure and details of the implementation itself. Various technologies that belong to the first two groups are introduced in this chapter, whereas the matters belonging to the second group are presented in chapter 4.

3.1 Bluetooth

Bluetooth is a low-power, short-range radio technology. Its radio technology is specified in IEEE (Institute of Electrical and Electronics Engineers) standard 802.15.1, other parts form the specification known as Bluetooth 1.2. However, this document discusses Bluetooth almost entirely from the viewpoint of the specification version 1.1. The reason for this is the current unavailability of mobile devices that implement the Bluetooth 1.2 specification: most manufacturers are not expected to release consumer products conforming to the new specification until second quarter of 2004. Some thoughts about changes in specification version 1.2 that are possibly relevant to the presented concept are discussed in chapter 4.4.1.

Bluetooth operates on frequency band known as ISM (Industrial Scientific Medical), located roughly at 2.4 GHz. It uses spread spectrum technology called FHSS (Frequency Hopping, Spread Spectrum). The frequency of carrier wave switches 1600 times per second between the 79 possible frequencies, located between 2,400 GHz and 2,4835 GHz. The new frequency is determined by a pre-defined sequence generated from the clock and the MAC address of local piconet master device. Piconets are formed when several devices with point-to-multipoint connections are in same area. Piconets can contain up to 8 active devices (one master and 7 slaves) and far more passive (parked) devices. Scatternets consist of multiple overlapping piconets. Bluetooth does also support point-to-point connections. It supports both synchronous connection-oriented links for e.g. voice data (SCO) and asynchronous connection-less links (ACL). The basic structure of Bluetooth protocol stack is presented in figure 4. The following chapters describe some of the most important Bluetooth protocol stack features for the scope of this project. OBEX (Object Exchange) will be discussed later separately and in more depth [16].

Regarding the reasons of choosing Bluetooth as the short-range communication solution for this project, there are three main reasons. First, it is becoming relatively common on even relatively affordable mobile devices, as well as having the advantage of not needing line of sight between transmitter and receiver, unlike IrDa. The protocol stack also contains direct support for location of different services, as described in chapter 3.1.5.

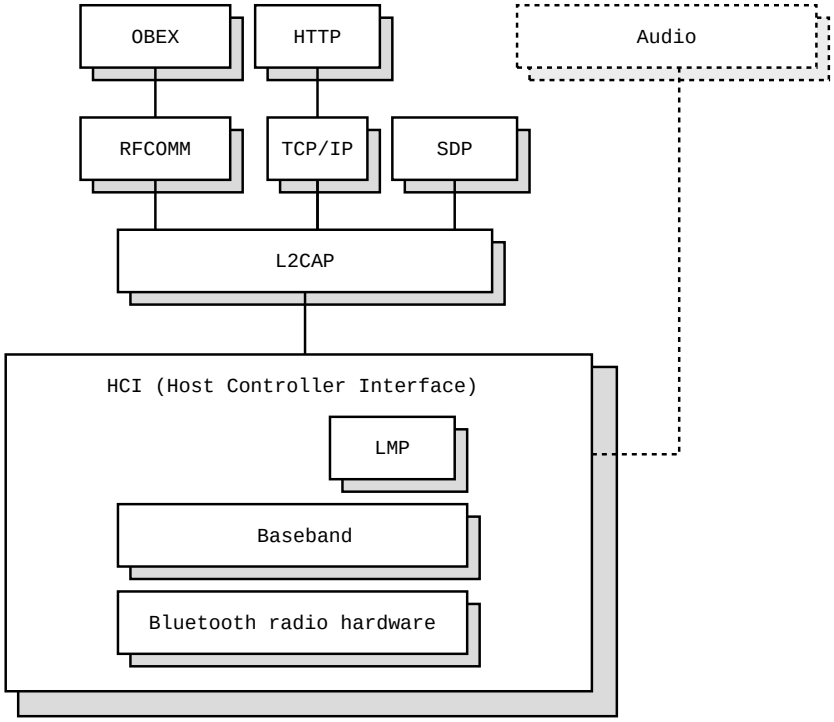

Figure 4: Bluetooth protocol stack

3.1.1 Baseband

Baseband is the lowest actual protocol on Bluetooth protocol stack; it resides on top of the actual radio transmission hardware itself. Some of its duties include the encoding and decoding of the data, power management, management of physical channels and paging/inquiry operations for discovery of other Bluetooth devices. It is usually implemented on the Bluetooth chip itself.

3.1.2 LMP

LMP (Link Manager Protocol) is located just above Baseband on Bluetooth protocol stack. Its features focus around setting up new links between Bluetooth devices as well as control of existing links. Some of the often used link management features include pairing

and link key management. The protocol also contains support for management of Bluetooths authentication and encryption features. LMP messages have higher priority than user data to achieve smooth link management performance, although excessive retransmissions may still cause delays for LMP traffic as well as for any other type of traffic. In addition to the previously mentioned tasks, LMP also performs some miscellaneous duties, such as support for requests to change the transmit power level, QoS, radio/link controller feature inquiries and role switching. Like Baseband, it is usually implemented on Bluetooth chip itself.

3.1.3 HCI

HCI (Host Controller Interface) is an interface that provides standardized way of accessing functionality that link management, Baseband and Bluetooth hardware itself provide. The presence of HCI includes the Bluetooth host system, HCI-specific transport layer and finally the Host Controller on the Bluetooth hardware itself. On host system the HCI is visible as an programming API (Application Programming Interface). This functionality is located on HCI driver, which in turn is located between higher layers and the HCI Transport Layer. The HCI Transport Layer takes care of actual flow of data between the Host and Host Controller. Its purpose is to increase abstraction and thus make the other two components of HCI independent of the physical bus between them. Finally, the Host Controller part exists on the Bluetooth chip itself. The actual physical bus solutions currently supported between Bluetooth hardware and host system are USB (Universal Serial Bus) and PC card (also known as PCMCIA).[16]

3.1.4 L2CAP

The L2CAP (Logical Link Control and Adaption Protocol) layer takes care of multiplexing and demultiplexing several connections over one link and also the fragmentation and defragmentation of packets. Additionally, it implements QoS (Quality of Service) functionality and grouping of links. L2CAP is defined only for ACL links, not for the guaranteed-bandwidth SCO links. [16]

3.1.5 SDP

SDP (Service Discovery Protocol) provides service discovery functionality to devices. Other service discovery solutions such as Jini^{TM} and SLP (Service Location Protocol)

can be used with Bluetooth, but SDP provides a service discovery solution that is designed from the very beginning to support the specific characteristics of Bluetooth such as the wirelessness and mobility. Basically, SDP can be used to perform a search of devices that implement a certain service. Additionally, SDP can also query various characteristics of services. The SDP Server keeps information about each service in a list of service records. Each service record contains a list of service attributes, which in turn contain attribute name and value. The attribute values contain so-called data elements. These contain a header and the actual data. Header specifies the type of data to follow (e.g. text string, unsigned integer or something more complex such as a sequence of data elements) as well as its size.

The PDUs (Protocol Data Unit) that are used for inquiring about service can be roughly divided into four categories. These include error response, service searching, inquiry of attributes of certain service and combination of previous two. A maximum of 12 different service UUIDs (Universal Unique Identifier) can be used in one request while searching services. When inquiring the attributes of service, either a specific service attribute UUID can be used, or a range of acceptable service attribute UUIDs specified. Other features like limiting the maximum amount of attribute data returned in response PDUs are also available.

3.1.6 RFCOMM

RFCOMM (RF communications) offers ETSI (European Telecommunications Standards Institute) TS 07.10 [20] compatible serial port emulation for use between Bluetooth devices. This corresponds to 9-pin RS-232 serial port. The emulation is a subset of the actual standard, lacking support for some frame types and other features that are rarely used or not applicate for Bluetooth. Up to 60 concurrent emulated serial ports can be used. RFCOMM is especially useful when porting existing programs that have previously used serial port as a communication link.

3.1.7 Profiles

Bluetooh profiles are basically lists of functionality and features. If device claims that it supports a certain profile, it must support all features listed for that profile. The idea behind profiles is that it is often much more comfortable to ask other device if it is suitable for a certain use (e.g. as fax) instead of separately checking support for every single feature that is needed in order to the device to work in that way.

The Bluetooth specification states that each Bluetooth device must implement at least GAP (Generic Access Profile) profile. The GAP profile provides abstractions for the basic functionality and interoperability features offered by various protocols in Bluetooth protocol stack. All profiles specified in Bluetooth Profile Specification [18] build upon this profile. Examples of other profiles include telephony-related profiles, (e.g. headset profile), LAN (Local Area Network) access profile for connectivity with traditional networked devices with PPP (Point-to-Point Protocol), OBEX (e.g. Object Push profile), generic Bluetooth access (GAP and Service Discovery Application profile) and transport (e.g. serial port profile).

3.2 OBEX

OBEX is a standard created by Infrared Data Association for exchange of simple data objects such as vCards, pictures and generic files. It corresponds roughly to the HTTP protocol, but it is designed with the limitations of the mobile devices, like comparatively small memory and computing power resources, in mind. Originally it was created to be used with IrDA as its transport, but it is not specifically bound to any particular transport. The goals set for the OBEX in the specification [21] include:

- Application friendly - provide the key tools for rapid development of applications
- Compact - minimum stain on resources of small devices
- Cross platform
- Flexible data handling, including data typing and support for standardized types - this will allow devices to be simpler to user via more intelligent handling of data inside
- Maps easily into Internet data transfer protocols
- Extensible - provide growth path to future needs like security, compression and other extended features without burdening more constrained implementations
- Testable and debuggable

The initial version (1.0) of OBEX was released in 1997. The current version at the time of writing this document is 1.2, released in 1999. OBEX can be divided into two separate parts; the actual protocol and the application framework. The protocol part of OBEX is

located on session layer at OSI (Open Systems Interconnection) model. The similarities between HTTP and OBEX are quite numerous. For example, the OBEX response (status) codes contain the corresponding HTTP status code values encoded as unsigned integers. OBEX supports also the inclusion of real HTTP headers as one of its header type.

Originally, OBEX was to be used as a comfortable higher level data transfer solution in this project, but due to late changes to the project architecture, its use was dropped.

3.2.1 OBEX Application Framework

The application framework is necessary to ensure interoperability between various devices and defines elements to use for basic OBEX services for common object exchange scenarios. Implementation of application framework is not necessary, but interoperability between various implementation can't be guaranteed if an implementation lacks it.

3.3 SyncML

SyncML is a relatively new protocol that has been designed to become a common standard for synchronization of data between devices. With the increasing amount of different mobile devices it becomes also increasingly important that the data can be synchronized between various devices and applications. The previous solutions for the same operation have been more or less vendor-, application- or operating system - specific. Another change that is happening is the appearance of remote synchronization. Previously the synchronization has mostly been local synchronization; the data is transfered e.g. from a PDA with infrared link to a personal computer. These days, however, it is starting to become increasingly common to access information over a network to and from various network services instead of basic point-to-point connection. This is tied closely to the previous point; if different services and devices use different protocols for synchronization, it is difficult or impossible to perform remote synchronization with different peers. SyncML tries to address both of these changes as well as other shifts in the data synchronization usage patterns [22]. At the moment, SyncML transport protocol bindings for HTTP, WSP (Wireless Session Protocol) and OBEX are officially defined, giving already a possibility of performing the synchronization between a wide variety of devices.

These attributes of SyncML as well as its forecasted increasing support make it an attractive choice to use in this project. Another detail in favor of the SyncML is that the specification itself does not include restrictions for transports that can be used for SyncML

to perform the synchronization process, making it suitably open-ended for future development of the presented concept.

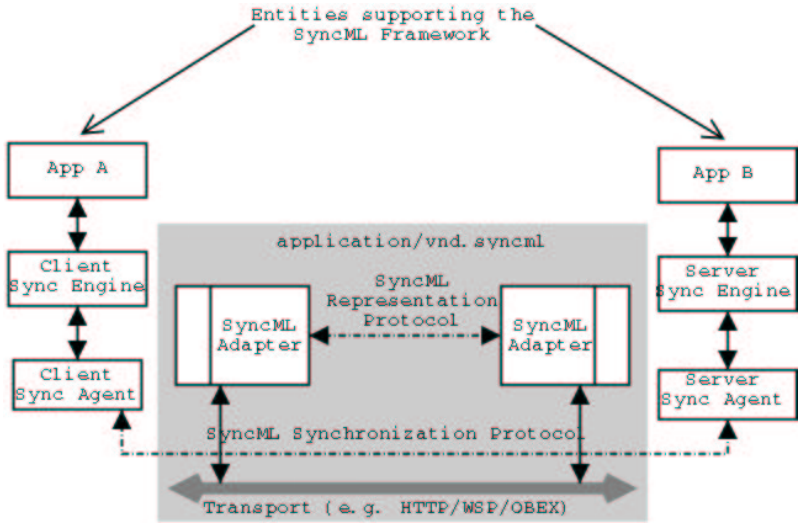

Figure 5: Structure of SyncML Framework (based on a figure in [22])

The structure of SyncML framework is illustrated by figure 5. The framework can be divided into two major separate parts; *synchronization protocol* and *representation protocol*. In addition to these main components of SyncML, the specification also defines *Meta Information DTD* (Document Type Definition) and *Device Information DTD*. Meta Information DTD is used for representation of various kinds of metainformation, whereas Device Information DTD allows devices to exchange information about their capabilities and status. The information is usually encoded as clear-text XML (Extensible Markup Language), but various SyncML DTDs allow also the use of WBXML [23](Wap Binary XML Content Format). This is a binary encoded variant of XML, where the contents of the document are tokenized into a sequence of integers in order to the inherent redundancy of the typical XML document. In most situations “normal” XML will work finely, but using WBXML can sometimes offer notable benefits. For example, due to more compact nature of WBXML, it can be used to achieve efficient use of bandwidth when using slow communication links or performing operations with large amount of data [22].

To complete the explanation of the figure 5 the meanings of terms “Sync Engine”, “SyncML Adapter” and “Sync Agent” must be explained. Basically, Sync Engine is a logical entity that tracks the changes to the local data and on server side also typically manages the creation, manipulation and storage of more complex version information about the data. The information is used to detect and resolve conflicts that arise during the synchronization. This can be either implemented in the application itself or by an external SyncML imple-

mentation. It uses Sync Agent to coordinate synchronization process. The Sync Agent should generate and process SyncML packages that comply with the SyncML Representation and Synchronization protocol specifications. Neither Sync Engine or Sync Agent are covered by the SyncML specification expect as what kind of roles they have; no implementation guidelines are present. Finally, SyncML Adapter is the framework entity that is used for interfacing with the network transport [26, 22].

One of the desired characteristics specified in chapter 1.1 was to have as few limits for the types of information that can be synchronized as possible. In this respect the SyncML framework is a viable choice to use as a part of the proposed solution, because it is open-ended about the type of information that can be synchronized. Most of the currently explicitly defined synchronizable data types are related to personal information (calendar entries, contact lists, emails and so on), which is exactly the type of information required by this project.

3.3.1 Synchronization protocol

Put simply, the *synchronization protocol* part of the SyncML framework defines the how the participants of synchronization must use the other SyncML protocols in order to be able to communicate successfully. In addition to the protocols that are used to do the actual transfer, it's necessary to have information such as when and how to use the various messages to achieve correct exchange of information.

When using Synchronization protocol, the participants must take so-called *Device Roles*. One must take the role of SyncML Client, the other role of SyncML Server. Of these, the SyncML Client role is both less resource-intensive and simpler. Because of this, it is in many cases sensible to have the mobile device to act as a client and PC as a server, if the role of the server includes non-trivial processing. The synchronization protocol also allows two servers or clients to communicate with each other by allowing devices to change their Device Role temporarily. To ensure availability of authentication, Synchronization protocol specification explicitly defines that devices wishing to claim conformance to it must implement at least certain types of authentication. The two types required by the specification are MD5 digest and basic authentication (combination of username and password encoded in Base64 [25]; useful when no real security is required). Once successful, authentication can be valid for entire SyncML session or performed separately for each message. The protocol includes 7 different synchronization scenarios for different needs, called *Sync Types* [22]. An example of synchronization process is presented in 3.3.5, including more information about some of the Sync Types.

It is also the responsibility of Synchronization protocol to provide unique identifiers for individual data items. Each item has an identifier on both client and server. The identifiers are known as LUID (Locally unique identifier) on the client side and correspondingly GUID (Globally unique identifier) on the server side. While the identifiers for the items can be same on both sides, this can not be trusted. Instead, a mapping between identifiers on both sides must be maintained. This is the responsibility of the Server. However, the client is free to create the LUIDs for new item submitted by the server; it only has to send the newly created identifier value to the server so that the mapping information stays valid. The identifiers on both sides make the conflict detection possible. When detecting conflicts, both sides report the identifiers of data items that have changed since the last synchronization. Server compares the lists by using its LUID/GUID mapping table to determine which items are in fact the same and thus have changed on both sides, requiring conflict resolution to be performed. This may also be done by the Client, but is usually the responsibility of the Server due to resource reasons.

Synchronization protocol also includes the concept of *Sync Anchors*, which can be thought as markers that contain information about a synchronization event. It contains typically a timestamp or an unique sequence number. This information should be suitable to determine whether the devices agree about the previous synchronization event. While performing synchronization, the devices send each other a copy of their sync anchors. If the anchors stored on different devices do not match, the devices know that something bad has happened (or the devices simply have not synchronized before) and can initiate an appropriate recovery process by e.g. performing a *Slow Sync* operation. During synchronization, two sync anchors are required. They are known as *Last* and *Next*. The Last anchor contains the information about the last synchronization event between the datastores on Client and Server. The Next anchor provides the same information about the forthcoming synchronization event. When a device receives the Last anchor of the other device, it is compared with the stored anchor to perform the failure detection that was mentioned above. After a successful synchronization, the Next anchor becomes the new Last anchor.

3.3.2 Representation protocol

Whereas synchronization protocol defined how different parts of SyncML specification are to be used together to achieve meaningful information exchange, representation protocol defines the actual syntax and semantics of the messages forming the information exchange in a SyncML session. The protocol is designed around two main concepts. As

the first concept, it introduces a consistent way to identify the data being synchronized - it provides flexible mechanisms to identify individual data items and as well as sets of multiple data items. As the second concept, it provides a vocabulary to express various operations on data, such as insertion, modification and deletion [22].

The identification of data is implemented with Target and Source elements, which each SyncML message contains. These elements contain either URI (Universal Resource Identifier) or URN (Uniform Resource Name) to identify the source or target of the message. It should be noted that their meanings are completely determined by the context. The context may switch even inside the same message. For example, Target elements containing URI in header of a SyncML message typically signifies the destination device or network address of the message. Inside the body of the message the meanings of URIs within Target elements can vary very widely from this. To illustrate the point, Target URI within the Sync command specifies the datastore to be used on the target device. However, the Target URI within MapItem command specifies the GUID, an unique identifier for the information on server side as explained in 3.3.1. These cases have very different meanings for the Target URI. Even the same URI in different contexts might be interpreted differently. The Target and Source element references can also be relative to previous references as in addition to absolute references. Representation protocol also supports TAF (Target Address Filtering). Basically, the purpose of target address filtering is to restrict the synchronization operation to a certain subset of the available information. Several different filtering grammars are defined in the SyncML specification for various types of datastores. This helps to keep the amount of unnecessary information low, for obvious benefits of reduced bandwidth and storage requirements. For the human point of view, this can also be used to transfer just the required information, for example all emails from certain person or all meetings scheduled on next week.

Representation protocol defines syntax for commands that implement the following operations: modifying data, adding data, deleting data, refreshing data and searching data. Additionally, it also defines a so-called container operation which purpose is to allow a number of other operations to be grouped together.

All messages defined by representation protocol consist of a set of Elements. A proper combination of these will create a well-formed SyncML Message. The elements can be divided into the following categories:

- The Message Container Elements
- The Protocol Management Elements

- The Command Elements
- The Common Use Elements
- The Data Description Elements

The Message Container Elements are used to encapsulate a SyncML Message. Three different Elements exist for this purpose; SyncML, SyncHdr and SyncBody.

The Protocol Management Element category contains only one Element; Status. It is used to indicate the result of a command.

The role of the Command Elements is to request actions to be done for the information accessible by the current session. These can be divided further to Data Command Elements, Datastore Command Elements and Process Flow Command Elements. The Data Command Elements (listed in table 2) request the manipulation of application data, whereas Datastore Command Elements request operations that affect the entire datastore. The final category, Process Flow Command elements contains other Command Elements as subcommands and specifies how they should be performed.

Data Command Element	Description of the Element
Add	Adds the specified data items to datastore.
Copy	Creates a copy of an existing item to current or another datastore. Can't convert between types of data.
Delete	Deletes the specified data item from a datastore.
Get	Retrieves the specified item. Most often used for retrieval of Device Information items.
Map	Used by client. Informs server about items added to the datastore by Add command.
MapItem	Contained inside Map command. Holds information about the item UID (Unique Identifier) mapping between Client and Server datastores.
Put	The counterpart of Get. Most often used for sending of Device Information items.
Replace	Like Add command, but replaces existing data item with the contained information

Table 2: Data Command Element requests

The Common Use Elements are elements which are valid when used as subelements for most of the Command Elements and SyncHdr element of Message Container Elements

group. Their purpose is to reduce redundancy both in the grammar specification and in the code required for parsing by providing most of the commonly used features with wide availability.

The Data Description Elements group includes in the current version of Representation Protocol specification just three elements; *Data*, *Item* and *Meta*. The Meta element is used to specify metainformation about the parent element. For example, it can be used to specify the MIME type of information, size of the object and other similar pieces of metainformation. As the name implies, Data elements are used to contain the actual data in messages, as opposed to the information about the data used by SyncML itself. The purpose of Item is more high-level; it separates the actual data (contained inside Data element) and information (metainformation, source, target) from the actual operations.

3.3.3 Meta Information DTD

Meta Information DTD (also known as MetInf DTD) defines various elements that are used to represent metainformation in messages. The purpose of keeping MetInf DTD specification separate from Representation Protocol is to allow use of metainformation elements also outside Representation Protocol. According to Hansmann et al. the elements of MetInf DTD can be divided into three categories: *Content related*, *Dynamic device characteristics* and *Misc Purposes elements* [22].

- *Content related* MetInf DTD elements are used to explicitly specify the content of the data; whether it is character-encoded or binary encoded, the MIME type of content and other details that minimize ambiguity.
- *Dynamic device characteristics MetInf DTD* elements are utilized in synchronization process; actually some of these are practically mandatory for the Synchronization Protocol to work. Other elements of this category can be used to e.g. to inform about the amount of memory available on device and the maximum size of SyncML message.
- As the name hints, *Misc Purposes MetInf elements* include three miscellaneous elements that are used for authentication and and non-standardized metainformation. The final element of this category is the root element of MetInf DTD.

3.3.4 Device Information DTD

The purpose of Device Information DTD [22, 27] (also known as DevInf DTD) is to allow the presentation of the information about communicating devices themselves and their capabilities in standardized format. This information can include things like the version of software used, the manufacturer and model of the device, the amount of available memory, supported sync types, formats of data the device can handle and so on. The information is transfered as a MIME (Multipurpose Internet Mail Extensions) object inside a Data element. Possible uses for DevInf DTD data might include for example the possibility of dynamically selecting format of data to an optimal choice supported by both ends, workarounds/notification for possibly buggy software versions etc. Also, DevInfDTD allows the recognition and use of the device-specific features when appropriate.

As an example, a message that announces the sender to be capable of accepting JPEG content type objects with size of 64 kilobytes might look somewhat like the one presented in Algorithm 1. First, the version of Device Information DTD that the message complies with is defined. Subsequent elements deal with the identifier of the device (e.g. serial number) as well as its generic type (phone, PDA etc). Finally, just before the end of the message, the acceptable size of the objects of this content type is defined. Although the Device Information message is presented here separately to keep the example on topic, in actual message exchange it will be included inside a SyncML message containing other information as well.

Algorithm 1 Device Information DTD example

```
<DevInf xmlns='syncml:devinf'>
<VerDTD>1.1</VerDTD>
<Man>Praxis Phones Ltd.</Man>
<Mod>PX-1000</Mod>
<DevID>1054-1572-1604-1987A</DevID>
<DevTyp>phone</DevTyp>
<CTCap>
<CTType>image/jpeg</CTType>
<Size>65535</Size>
</CTCap>
</DevInf>
```

3.3.5 The synchronization process

As an example of synchronization process, the following case is presented: The synchronization is performed between a mobile device and a personal computer. The synchro-

nization is initiated by the Server, which uses server alerted sync in order to alert the Client to perform the actual synchronization. Only some of the data is modified, so conflicts must be detected and resolved to ensure the integrity of the data. This also implies that simple refresh sync scenarios are out of question, because these types dump all the data to the other end, effectively replacing previous information. This can be compared quite well with the import and export functions in everyday applications such as e-mail and calendar programs, which are used to input or output all available data in one go. Such behavior is not obviously approriate in this case. With these requirements, either *Slow sync* or *Two-Way Sync* are sensible choices.

In Two-Way Sync, the client will send a list of modifications that have happened since the last synchronization to the Server. The Server processes the list, makes potential corrections to its databases and sends information regarding the state of synchronization and possible required changes to the Client. Client now uses this information to update its databases, if necessary. After this is done, Client informs the Server about its status after processing the message(s) received from the Server unless the Server has indicated that it does expect a reply. The Client can send the status report even if not requested by Server, however. If the status report is sent, the Server will respond one more time by acknowledging that it has received the information. In Slow sync, the Client sends all its data to the Server, which then performs the actual operations needed to detect and resolve conflicts. When this is done, the Server should have the differences in data worked out and ready to be sent to the Client. As in Two-Way Sync, the Client can send a status report about the results of the changes caused by the processing of information from the Server. The major difference between this and Two-Way Sync is that with Slow Sync, all information is sent to the Server, instead of only sending the information that has been changed since the last synchronization.

For the purpose of this example, the Slow Sync is chosen and explained in more detail. However, because the initiative for the operation is performed by the Server, the actual Sync Type is so-called Server alerted sync. This type is not an actual Sync Type in the same meaning as the previously mentioned types are. Instead, it only sends an alert to the Client, asking it to perform a certain type of sync. When client agrees and starts the operation, the part of process that could be called the actual synchronization begins. This operation can be divided logically into several different phases. The phases are associated in Synchronization Protocol with concept of packages, numbered from 0 to 6. Each package may contain one or more SyncML messages. A typical reasons of having several messages to form one package can be for example a small buffer for outgoing data or a transport protocol with limitations regarding the message size.

The synchronization process happening in this example is illustrated in figure 6. The first phase is the Server Alert Phase, in which the Server sends the alert to the Client to perform the sync. This phase happens only when Server alerted sync is performed, such as in this case. It is numbered as package #0. When the Client has noticed the package, it proceeds with creation of the actual Slow Sync synchronization session.

The Client now sends the messages that form package #1 to the Server. This is the beginning of the Initialization phase. The package #1 and the corresponding reply from Server (package #2) include negotiation of various session parameters. These include the possible authentication for the whole SyncML session, exchange of device capability information, negotiation of what information to synchronize, which Sync Type to use and checking for the consistency of previous synchronization history. After the Client and Server have performed these negotiations, the actual synchronization of data can begin.

This happens in Data Exchange Phase. Our mobile device sends the selected datastores in collection of SyncML messages that together form package #3. After this has been completed, the Server starts the detection and resolving of possible conflicts. For example, assume that the data to be synchronized includes the email address of the user. If the address has been changed both on Client and on the Server since the last synchronization, the Server must somehow determine which version of the address will be spared. If additional information such as timestamps are available, the choice can be easy. In more difficult cases where no suitable heuristics or additional information is available it may be possible that the Server can not resolve the conflict by itself and merely creates a new data item with unique identifier. After this and other conflicts have been resolved, the Server can send the resulting changes to the Client, which then modifies its own datastores accordingly. This is the package #4 and completes the Data Exchange Phase.

The final phase is known as Completion Phase. If the server has not found anything to be synchronized and has indicated that no response is necessary, this phase may be left out and the synchronization process ends at Data Exchange Phase. As the name implies, this takes care of closing the session after ensuring that the synchronization has been performed successfully. Also, during this phase the Client can send identifier mapping information to the Server inside package #5. This happens when Server has sent completely new information to the Client. The Server requires information about the LUIDs of the new data items in order to maintain a consistent LUID/GUID mapping table, as described in chapter 3.3.1.

Although the phases are listed in strict order in this example synchronization process, the phases may overlap in some situations. This can happens when a package consists of

several SyncML messages. A such package is considered complete only when a Final element of Representation protocol is included in the message. The other party may send message(s) regarding the next package, but not close it before the previous package is marked as closed as well. However, the overlap does not happen automatically in situations with multiple messages per package due to limits placed by the Synchronization protocol.

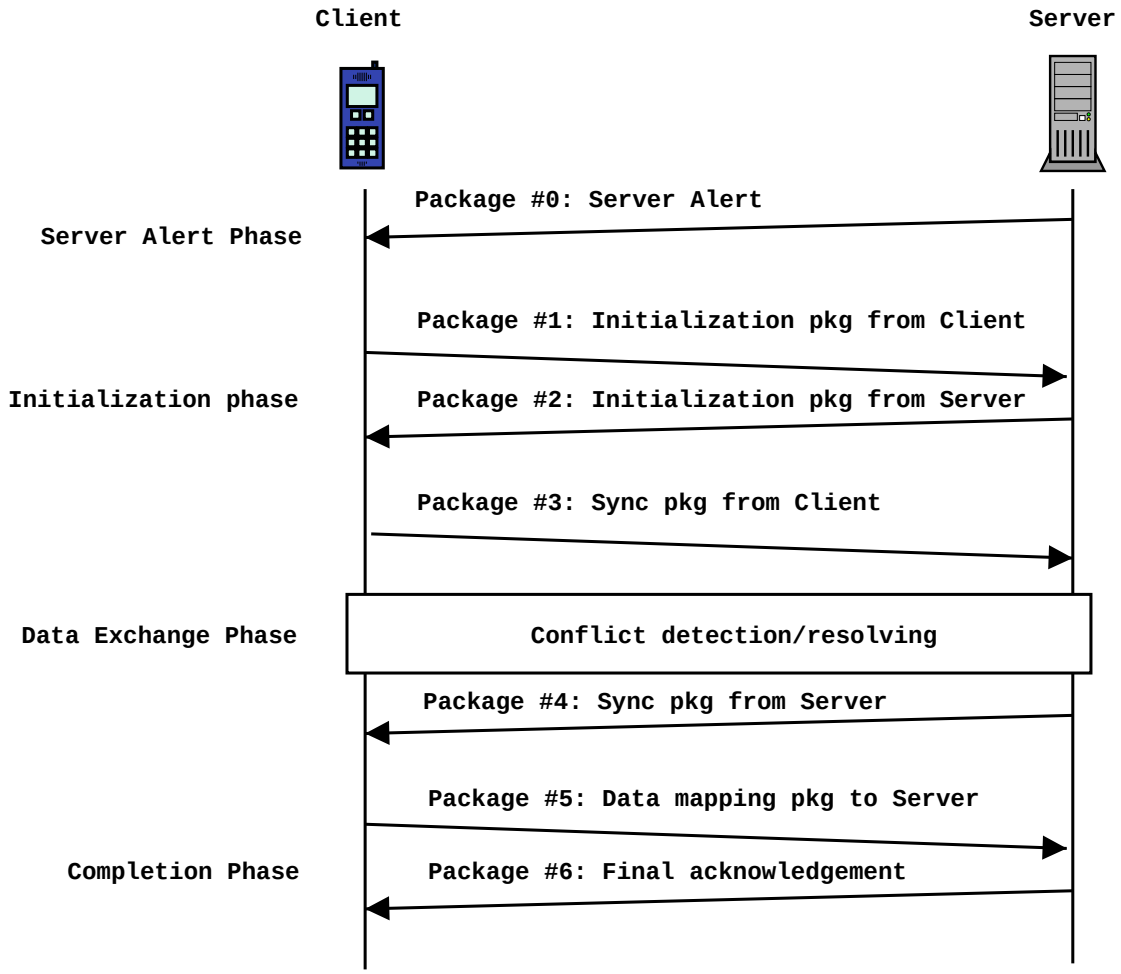

Figure 6: SyncML Synchronization example scenario

3.4 DOM

In addition to the standards related to the transfer (Bluetooth) and representation of personal information (SyncML), the standard for the manipulation of the documents on the web browser is one of the important components in implementation of the presented concept. It is known as DOM (Document Object Model) [28] and is “a platform- and

language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.". Initial version of DOM, known as DOM Level 1, was released in October 1998 and followed in November 2000 by DOM Level 2, the most recent version of the specification that is complete. The latest version of DOM specification, DOM Level 3 is in progress, but not yet completely finished. Also, DOM Level 0 term is sometimes used while referring to the unofficial DOM-like document manipulation features present in older browsers, such as Netscape 3.0 and Internet Explorer 3.0. DOM can be considered to consist of several modules, each covering a different kind of functionality, such as Core, HTML, Style Sheets and Range. W3C-specified bindings of DOM exist officially for ECMAScript and Java, although bindings by third parties for other programming languages are also available. Only a tiny subset of DOM modules and their contents is required to implement the web document manipulation and access functionality required by this project. The focus on this introduction to DOM will be on these areas in order to keep it brief and relevant.

The reasons of using DOM in this project are twofold. First, the use of DOM ensures that any page access and manipulation functionality can easily be implemented on another platform. This is important, because in order to be actually useful, the proposed solution needs to be as portable as possible to be available for a significant number of users. The second reason is the very convenient access to the DOM API implementation of the host browser. Because a complete (within limits of the implementation in particular browser) DOM implementation is available to the plug-in, there is no reason to not use it.

As specified in the DOM Core API, DOM offers a logical representation of document as a hierarchy of nodes forming a tree, regardless of how the browser stores data internally. In DOM Level 2 specification, there are 12 different types of nodes, of which Element is the most relevant one concerning this work. Other significant types of node include attribute nodes, Document nodes and text nodes . With these node types and the more specialized interfaces that extend them, it is possible to represent and modify typical personal information on the web documents. An example of a simplified DOM tree that contains element and attribute nodes is presented in figure 7. The root node of the document is the Document node. All other nodes are its child nodes. The topmost of element nodes is the <HTML> element node. The next four levels of this example tree contain only element nodes. However, the child nodes of the fifth level level element nodes are all attribute nodes.

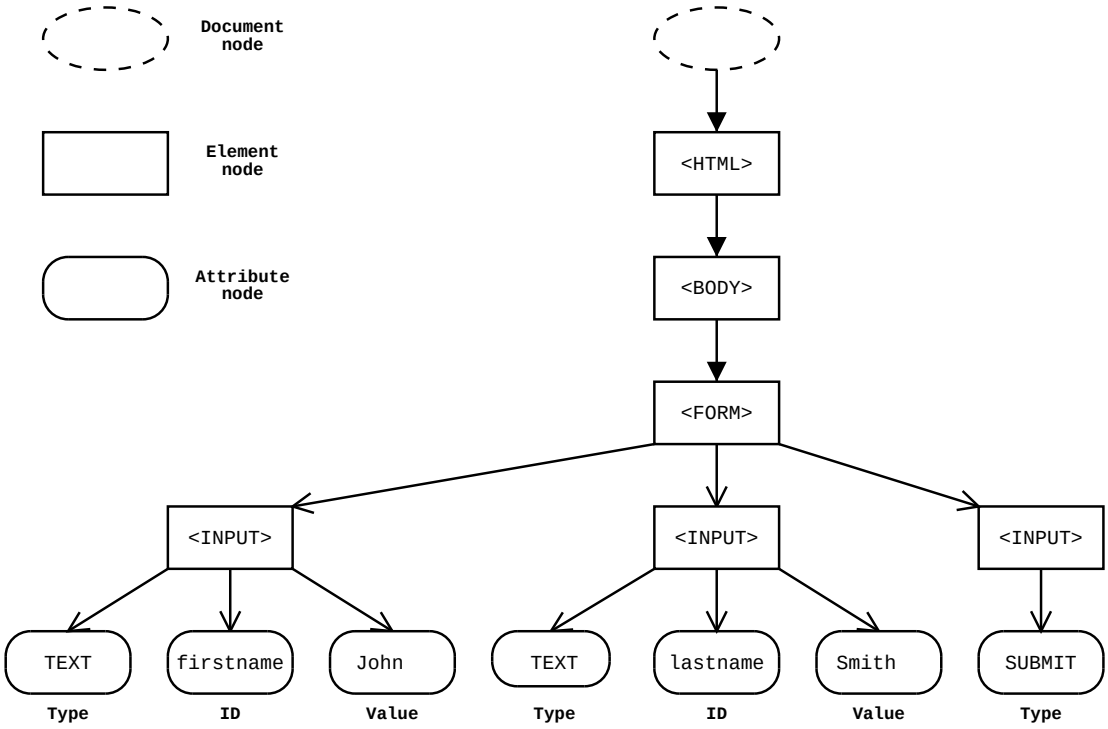

Figure 7: DOM tree example

HTML API extends the Core API by adding objects and methods that are specialized for the handling of HTML (and XHTML) documents. While staying with the Element node type as an example , several new interfaces are made available to take in account the special features of different features. For example, <INPUT> element has very different meaning and attributes than Font element and a single interface that would allow full manipulation and access to both would be unwieldy. Because of this, it makes sense to have specialized interfaces for the low-level manipulation and access of different elements. *HTMLElement* interface offers access to the attributes that every element has, such as the id and language of the element, whereas specialized interfaces exist for the elements that have more specific attributes. To continue with the <INPUT> element, in DOM Level 2 specification the *HTMLInputElement* interface presents access to 18 different attributes and 4 methods. In the figure 7, first two <INPUT> elements has three of these attributes defined and the third only one. No text nodes are included in this DOM tree. This is because the Value attribute defines the initial value of the <INPUT> element, which can subsequently modified by the user if desired. However, as an example of the use of text nodes, when using e.g. <A> element, e.g. <A HREF=*"http://www.foo.bar.invalid">the text to be changed*</A>, "the text to be changed" is a text node.

4 Design and implementation

The design of the system centers around a plug-in component for the Mozilla browser or compatible products. However, the high-level design should still be adaptable for other browser families like Internet Explorer or Opera without major changes. Only implementation details would have to be changed. The high-level design goals for the plug-in include:

- Use of existing components and libraries where possible
- Complies with the standards for Bluetooth protocol stack, SyncML and other protocols used
- Capable to handle vCard information inside SyncML messages.

Because the mobile devices currently seem to have a rather nonexistent support of performing synchronization directly over the combination of Bluetooth and OBEX, the original plan was to have a separate component running on the mobile phone between the mobile device operating system and the plug-in. The component would have bypassed the possible existing SyncML handling functionality on the mobile device and offered service for accessing and encoding/decoding the personal information to and from SyncML, as well as transferring data between the mobile device and the plug-in. The component would have registered itself to the SDP service of the mobile phone in order to be detectable by the plug-in during the search for suitable devices to perform synchronization with. Unfortunately, due to technical restrictions in existing devices that were available this turned out not to be possible within a reasonable timeframe. For this reason, a more direct approach had to be chosen. In the new design the plug-in handles almost all processing and transfer of the personal information by itself, instead of being able to rely on external help for some tasks.

The system can also be thought as an implementation of subset of the ME (Mobile E-Personality) architecture [29], which is presented in figure 8. The aim of the ME architecture is to utilize personal information stored on a PTD (Personal Trusted Device) to largely same purposes that were described in chapter 1. The SAA (Service Addressing Application) is located either on the PTD (Personal Trusted Device), on SAD (Service Accessing Device) or not used at all. Three different cases are specified for the utilization of personal information stored on the PTD. In the first case, the PTD itself is used for accessing online services. The SAA is typically a web browser running on the PTD itself.

In this case, both SAA and ME are located on the same device and SAD is not needed. In the second case, access to online services is performed on an external device, such as a personal computer. In this case the personal computer fulfills the role of SAD and the SAA is typically e.g. a web browser. When personal information is required by an online service, the SAA handles the job of requesting personal information from the ME service. In the third case, SAD is again not needed. Instead, the external services are able to contact the ME service itself directly, without needing SAA at all.

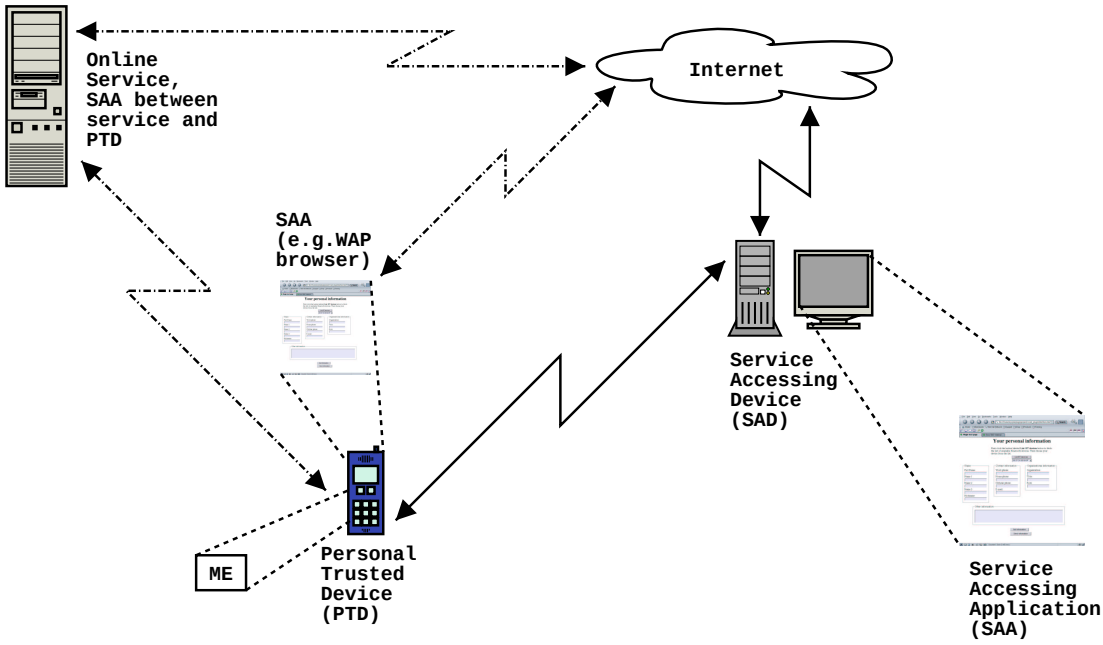

Figure 8: The Mobile E-Personality architecture

The configuration of the proposed solution presented in this thesis corresponds to the second case described in the previous paragraph. The plug-in and its host web browser implement the role of SAA, the host computer the role of SAD and the mobile device the role of the PTD. The aforementioned originally planned and subsequently rejected component would have implemented the ME service. In the figure 8, this particular case is presented with the solid arrows.

The mapping between items of personal information and the web browser is done with the same schema that is used by the Mozilla Form Manager, as described briefly in chapter 2.1.1. A recommendation for form field names to be used with e-commerce does exist [30] and at some point Mozilla might support it [31], but for simplicity and consistency with the current Form Manager design, the schema currently used by Mozilla is suitable enough for the proof-of-concept level implementation as well.

The plug-in component can be divided further into four subcomponents, as illustrated in figure 9. First of these components handles the access and manipulation of the DOM tree that represents the current web page. Second component handles the actual creation, transfer and parsing of SyncML messages. Third component performs the higher-level network operations such as searching suitable devices for synchronization and acts as an abstraction layer to reduce ties to particular transports. Unfortunately, due to used SyncML implementation, it's not possible to isolate all network functionality inside this class; the SyncML toolkit has its own network connectivity implementation, which effectively locks the transfer of SyncML messages to happen inside the SyncML component. The fourth and final component handles the actual transport - specific functionality. The subdivision has added benefit of making future changes to the plug-in easier. The figure also contains ghosted-out links to so-called Corba IDL (Interface Description Language) component, which would be used to provide list of available synchronization - capable devices without need for the plug-in itself to perform scans to find devices with suitable features. The component is not currently available, but is listed for the sake of completeness and as an indication of possible major future change to the design.

4.1 The user interface

The user interface of the plug-in is relatively minimal in order to not disturb the native look of web pages. Three additional user interface components are necessary; a search button, an information synchronization button and a device menu. When user presses the search button, the plug-in begins to search for suitable devices and creates internally a list from the results. The list of devices is presented to the user as a standard HTML menu. User can now select the correct device from the menu and ask for initial synchronization by pressing the synchronization button. The plug-in will compare personal information available on the mobile device to the types of information the web page can accept. For any matches, the corresponding information is inserted to the page if the container element of the information is initially empty. Otherwise, the timestamps of the information on the page and on the device are used to determine whether to replace the existing information on the page or not. This initial synchronization is not necessary, but is probably a good idea in order to fill out as much of the information as possible to keep the required manual information entry minimal. Assuming that the personal information is placed into typical web form fields as is generally the case, information can now be edited freely as any other text, regardless of its original method of input. When the modifications and additions to the personal information are finished, the information can be synchronized again with the

mobile device by clicking the information synchronization button. It should be noted that this can be also used for the initial entry of the personal information into the personal mobile device even if no personal information is yet stored. This method of entering the initial information more comfortable by avoiding the use of the generally clumsy input methods of the current mobile devices, which was listed as one of the possible benefits in chapter 1. The buttons are static elements of the page, whereas the menu has to be (re)created dynamically because the mobility makes static lists of devices unfeasible for any significantly long periods of time.

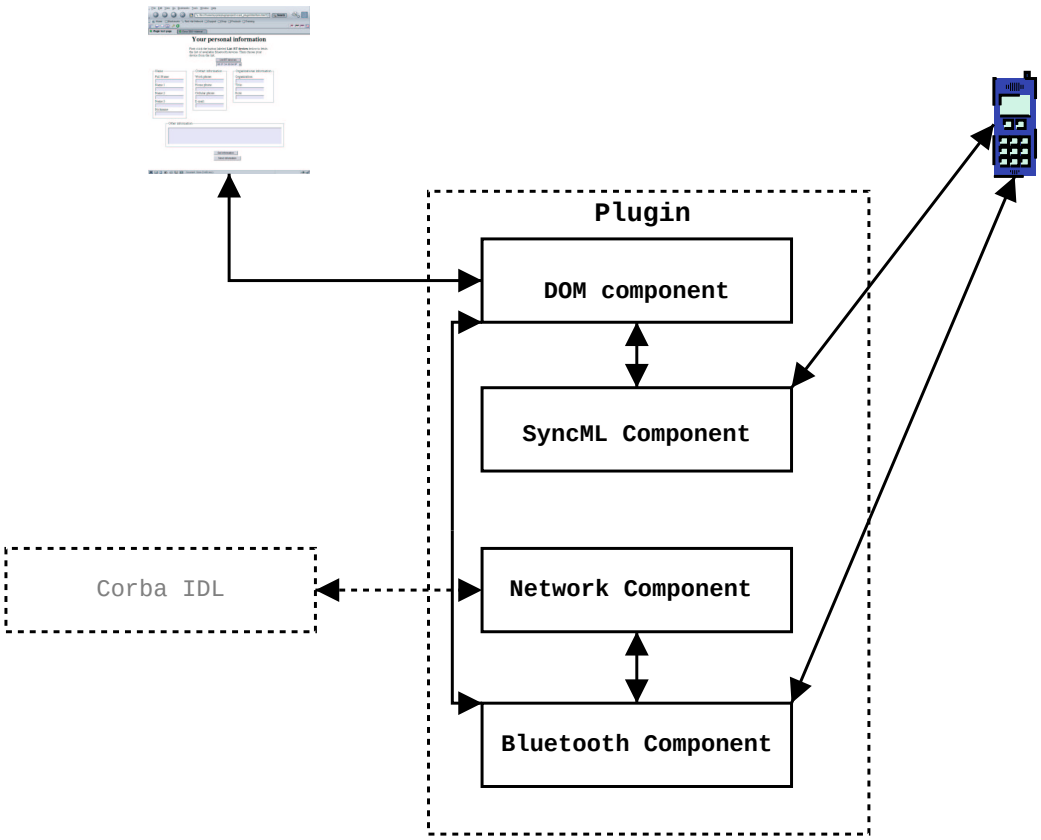

Figure 9: The high-level design

4.2 Mozilla plug-in architecture

Mozilla offers several different ways to create plug-in functionality, including Netscape 4.x - style plug-in API [32] and more recent scriptable plug-in API. The scriptable plug-in API is basically an extended version of the Netscape 4.x plug-in API that has been created in order to allow the plug-ins to regain the lost scriptability. Additionally, plug-ins can be implemented as so called pluglets [33]. The pluglets are plug-ins written in Java and work

otherwise just like the scriptable plug-ins with nearly identical API. Indeed, the purpose of pluglets is to provide a platform-neutral and more compact option of developing plug-ins. Of these choices, the chosen method was the new scriptable plug-in API. Main reason for the choice was that C++ was more familiar as a development language than Java. Documentation for the scriptable plug-in API was also more complete and available in larger quantities than for pluglets. As far as the basic Netscape 4.x - style plug-in API was concerned, it is more or less made obsolete by the more recent APIs.

Technically, the scriptable plug-ins are XPCOM (Cross Platform Component Object Model) components that use XPConnect [34, 35] as a bridge between browser-side Javascript and the components. The objective of the XPCOM is to provide support for reusable components in order to promote creation and reuse of highly modular code. The interfaces for XPCOM components are specified using XPIDL (Cross Platform Interface Definition Language), a slightly modified version of IDL (Interface Description Language). As an example, the XPIDL definition of this plug-in is presented in algorithm 2. The XPIDL definitions can be used to automatically generate source code for various programming languages.. The methods defined in interfaces of these components can be called from browser side with Javascript with parameters and return values. In similar way, variables can be passed between component and browser. Although in this document the XPCOM is mostly discussed in context of scriptable plug-ins, use of XPCOM components is not limited to scriptable plug-ins. Many features of Mozilla visible to user as a single entity are in fact implemented by several interoperating smaller XPCOM components. The synchronization plug-in component methods and their parameters are described with more detail in 4.3.2. The choice of plug-in implementation method also defined the development language to be used to be C++.

Algorithm 2 XPIDL definition of the plug-in interface

```
#include "nsISupports.idl"
#include "nsIDOMElement.idl"
[scriptable, uuid(d1775c66-c073-4603-9650-43a52a3424ae)]
interface nsISyncML : nsISupports {
void findDevices(in nsIDOMElement realityAnchor);
void synchronizeWithDevice(in nsIDOMElement realityAnchor);
};
```

4.3 Implementation

A simple implementation of the previously described solution was created to act as a proof of concept. The development was done on Linux operating system, more specifically using Redhat Linux 7.3 as the base distribution for development environment. The chosen Mozilla version for the development was 1.02; although newer versions have become available during the duration of the project, performing changes to the development environment without truly significant reasons was decided against. However, the necessary APIs in 1.0 Mozilla branch are frozen, which gives a very high probability that the plug-in works with future releases as well. Even outside the 1.0 branch, the API changes should be relatively minor in near future, so the plug-in should work with little or no modifications. For code generation, most of the editing and compilation was done by using rather traditional tools; Emacs and GCC (GNU C Compiler). Because mistakes related to the handling of memory allocation are a common problem plaguing C/C++ program development, a separate debugging tool for such errors was used extensively. Regarded as one of the most advanced open source memory handling debuggers, Valgrind [36] was chosen for the task. The Sony Ericsson P800 smartphone was used in the role of personal mobile device due to its built-in support for SyncML-based synchronization.

4.3.1 The SyncML connectivity of the P800

Because the following chapters refer to the communication features of the mobile device used in testing of the implementation prototype, a description about its SyncML features and connection set-up procedure is presented before going into further details about the internals of the plug-in implementation. The P800 is based on the popular Symbian operating system, which also includes support for SyncML since version 7.0. Thus, the SyncML support and implementation of the other recent mobile phones that use Symbian should be quite similar to one included in P800. The synchronization can be performed over several different physical medias, including Bluetooth, infrared, GPRS (General Packet Radio Service) and USB cradle. All of these options use HTTP transport over a TCP/IP connection created between the phone and the other participant of the synchronization process, regardless of the physical media used. While the use of single transport protocol makes some aspects of the synchronization software building easy, the process of creating the actual connection is somewhat complicated, as described in the next paragraph.

The general principle of creating the TCP/IP connection on the P800 appears to depend

heavily on m-Router protocol. The m-Router protocol is included as a standard component in recent Symbian operating system versions and offers IP - based connectivity over various physical medias and is supposedly based on PPP. The following description applies to creation of connection over Bluetooth, but the principle (contacting the phone over an equivalent of a serial port) should be similar on other physical medias as well. It seems not to be possible to initiate a connection directly from the plug-in or any other program on the personal computer. Instead, the computer must contact the phone, which will then attempt to create a m-Router connection with the computer to set up the actual connection. In case of Bluetooth, this is done by finding the serial port profile of the phone and sending anything to it. The phone will close the connection and immediately try to contact the serial port profile of the computer, if available. This needs the SDP service running on the computer as well as a serial port profile registered on it. Additionally, the computer must be able to respond to the connection attempt performed by the phone. This can be done by running *dund* (a dial-up networking daemon), which starts PPP once the phone attempts to connect. Apparently, the PPP daemon implementation available on Linux systems, once configured with a suitable chat script that corresponds to the m-Router initial messages, is close enough match to the m-Router protocol to work. Once the PPP starts, it creates a TCP/IP connection between the devices in similar way as it would do in any dial-up networking setup. However, the m-Router protocol attempts to resolve hostname “wsockhost.mrouter” and if this does not succeed during a certain time interval, the connection will be terminated. Obviously, this is not acceptable because the connection may be lost before user has finished whatever activities are taking place. Thus, a nameserver with suitable configuration to resolve this hostname must be accessible by the computer. If all of these steps succeed, the connection should be stable. In case of this plug-in implementation, the connection creation is performed very crudely: the plug-in executes an external shell script that receives the hardware address and target RFCOMM channel as parameters. A more elegant solution would be to perform this with the APIs provided by the Linux Bluetooth stack.

After the connection has been created, it is finally possible to use the SyncML functionality of the phone. However, there does not seem to be an apparent way to initialize the synchronization anywhere else but from the synchronization application itself, i.e. no support for server-alerted sync. This is confirmed by the contents of the SyncML messages, where the phone indicates explicitly that it supports only “two-way sync” and “slow sync” synchronization types.

4.3.2 Synchronization plug-in API

The Synchronization plug-in API is the most high-level API on this project. It performs the communication between the browser and the plug-in. All other classes and the services provided by them exist merely to implement the features provided by this API. The API for the synchronization plug-in is created with the use of XPIDL and related tools as described in 4.2, including the API definition. The API must allow the browser to ask the plug-in to perform the following two operations: Discover suitable devices and perform the synchronization between the data available on the mobile device and the browser. The discovery of suitable devices is performed by *findDevices* method. The purpose of *realityAnchor* parameter is related to the DOM manipulation, which is discussed in depth at 4.3.3. The method does not have any return values because the list of suitable devices is made available on the browser side as well. The searching process for the devices uses methods defined in networkTool class. After the devices have been discovered, methods of DOM manipulation class, *DOMTool*, are used to embed the results into the web page. The second method, *synchronizeWithDevice* method is used to call the SyncMLTool class, which performs the actual synchronization of personal information between the plug-in and the mobile device. The plug-in performs the synchronization with the device that is the currently selected item in the device menu. This information is supplied by the methods of DOMTool class, which also handles the insertion of retrieved information in the page.

4.3.3 DOM manipulation

DOM manipulation was implemented by utilizing the built-in DOM classes of Mozilla as building blocks for the methods of this class. The instances of this class must have capabilities to do the following operations: Being both able to read and (re)write the content of certain node as well as create, refresh and hide a list of available devices as menu on the document. Naturally, these capabilities also require that the correct node must be located from the document DOM tree. The methods of this class are listed in table 3. Because various HTML elements have different interfaces in Mozilla (or indeed in DOM; that is the whole point of deriving interfaces for individual elements in addition to functionality provided by interfaces higher in hierarchy), it would need additional effort to support a wide variety of possible elements. As this implementation is a proof of concept rather than a complete real-world product, only <INPUT> form field elements are supported. Furthermore, modifying attributes other than the Value is currently not

supported, so usefulness of e.g. <INPUT> fields with Type attribute value of “checkbox” is very limited or nonexistent. However, with more development resources adding support for a wider variety of elements should not be a major problem. Even extending the support of <INPUT> element that require other attributes than Value would make most of the typical form fields work usable.

Method	Input	Output	Description
doesNodeExist	nsAutoString *name*	Pointer of type nsIDOMElement if successful, NULL otherwise	Checks whether node with ID *name* does exist or not
setNodeContent	nsIDOMElement, nsAutoString *content*	nsresult *retval*	Sets the content of selected (input) node to the string contained in *content*
getNodeContent	nsIDOMElement target	nsAutoString *content* if successful,	Gets the content of selected node and returns it in *content*
createDeviceMenu	GSList pointer to devicelist	nsresult retval	Creates/replaces the list of suitable devices on the web page. Devices must be searched before calling this.
modifyMenuVisibility	int operation	nsresult retval	Controls the visibility of device menu

Table 3: Methods of DOMTool class

The check for existence of the node is performed by *doesNodeExist* method. The pointer to the web page element with ID attribute value “plugin” is obtained initially on the browser side with the DOM API method *GetElementById* and subsequently passed to the plug-in when the Javascript calls the plug-in method from the document. This could also be done by the plug-in itself, if minimal reliance to any Javascript functionality is a priority. This pointer is the same realityAnchor pointer that has already been mentioned in the synchronization API definition. By using this pointer as a starting point, the plug-in can obtain the reference to the root of the current document tree, represented by nsIDOM-Document interface, which is subsequently stored by the DOMTool instance. Once this

is done, the plug-in can check the existence of input elements with suitable ID attribute value by using doesNodeExist method. When the element has been located, it's easy to use the DOM API implementation provided by Mozilla itself to get or set the contents of the element. This functionality is wrapped inside the *getNodeContents* and *setNodeContents* methods of this class. The DOMTool class does not participate in the formatting and parsing of the personal information in any way.

DOMTool class also handles the operations concerning the device menu mentioned in chapter 4.1. The *createDeviceMenu* method starts the search for suitable devices by using networking class functionality and once done, creates a normal HTML menu by modifying the DOM tree. If the menu already exists, its contents are replaced with more up-to-date information. The location of the menu is specified by the placement of <SELECT> element which has ID attribute of "BTADDR". The actual menu items are then created inside this skeleton of a menu, which can be initially invisible and non-obtrusive by defining its visibility DOM attribute to have a suitable value. The plug-in will search for this combination and either create or recreate its contents to correspond with the current list of devices. To have control over the visibility of the menu (for example, to hide it when no devices are detected) *modifyMenuVisibility* method is also included. On browser side, this could be done with Javascript, but it was felt necessary to provide an easy way for the plug-in itself to do it.

4.3.4 SyncML

Mozilla itself does not currently have support for SyncML. Because of this, use of an external implementation was necessary. The choices were to either create a suitable implementation from scratch or to use existing third party SyncML solutions. Because the reference toolkit of SyncML Initiative was freely available, it was chosen in order to reduce development time by not having to create an implementation from scratch.

Originally, the plan was to use OBEX over a Bluetooth connection as a transport. The SyncML toolkit did not include actual code for this transport, although its addition at some point was planned. Thus, some code was added to the toolkit source in order to have OBEX over Bluetooth transport support. This transport would have been used to communicate with the component residing between the operating system of the mobile phone and the plug-in, as described in chapter 4. However, because this component was abandoned, the transport was changed to the built-in HTTP transport of the SyncML Toolkit. This choice was dictated by the fact that the mobile phone used in testing supports only HTTP transport for SyncML. In similar way, the limitations of the mobile phone

were also behind the choice of selecting two-way synchronization as the synchronization type.

Before the description of the actual SyncML class, a short introduction about the general structure of the reference toolkit is presented in order to give useful background information about some of the details that are mentioned in the class description. The toolkit provides functionality to receive and send SyncML messages that reside on memory buffers (known as workspaces). Additionally, the toolkit allows decoding and encoding of the workspace contents. When decoding, the host application must provide a set of callback functions for handling of various SyncML entities. The contents of the current entity are available to the corresponding callback function via a pointer to data structure. The function may perform any processing and additional storage (for example, in different format) of the contained information it deems necessary. When performing encoding, the toolkit provides functions that perform the construction of actual SyncML. The functions include safeguards that are used to ensure that the document is kept valid. This helps to prevent accidental markup errors that might happen if the generation of SyncML messages would be entirely the task of the host application. What the toolkit specifically does not provide is actual synchronization functionality (i.e. synchronization protocol implementation), which is entirely left to be the responsibility of the host application.

The SyncML class includes 5 public methods, as listed in table 4. The first of these, *getSyncPackage*, is used to keep track of the current synchronization package number outside the *syncMLTool* class instance itself. The detection for end of package is handled internally by the class itself. This is necessary, because the plug-in performs somewhat different tasks in reception and creation of each package. The *receiveSyncML* method performs the actual retrieval of SyncML message from the mobile device. It uses the toolkit to set up a connection with the chosen device, gets the size and MIME type information about the message. After that, it receives data in a loop until the same amount of data as specified by document information has been transfered or an error has occurred. The loop is necessary, because the mobile device may not necessarily transfer the whole SyncML message at once due to buffer size restrictions or other similar reasons. Because a SyncML package may consist of several messages, it may be necessary to call this method several times. The receiveSyncML method in itself does not perform any processing for the received data. This is handled by the *parseSyncML* method, which requests the toolkit to perform the parsing process for the whole message. As explained before, the various elements of the message trigger execution of a corresponding callback function. In this implementation, the callback functions store the results of parsing process into an internal structure that contains the type of current message component as

well as pointer to the actual data structure provided by the toolkit. During the parsing process, these structures are added into a STL vector that is a private member of the current message class instance. When the whole message has been parsed, the current message class instance is in turn inserted into a vector of message class instances. This allows the information contained in any message to be accessed later if necessary. While not very efficient in terms of resource usage, this should not be a problem with available modern personal computer hardware resources and allows more flexibility in the use of received information.

After a package that includes receiving and parsing of information from the mobile device has been completed, the next package usually includes creation and subsequent sending of a reply message to the device. The creation of one or more messages are handled by *createPackageX* methods, where X is replaced by the numeric value of the corresponding phase. This is not a very flexible approach, because this effectively locks the plug-in to act as a server. However, for purposes of limited proof of concept implementation this is acceptable. The methods construct the response message to the workspace reserved by the toolkit. After the message has been constructed, it is sent to the mobile device by calling *sendSyncML* method. This method is very similar to the receiveSyncML method: the general sequence of communication API calls is almost identical with the sequence used in receiveSyncML method, with only a few changes that change the role of the plug-in from a receiving client to a sending server. For the same reasons that are explained in description of receiveSyncML method, the sending method performs the transmission of the message inside a loop, where the sending operation is repeated until the whole message has been successfully sent or an error has occurred. The above methods are used by the synchronization plug-in API to receive, parse, and reply to messages from the mobile device, until all phases of the synchronization have been covered or an error has terminated the synchronization process.

Any information that does not currently exist on the mobile device at all can be transfered and stored immediately. However, if the mobile device already contains a personal information item with different value existing on the form field, a conflict does exist and must be resolved. In this application, the role of resolving entity falls to the plug-in due to the superior data processing capabilities of the plug-in. The resolving process is based on Sync Anchors and timestamps of the data in form fields, as described in more detail below. Most of the methods return directly the result given by SyncML Toolkit, which is either an error code defined by the toolkit or an indication of success, if the whole method completed successfully.

Method	Input	Output	Description
getSyncPackage	None	int package	Returns the current synchronization package value
receiveSyncML	None	Ret_t result	Wait for a SyncML message from the device
parseSyncML	None	Ret_t result	Parse the received message
createPackageX	None	Ret_t result	Three separate methods; X corresponds to packages 2,4 and 6.
sendSyncML	None	Ret_t result	Sends the constructed SyncML message to the device
shutdownSyncML	None	Ret_t result	Shut down the current SyncML Toolkit instance

Table 4: Methods of SyncMLTool class

The information passed is wrapped inside SyncML messages. For the purposes of easier debugging and general human readability, the messages are not WBXML encoded.

Support for the SyncML sync anchors is implemented by using timestamps as anchor information. On the mobile device, the device itself keeps track of the timestamp by whatever method it happens to use. On the browser side, the timestamps are based on the moment of time when the synchronization has last happened. Additionally, between synchronization the changes to form fields of the current page are also monitored and the timestamps stored. The DOM Level 2 Event API is useful for this purpose, because it offers a possibility to add an event handler to create an alert when a certain part of the document changes. The timestamp of the change event is available via the DOM API, so the information can be used in the process of resolving conflicts as well as updating sync anchors on the browser side.

4.3.5 Networking

Networking class, called networkTool defines an abstract class for common networking requirements. The methods defined by the class are implemented in subclasses for various transports. Parameterized factory method design pattern [37] is used to create objects that implements the interface specified for the networkTool and hide the differences and peculiarities of various transport technologies. Basically, this means that when an instance of network class is required, the desired transport is specified for the constructor and an object of approriate subclass is returned. After the creation of the object the plug-in does not need to know anything about the used transport, because the implementation details are handled by the subclasses. The goal of using the parametrized factory method design pattern is to make addition of new transport classes easier, should such need arise. Currently, only Bluetooth implementations are available and described in 4.3.6. The methods defined by this class are included in table 5.

Method	Input	Output	Description
findDevices	nsIDOMElement pointer to the object element	None	Searches the devices that implement suitable services to make synchronization possible
listDevices	None	GSList devicelist	The method returns a list of devices searched with findDevices call as a linked list
create	int transport	Object of suitable subclass	Creates instance of a requested transport to handle actual operations

Table 5: Methods of networkTool class

The *devicelist* attribute of the class contains the names of the suitable synchronization partners as well as transport-specific information that is useful for connecting to the device. Although in many cases the name might suffice, the specifically chosen additional information guarantees that regardless of the transport, creation of connections can be done easily. The *findDevices* method implementations will overwrite an existing list of devices, or create an initial list of devices in case of first invocation. The invocation of *listDevices* offers the other classes a way of having access to the list of devices created by the networkTool (or more specifically, the subclass that has actually been created according to the requested transport). Finally, the *create* method will make an instance of

a subclass that corresponds to the type of requested transport. This instance will be used for subsequent high-level networking functionality.

4.3.6 Bluetooth transport

The Bluetooth communication was implemented by using BlueZ, the official Bluetooth stack for Linux [38]. The main higher level protocols used from the BlueZ stack were SDP and RFCOMM. The core BlueZ libraries used for the development were the standard BlueZ codebase included with Linux kernel versions between 2.4.20 and 2.4.25. These upgrades were more an side effect of other reasons to upgrade the kernel (e.g. security holes) than due to need of upgrading the BlueZ itself. Some additional components of BlueZ such as SDP libraries and headers were installed separately due to not being included in the kernel sources and later upgraded to fix some bugs present in older versions. The *bluetoothTool* class acts as an implementation of the methods defined in networkTool class. It is the only transport subclass implemented within the scope of this project.

In case of Bluetooth, the findDevices method is implemented by using SDP to discover all devices that implement the serial port profile. The original plan was to use a specific profile that would have been registered to the SDP service of a mobile device by the additional component, as mentioned in 4. Unfortunately, for reasons already explained, this was not possible. Instead, discovery functionality that allows use of the Symbian native synchronization features was implemented. For each of suitable devices, the MAC address and the RFCOMM channel of the serial port profile is stored in order to facilitate connection creation later on without need to search the suitable devices again. The discovery functionality in its current form is not very good, because it has no way of determining whether the device actually supports SyncML or not, resulting to to potential cluttering of the device list. Additionally, devices that implement SyncML support over Bluetooth in different way that the methodology described in chapter 4.3.1 are missed by this implementation (and would not work with the plug-in even if discovery would succeed). The default paging scheme is used while creating the connection due to being the only one that is supported by all Bluetooth standard 1.1 compliant devices.

4.4 Results

The implementation was not completed in the time this document was published, so some parts of the implementation description should be considered to be closer to very detailed design plan than actual reality. At this time, the plug-in can successfully retrieve the first

message from the mobile phone, but not successfully reply to it, let alone perform the actual full synchronization. This unfortunately limits the amount of presentable results that are related to the actual synchronization process.

The project turned out to be difficult to implement. The implementation difficulties focused mostly around the SyncML Toolkit documentation and the connection setup procedure between the mobile device and the plug-in. In addition to actual implementation difficulties, the Bluetooth is not ideally suited to this kind of application in some respects, as described in next chapter. While the toolkit itself was very helpful in some respects (e.g. the parsing of the SyncML messages), the documentation that was available with the used toolkit version was quite sparse and in some places even misleading. In several cases it was necessary to browse the toolkit source code and add additional debug statements to the toolkit itself in order to gain necessary insight about the current situation. As for the connection setup procedure, its complexity and numerous requirements have already been described. In addition to this, it was also difficult to initially find out any information about the generic design concept of connectivity between P800 and other devices.

4.4.1 Limitations of Bluetooth

Although excellent in many respects such as not requiring line of sight, the Bluetooth specification 1.1 features a few details that unfortunately cause some performance and usability loss for this kind of application. The most significant of these is the relatively long duration of connection setup and creation. With Bluetooth 1.1 devices the combined device inquiry and connection times can in order of tens of seconds and even the average time is about five seconds. For comparison, usability studies [39] suggest that acceptable response times have several thresholds, two of which are one second (the attention of the user starts to have gaps) and ten seconds (the attention of the user turns away from the current task). The current Bluetooth implementations miss the first threshold almost all the time and even the second threshold of ten seconds quite often in practice. While some improvements might perhaps be gained by using different paging schemes than the mandatory scheme defined in Bluetooth 1.1 standard, the mandatory scheme would still have to be used before the devices could negotiate the use of another scheme.

However, the incoming Bluetooth 1.2 specification will supposedly include significantly improved connection performance. Improvements in order of over fifty percent have been suggested for the connection times, but it remains to be seen how close these figures are to the reality. However, if true in reality, the violation of the second response time threshold

of ten seconds would become relatively rare situation. This would improve this aspect of the plug-in usability significantly.

Another feature in Bluetooth specification 1.2 that might improve the usability of this work is AFH (Adaptive Frequency Hopping) . AFH may improve the performance heavy under interference with other devices that are using ISM band, although the concept as presented here has relatively minor performance requirements to begin with. Thus AFH is probably in this case significant only during truly heavy interference that threatens to entirely prevent the connectivity and adds little or no benefit in typical situations.

5 Conclusions

The synchronization plug-in presented in this thesis can provide centralized storage of the personal information on a personal mobile device, as well as a convenient way to enter personal information to online services. Because a significant portion of people with mobile devices carry their devices with them almost everywhere, the stored personal information is potentially available for use most of the time. However, a few practical problems remain. The first of these is the inertia of service providers. Unless a significant number of service providers perform the required minor changes to their web pages, the use of the proposed synchronization service remains very limited. Another, somewhat smaller hindrance for the actual use of the plug-in, is that the plug-in should be ported to a wider variety of browsers and operating systems. The share of Mozilla / Netscape 7.x browsers running on Linux is minuscule when the global web browsing audience is concerned. However, this could be quite feasible to solve, because the plug-in is built upon standard solutions and standards. In the viewpoint of the author, the situation can be summarized by saying that while the concept itself seems to be theoretically reasonable, it does not probably have large potential for success at the moment in the real world.

Regarding the possible future changes to the plug-in implementation, an obvious improvement would be to create implementations for the other combinations of transports and physical media in order to increase the amount of devices that can be used as synchronization partners. Although HTTP and IrDA are already supported by the SyncML Toolkit, both the communication abilities of the toolkit and the device discovery related networking functionality need support for several other combinations in order to be truly universal. In order to be useful, the plug-in would also need support for the connection setup procedures for other physical medias and platforms than Bluetooth and Symbian.

References

[1] T9, T9 Text Input Home Page, http://www.t9.com/ [referred 22.10.2003]

[2] Communication Intelligence Corporation, CIC Jot, http://www.cic.com/products/jot/ [referred 22.10.2003]

[3] M. Wahl et al, Lightweight Directory Access Protocol (v3) RFC 2251, December 1997

[4] Mozilla Organization, Roaming - 4.x compatible, available at: http://bugzilla.mozilla.org/show_bug.cgi?id=124029 [referred 09.01.2003]

[5] Mozilla Organization, Roaming Access - keep bookmarks/cookies/history/etc in central repository, available at: http://bugzilla.mozilla.org/show_bug.cgi?id=17048 [referred 09.01.2003]

[6] D. Kristol et al, HTTP State Management Mechanism RFC 2965, October 2000

[7] Microsoft Corporation, Microsoft Security Bulletin MS00-033 May 2000, available at: http://www.microsoft.com/technet/security/bulletin/FQ00-033.asp#B [referred 16.01.2003]

[8] Slemko Marc, Mozilla Cookie Exploit, January 2001, available at: http://www.cgisecurity.com/archive/browsers/mozilla_cookie_theft.txt [referred 16.01.2003]

[9] Microsoft Corporation, Passport Review Guide, June 2003, available at: http://www.microsoft.com/net/downloads/passport_reviewguide.doc [referred 21.08.2003]

[10] Jäppinen P. and Porras J, Analyzing the Attributes of Personalization Information Accecting Storage Location, June 2003

[11] Marc Slemko, Microsoft Passport to Trouble, November 2001, available at: http://alive.znep.com/~marcs/passport/ [referred 25.08.2003]

[12] Qazi Ahmed, Multiple Vulnerabilities found in Microsoft .Net Passport Services, May 2003, available at: http://www.pakcert.org/advisory/PC-080503.html [referred 25.08.2003]

[13] Microft Corporation, Microsoft Expands Passport to Enable Universal Single Sign-in, September 2001, available at: http://www.microsoft.com/presspass/press/2001/sep01/09-20PassportFederationPR.asp [referred 07.01.2004]

[14] Liberty Alliance Project, Liberty Alliance Identity Architecture, March 2003, available at: http://www.projectliberty.org/ [referred 25.08.2003]

[15] Liberty Alliance Project, Identity Systems and Liberty Specification Version 1.1 Interoperability, February 2003, available at: http://www.projectliberty.org/resources/whitepapers/ Liberty%20and%203rd%20Party%20Identity%20Systems%20White%20Paper.pdf [referred 02.01.2004]

[16] Bluetooth SIG, Bluetooth Core Specification Version 1.1, February 2001, available at: http://www.bluetooth.com/pdf/Bluetooth_11_Specifications_Book.pdf [referred 14.01.2003]

[17] Dean A. Gratton, Bluetooth Profiles The Definitive Guide, Prentice Hall 2003

[18] Bluetooth SIG, Bluetooth Profile Specification Version 1.1, February 2001, available at: http://www.bluetooth.com/pdf/Bluetooth_11_Profiles_Book.pdf [referred 15.01.2003]

[19] Jäppinen P, Bluetooth wireless technology based guidance system, October 2001, Lappeenranta University of Technology

[20] ETSI, TS 101 369 (GSM 07.10). Version 6.3.0 ETSI, 1998

[21] Infrared Data Association, IrDA Object Exchange Protocol version 1.2, available at: http://www.irda.org/standards/pubs/OBEX1p2_Plus.zip, March 18 1999 [referred 27.11.2002]

[22] U Hansmann et al, SyncML: Synchronizing and Managing Your Mobile Data, Prentice Hall 2002

[23] Open Mobile Alliance Ltd, WAP Binary XML Content Format Specification, available at: www.wapforum.org/what/technical.htm [referenced 07.01.2003]

[24] SyncML Initiative, SyncML Sync Protocol, Version 1.1, available at: http://www.openmobilealliance.org/syncml/docs/ syncml_sync_protocol_v11_20020215.pdf [referred 25.10.2003]

[25] The Internet Society, RFC 3548 - The Base16, Base32, and Base64 Data Encodings, July 2003

[26] SyncML Initiative, SyncML Representation Protocol, Data Synchronization Usage, version 1.1.1, available at: http://www.syncml.org/docs/syncml_sync_represent_v111_20021002.pdf [referred 13.01.2003]

[27] SyncML Initiative, SyncML Device Information DTD version 1.1.1, available at: http://www.syncml.org/docs/syncml_devinf_v111_20021002.pdf [referred 27.01.2003]

[28] W3C DOM WG, W3C Document Object Model, June 2002, http://www.w3.org/DOM/ [referred 16.09.2003]

[29] Jäppinen P, Porras J, ME: Mobile E-Personality, WSEAS Transactions on Computers, Issue 2, Volume2, April 2003

[30] Eastlake et al, ECML v1.1: Field Names for E-Commerce, April 2001

[31] Mozilla Organization, Form manager should support RFC 2706 (form field names for ecommerce), available at: http://bugzilla.mozilla.org/show_bug.cgi?id=136414 [referred 21.01.2003]

[32] Netscape Communications Corporation, Plug-in Guide, January 1998 [referred 19.12.2002]

[33] Kushnirskiy Igor, Arora Akhil, Java Pluglet API, July 2001, http://www.mozilla.org/projects/blackwood/java-plugins/ [referred 15.09.2003]

[34] John Bandhauer, Scriptable Components (XPConnect), available at: http://www.mozilla.org/scriptable/, 01.02.2000 [referenced 19.12.2002]

[35] Boswell et al, Creating Applications with Mozilla, O'Reilly, September 2002 (also available at: http://books.mozdev.org/index.html)

[36] Julian Seward et al, Valgrind - A GPL'd system for debugging and profiling x86-Linux programs, available at: http://valgrind.kde.org/ [referenced 29.12.2003]

[37] Erich Gamma et al, Design Patterns Elements of Reusable Object-Oriented Software, Addison-Wesley 1995

[38] BlueZ Project, BlueZ Offical Linux Bluetooth Stack, available at: http://bluez.sourceforge.net/ [referred 31.01.2003]

[39] Nielsen Jacob, Designing Web Usability, New Riders 2000