This document is published in:

“Consumer Electronics, IEEE Transactions on, November 2012, 58 (4), 1425-1433. Doi: http://dx.doi.org/10.1109/TCE.2012.6415016 .

SuSSo: Seamless and Ubiquitous Single Sign-on for Cloud Service Continuity across devices

Patricia Arias Cabarcos, *Student Member*, IEEE, Florina Almenárez Mendoza, *Member*, IEEE, Rosa Sánchez Guerrero, *Student Member*, IEEE, Andrés Marín López, *Member*, IEEE, and Daniel Díaz-Sánchez, *Member*, IEEE

Abstract — *The great variety of consumer electronic devices with support of wireless communications combined with the emerging Cloud Computing paradigm is paving the way to real anytime/anywhere computing. In this context, many services, such as music or video streaming, are delivered to the clients using Cloud-based providers. However, service continuity when moving across different terminals is still a major challenge. This paper proposes SuSSo, a novel middleware architecture that allows sessions initiated from one device to be seamlessly transferred to a second one, as might be desirable in the enjoyment of long running media*[1].

Index Terms — Cloud Computing, service continuity, session handoff, personal multimedia devices.

I. INTRODUCTION

A. Problem Statement

Many organizations are seeking to deliver services to their users by means of Cloud-based providers (e.g. video or audio streaming). This is typically motivated by a desire to avoid management of commodity services which, through economies of scale, can often be delivered more efficiently by such providers.

This trend, together with the increasing usage of small portable devices and wireless networks is paving the way to real anytime/anywhere computing. Nowadays, due to the dramatic evolution of technological convergence, the different consumer electronic devices that a user owns have similar capacities and may allow him to access the same applications and services. The usage of one device or another will depend on the context, i.e. on which option suits better to the current situation. Furthermore, users want to enjoy services on the move, which entails dynamic changes of context. As a consequence, a user enjoying a long duration service may desire to keep the same session across different devices during the lifetime of the service consumption, maintaining continuity. In this context, the need of adequate Multi-device Single Sign-on (MD-SSO) technologies comes into scene. MD-SSO is defined as "*Single sign-on for users that crosses devices, i.e. the session is initiated from one device or user-agent, and subsequently transferred to a second, as might be desirable in the enjoyment of long running media, e.g. streaming video*" [1].

A use-case that perfectly illustrates the added value of MD-SSO is depicted in Fig.1: Bob is on his way home listening to his favorite song list; while he drives, the music is reproduced by the car audio system. When he goes out of the car and walks towards the house, the session is seamlessly transferred to his personal music player and it continues with the current song in the same exact point. Finally, when he enters home, the music session is switched again from Bob's player to his home sound equipment. This switching between Bob's personal music player, in-car player, and house music equipment provides service continuity. Another example could be the case where a user arrives home and wants to transfer his browsing session and current activity on social networks from his Smartphone to his TV, which offers a better display and prevents his phone to run out of battery.

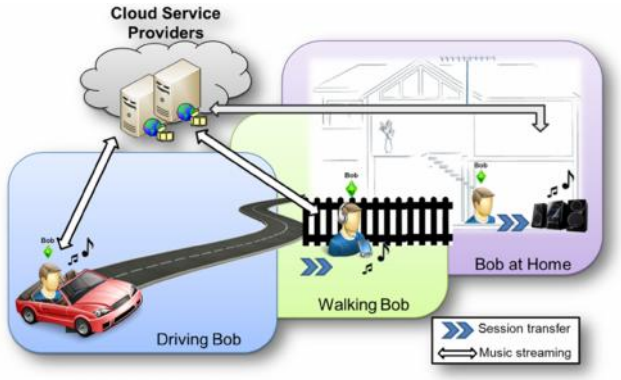

Fig. 1. Multi-device Single Sign-On scenario: a mobile user switches his active session with a music streaming cloud provider between different personal devices, achieving service continuity

Both use cases are feasible with current state of the art technology, and the concept of session mobility across terminals is far from new. In fact, the idea is described as an essential pillar in the traditional literature on ubiquitous computing [2], [3]. Nevertheless, few implementations have been developed and there is no mainstream adoption of MD-SSO technologies, which is especially remarkable and led us to analyze the reasons and propose a solution.

[1] This work was supported in part by the State of Madrid (Spain) under the contract number S2009/TIC-1650 (e-Madrid), and the Spanish Ministry of Science and Innovation under the project CONSEQUENCE (TEC2010-20572-C02-01).

Patricia Arias Cabarcos is with the Telematic Eng. Department, Carlos III University, 28911, Leganés, Madrid, SPAIN (e-mail: ariasp@it.uc3m.es).

Florina Almenárez Mendoza with the Telematic Eng. Department, Carlos III University, 28911, Leganés, Madrid, SPAIN (e-mail: florina@it.uc3m.es).

Rosa Sánchez Guerrero with the Telematic Eng. Department, Carlos III University, 28911, Leganés, Madrid, SPAIN (e-mail: rmsguerr@it.uc3m.es).

Andrés Marín López is with the Telematic Eng. Department, Carlos III University, 28911, Leganés, Madrid, SPAIN (e-mail: amarin@it.uc3m.es).

Daniel Díaz-Sánchez is with the Telematic Eng. Department, Carlos III University, 28911, Leganés, Madrid, SPAIN (e-mail: dds@it.uc3m.es).

B. Related Work and Challenges

Related academic work in the field usually focus on specific modifications to protocols [4], [5], and industry work concentrates on proprietary implementations that only work within a narrow set of devices belonging to the same manufacturer or for a specific application or service. The majority of works focus on SIP-based session mobility [6], [7], so the solutions are tied to the use of this protocol. Besides, some recent approaches, such as [8] and [9] are based on synchronization of session data, which are stored on a proxy or external entity accessible by different devices. As it can be seen, the main problems of the current solutions are related to interoperability across heterogeneous systems. Furthermore, in the case of synchronization, outsourcing the storage of personal data to a third entity raises privacy issues, since sessions may be tracked. Regarding usability, repeated authentication is required when accessing synchronization data from a different device, which is a cumbersome task and imposes an undesirable delay. Therefore, it should be possible to switch between devices in a seamless way without the need to re-authenticate again on each device [10]. Finally, [11] proposes an architecture to support application-level session handoff between heterogeneous devices. This is focused on multimedia sessions (i.e. MPEG), adaptation of resources and management of multiple sessions. Nevertheless, security considerations are not included in the architecture.

In order to overcome the aforementioned barriers, we state that an open holistic architecture is required to guarantee flexibility and allow interoperation across all platforms. Thus, a new layer must be defined and embedded in consumer electronic devices to abstract the complexity of session transference and to achieve service continuity. With these premises in mind, we propose SuSSo (Seamless and ubiquitous Single Sign-on): a middleware architecture aimed at enhancing the user experience when consuming services on the move and maintaining a smooth personal experience even when changing terminals. The original value of our contribution is that serves as a foundation to construct universal MD-SSO for any kind of applications and services, and allows interworking between heterogeneous devices. We also focus on maintaining the security sessions (i.e. authentication/authorization state) when changing terminals to accomplish service continuity based on single sign-on.

Other considerations related to the application properties (e.g. multimedia transcoding or resolution) and to the network level (e.g. QoS parameters) are out of scope of this paper. This is oriented to the application and session levels. In [12] a first approach about transferring trust information and security session data between handheld devices belonging to the same user was proposed. Nevertheless, this proposal requires additional mechanisms in order to achieve the goals defined above, and to include standard languages to achieve interoperability. Our approach involves four well-defined steps: identification of requirements, architecture definition, prototype implementation, and performance evaluation. According to this procedure, the remainder of this paper is structured as follows: section II summarizes the main functional and non-functional requirements to build a generic MD-SSO system. Then, section III gives a global overview of the proposed middleware architecture. Sections IV and V explain the core elements of the architecture, namely the components in charge of handling session data and the mechanisms to transfer this information. Next, section VI describes our work in the prototype implementation, giving some details about performance measurements; and finally, section VII outlines the main conclusions and future research lines.

II. REQUIREMENTS FOR A GENERIC MD-SSO SYSTEM

In this section, we identify a basic set of requirements that will be used as principle guidelines to model the proposed architecture for MD-SSO.

On the one hand, the functional requirements for the system to accomplish the main goal of session continuity across multiple devices are:

- **FR1**. *Context management:* contextual information must be extracted and processed to determine when a session transfer can be performed.
- **FR2**. *State management:* state data of the current running applications in the origin device must be obtained before any session transfer. The security context is especially relevant for future session restoring.
- **FR3**. *Session transfer:* application data as well as security context information must be packed and transferred between different devices.
- **FR4**. A*utomatic session restoring*: the destination device must be capable of automatically restoring the applications that were running on the origin device, as well as its associated security sessions on behalf of the user.

On the other hand, is important that the MD-SSO system fulfills the following basic non-functional requirements:

- **NFR1**. *User centricity*: the system must be user friendly, user controlled and take user preferences into account.
- **NFR2**.*Flexibility:* the system must allow interoperation across heterogeneous platforms and protocols and be capable of accommodating different applications and services.
- **NFR3**. *Performance*: the performance of the system in terms of response time when moving sessions must be reasonable to ensure a smooth user experience.
- **NFR4**. *Security*: communications and data handling by the system must consider security as a primary concern.

How different approaches in the related work cover these non-functional requirements is analyzed in Table I. The functional requirements are covered by such works, except FR4, because they do not address automatic session restoring.

TABLE I
RELATED WORK COMPARISON

Proposal Focus	NFR1	NFR2	NFR3	NFR4
Protocol modification [4], [5]	-	-	NA	+
SIP-based [6], [7]	-	-	NA	+
Proxy-based [8], [9]	-	-	NA	Security + Privacy -
Application level [11]	+	+	+	-
SuSSo	+	+	+	+

Comparison of previous work according to identified non-functional requirements. Symbol + indicates that the requirement is addressed, symbol - means that the requirement is not addressed, and NA stands for Not Applicable.

As we can observe in Table I, those proposals with focus on protocol modification, based on SIP, or those works that rely on storing session data in a proxy server, incur in a lack of user centricity (NRF1). In fact, usability is hindered whether by requiring re-authentication or because appropriate user interfaces to allow easy interaction are not designed.

Another remarkable issue is that the majority of the existing works do not take into consideration flexibility and interoperability (NFR2). That is, instead of providing a generic framework that allows to move different sessions belonging to different applications, they just center in a concrete type of session and give a tailored solution. The flexibility provided by SuSSo, which is based on a *plug-in* mechanism, is one of its main strengths.

Regarding to performance (NRF3), the proposal in [11] cares about having a smooth device to device session transfer. It is to say that the rest of proposals are not comparable from this perspective, because they involve third parties in the architecture or are tied to a specific protocol.

Finally security (NRF4) is addressed by the majority of the works, although proxy-based solutions raise privacy issues derived from storing user data in external servers and [11] does not even take into account security considerations.

All reported approaches deal with at least two of the identified non-functional requirements, but none of the solutions considers all these requirements.

To summarize, the whole set of functional and non-functional requirements previously described will lay the foundations for the generic MD-SSO architecture explained in the next section.

III. SUSSO: MIDDLEWARE ARCHITECTURE FOR MD-SSO

The architecture of the proposed middleware for MD-SSO consists of a series of interconnected software blocks that distribute all the functionality in a modular fashion, as depicted in Fig.2. To clarify the details of these modules, as well as the relationships existing between them, we provide an individual explanation of each one and explain how the functional and non-functional requirements defined in the previous section are fulfilled by the system:

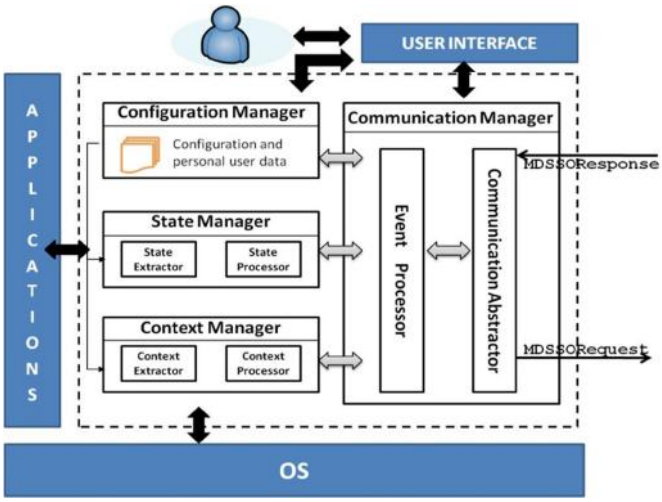

Fig. 2. The SuSSo middleware architecture for multi-device Single Sign-on consists of four modules which distribute all the functionality in a modular fashion: Configuration Manager, Communication Manager, State Manager and Context Manager

A. Context Manager

The *Context Manager* is in charge of extracting context information, such as remaining battery of the origin device, available devices in the proximity, current location or time. This context is then processed by the system to determine if a session transfer can be performed. In order to make this decision, user configuration data provided by the *Configuration Manager* is also employed. Accordingly, whenever a session can be moved to another device an event is triggered to notify the *Communication Manager*. Hence, this module is indispensable for the fulfilling of requirement FR1, as well as it constitutes the basis for ubiquitous session transfer.

It is to note that, there are many research works and real implementations, such as [13], [14], and [15], which are related to context extraction in different kind of devices and for different operating systems. Specially, there are remarkable works considering proximity detection using Bluetooth, RFID [16], or natural interaction [17], which are also applicable here. Thus, the integration of this functionality in our generic MD-SSO architecture is straightforward.

B. State Manager

The main function of the *State Manager* is obtaining the state of the applications that are currently running in the origin device, so it can be subsequently transferred to the destination device. This task is performed by the *State Extractor* sub-module. Similarly, the *State Processor* sub-module is in charge of reading the information regarding the application session transferred to the destination device and restoring them.

On the one hand this module communicates with the *Configuration Manager*, in order to get all the configuration information required to operate. And, on the other hand, this module is also connected to the *Communication Manager*, for

sending and receiving the state of transferred application sessions. It is to note that the state of running applications also encompasses its security context.

The *State Manager* is thus the key for flexibility and it allows the system to fulfill requirements FR2, FR4, NFR2 and NFR4. For the purpose of maintaining flexibility, any application developer or service provider that wishes to offer MD-SSO to their final users, should implement a *plug-in* component for this module. The *plug-in* should conform to specific defined rules (as it will be defined in section IV) and its function is to retrieve the state of the associated application and encapsulate it in a XML file. Furthermore, when a session transfer is received at the destination device, the correspondent *plug-in* will be provided with the XML file with the state information in order to notify its associated application and restore the session.

C. Configuration Manager

The *Configuration Manager* module is responsible of configuring applications in the framework and also handling user configurable information: preferences for session mobility, user credentials that may be required for authentication of terminals on behalf of the user, etc… This information is accessed and manipulated via a user interface, empowering users so they are a central part of the system. Therefore, this module allows the system to fulfill requirement NFR1. This module provides information to all the rest of the modules in the architecture so they can apply the rules as defined by configuration policies. Finally, there are several ways to create these configuration policies: they may be interactively constructed by prompting the user to do so via the user interface; they can be based on the history of the transactions and be dynamically constructed; or a hybrid approach can be employed that combines user feedback and system knowledge.

D. Communication Manager

The *Communication Manager* handles communication with other external devices. It receives notifications from the *Context Manager* when a session transfer is possible. As a reaction to these events, the module asks the *State Manager* to get the information of the running applications that are suitable for a session transfer. Under user consent or following automation policies, the *Communication Manager* sends a package with all the state information required for session restoring in the destination device. Two communication primitives have been defined for this purpose: `MD-SSORequest` and `MD-SSOResponse`, which will be later described in section V. The first message includes data about the sessions to be restored; and the second one notifies the origin device indicating whether the session transfer has been successful or not. It is to mention that different underlying communication technologies (e.g. Bluetooth, RFID, Wi-Fi) can be used depending on availability and/or user preferences. When selecting this technology, security must be also taken into account. More specifically, encryption will be used to exchange all the information transferred between devices and ensure data confidentiality (e.g. by means of Secure Sockets Layer [18]). Therefore, with this module the system fulfills requirements FR3 and NFR4.

It is to note that requirement NFR3 is to be fulfilled by developing efficient implementations of the system.

Finally, we consider that the fundamental blocks in the SuSSo architecture are the *State Manager* and the *Communication Manager*, as they are the basis for a generic MD-SSO system. For this reason, our focus and main contribution is the definition of the inputs and outputs of these modules, as well as their formal operation. Thus, Sections IV and V elaborate on these issues.

IV. SESSION MANAGEMENT

Here we define the input and output formats for session data handling, as well as the mechanisms to achieve session storing, processing and restoring.

Inspired by [1], we consider each active session associated to a running application as a combination of two information blocks: 1) *Application State*; and 2) *Security State*. The *Application State* refers to the context of the application, i.e. what it was doing when it ended at the first device, and that should be reestablished at the second. The specific contents of the context that are to be captured and shared between devices will depend on the application, e.g. the relevant data would be very different for a game than for audio streaming. Accordingly, the format to include these application data is general in order for the system to be flexible. For this purpose, a *plug-in* based solution is employed that allows applications and service providers to incorporate and process application specific context data.

On the other hand, the *Security State* contains the security session information of the application in order to enable Single Sign-On across devices. By using the transferred security context, the second device will be able to re-establish a new security session on the behalf of the user. The *Security State* encompasses both the security data required for user authentication at the destination device, as well as the security information of the current sessions maintained by running applications. The data format for the *Security State* is also aimed at guarantying flexibility. For this reason it may include any kind of assertions and/or tokens (e.g. OAuth tokens [19], SAML assertions [19], etc.) required by the destination device to restore the security sessions with external Service Providers.

Both *Application State* and *Security State* are packaged to be transferred in a `MD-SSORequest`. It is to mention, as it will be explained later in Section V, that the user may wish to transfer several sessions associated to different applications. Therefore, all the contents are packed together in order to be encapsulated and sent in a `MD-SSORequest`.

According to this description, Fig. 3 shows the schematic of a package unit containing session information for applications in the SuSSo framework. More specifically, a protected XML file (*AppSessionFile)* is created for each application whose session is to be transferred. For this *AppSessionFile*, we define

the tags `<AppName>`, `<AppState>`, and `<SecurityState>` that contain the name of the application, the application state data and the security state data respectively. The information is ciphered using a secret user-key, so that only user's devices can use such information. Such secret key may be symmetric or asymmetric.

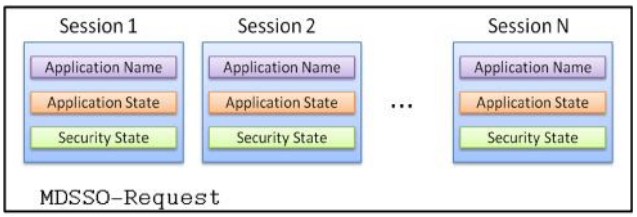

Fig. 3. Schematic overview of a package unit containing session information. Each package may include a number N of sessions; each of one comprises application data as well as security related data

Now the format for session data representation in SuSSo is understood, we explain how the *State Manager* module handles session storing, processing and restoring. Basically, these tasks are performed following a *plug-in* based approach which allows the system to maintain flexibility and to achieve interoperability. Therefore, any application developer or service provider -that wishes to offer Single Sign-on functionality across devices to their final users- should implement a *plug-in* component. The mechanisms for *plug-in* acquisition may vary, but e.g. they can be shipped with the application so the user can download it, install it and configure it for MD-SSO usage in the SuSSo framework.

Every *plug-in* must fulfill the following rules in order to be integrated in the system:

1) Contain an executable file capable of obtaining the state of the application (with its specific information) and generating an *AppSessionFile* with the appropriate tags according to the session format in Fig. 3. This program will be called by the *State Extractor* sub-module, as explained in Section V. We use the name *SessionCapturer* in order to refer to this part of the *plug-in*.

2) Contain an executable file capable of reading an input *AppSessionFile* with session information, and restoring the state of the application. This program will be called by the *State Processor* sub-module, as explained in Section V. We use the name *SessionRestorer* in order to refer to this part of the *plug-in*.

3) Contain a configuration file (*ConfigFile)* that will be read by the SuSSo framework in order to configure and install the *plug-in* on the system. More specifically, we have defined an XML format for the configuration file. According to this, the file must include the tags `<AppName>`, `<AppSessionFile>`, `<SessionCapturer>`, and `<SessionRestorer>`. The first two tags contain the name of the application and the name that will be given to the *AppSessionFile*, respectively. On the other hand, the last two tags are used to indicate the names of the executable files to capture and restore the application session, so they can be later called by the *State Manager*.

V. OPERATION FLOW FOR SESSION TRANSFER AND RESTORING

According to the formats defined in the previous section, we now go into detail about how the session transfer is actually performed. It is worth mention that the transfer of the session is protected using SSL with mutual authentication, although the details are not included below.

The combination of both, input/output formats for session data and operation flow, lay the foundations for the MD-SSO system to be generic and abstract enough to be easily implemented over any OS, and inside any consumer electronic device.

A. Operation flow

The operation flow in SuSSo encompasses the following steps:

Step 1: The applications whose sessions are to be moved between terminals are initially registered in the MD-SSO system, both in the origin and in the destination devices. The registration of an application in SuSSo is performed through the user interface, which allows to select the appropriate *ConfigFile*. This file is read by the *Configuration Manager*, which takes the information and stores it in the system using a *Mapping Table* with entries *AppName*, *SessionCapturer*, *SessionRestorer* and *AppSessionFile*. The image in Fig. 4 shows the structure of the SuSSo Mapping Table and content example:

AppName	SessionCapturer	SessionRestorer	AppSessionFile
nameOfApp1	SessionCapturerApp1.executable	SessionRestorerApp1.executable	SessionApp1.xml
nameOfApp2	SessionCapturerApp2.executable	SessionRestorerApp2.executable	SessionApp2.xml
...	...	...	...

Fig. 4. Storage of configuration data for applications registered in the SuSSo framework

***Step 2*:** Whenever a change of context occurs the user is notified about the possibility of switching the current active sessions. Thus, the user is prompted to enter which sessions he wishes to move. After this selection, a list (*AppList*) with the name of the applications to be moved is generated to be used by the *State Manager* in the next step.

***Step 3*:** The system in the origin device, through the *State Manager*, sequentially calls the *plug-ins* associated to those applications whose sessions are going to be transferred to the destination device. As a result of this action, the *AppSessionFiles* (with application data and security data) required for session restoring at the destination device are obtained. This step is summarized by algorithm `getCurrentSessions`, shown in Fig.5.

***Step 4*:** The system in the origin device, through the *Communication Manager*, builds and sends a `MD-SSORequest` to the destination device. The underlying communication technology may be selected according to several criteria: security, availability, performance, etc.

Algorithm 1: `getCurrentSessions`

```
1: for each appName in appList do
2:  get SessionCapturer in MappingTable(appName)
3:  call SessionCapturer
4: end for
```

Algorithm 2: `restoreSessions`

```
1: for each appName in appList do
2:  get SessionRestorer in MappingTable(appName)
3:  call SessionRestorer
4: end for
```

Fig. 5. Algorithms executed in SuSSo for session extraction and restoring. Algorithm 1 is run by the *State Extractor* in the *StateManager* module of the origin device to obtain the application session files to be transferred. Algorithm 2 is run by the *State Processor* in the *StateManager* module of the destination device to restore the application sessions whose session files were received

***Step 5*:** Upon receiving a `MD-SSORequest` message, the *Communication Manager* in the destination device extracts the session files with the contexts to be restored from the request, i.e. the *AppSessionFiles*, and generates a list (*AppList*) with the name of the applications to be restored.

***Step 6*:** The *State Manager* module iterates over the application list generated in step 5 and sequentially calls the associated *plug-ins*. These tasks are summarized by algorithm `restoreSessions`, shown in Fig.5.

***Step 7*:** The destination device constructs a `MD-SSOResponse` to inform the origin device whether the transfer has been successful or whether any problem exists.

Next, we define the format of the communication primitives and protocol used in steps 4 and 7 to transfer the application sessions from one device to another.

B. Communication Primitives

The session transfer protocol in SuSSo involves two communication primitives: `MD-SSORequest` and `MD-SSOResponse`, which are constructed and processed by the *Communication Manager*.

In Fig.6 we show the format of a `MD-SSORequest`. The header of the message contains the following fields:

- **`ID`** is the message identification number (1 byte). Every response must match the ID of the request being answered.
- **Playload Descriptor (PD)** indicates whether the message is a `MD-SSORequest` or a `MD-SSOResponse` (1 byte). Takes the value 1 for the request case, and 0 for the response.
- **Number of Sessions (NoS)** indicates the number of sessions to be transferred when the message is a request; or the number of sessions that were not successfully restored at the destination device, when the message is a response (1 byte).

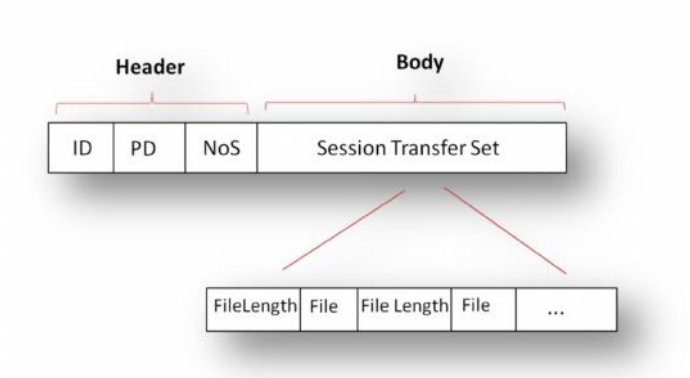

Fig. 6. `MD-SSORequest` message format: header and body fields.

On the other hand, the body part of the `MD-SSORequest` message contains the field:

- **`Session Transfer Set`** consists of number of **`File`** fields (i.e. the *AppSessionFiles)*, each of one preceded by a **`File Length`** field indicating its length (4 bytes).

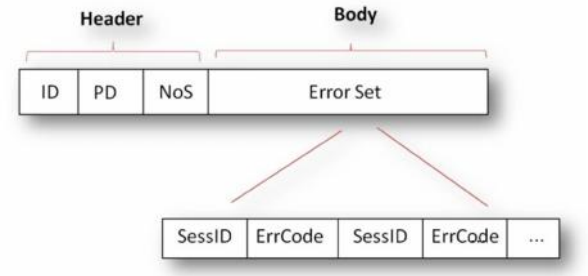

Fig. 7. `MD-SSOResponse` message format: header and body fields.

Similarly, the format of a `MD-SSOResponse` is shown in Fig.7. The header of the message is the same as described for the request. In this case, the body part of the message is composed by the field:

- **`Error Set`** contains two fields per each session that could not be properly restored at the destination device: **`SessID`** and **`ErrCode`**. The **`SessID`** field (1 byte) indicates the position in which the application's session file was sent in the associated `MD-SSORequest` (so it can be identified in the origin device). And the **`ErrCode`** field (1 byte) contains a numeric error code with information about the reasons of restoring failure.

According to the definition of the communication primitives for the SuSSo framework, formula (1) shows how the size of a `MD-SSORequest` can be calculated:

$$SizeOf(MD-SSORequest) = SizeOf\ (Header) + SizeOf\ (Body) = SizeOf(Header) + \sum_{N=1}^{N=NoS}[SizeOf\ (FileLength)_N + SizeOf(File)_N] = SizeOf(Header) + NoS \times SizeOf(FileLength) + \sum_{N=1}^{N=NoS} SizeOf(File)_N \quad (1)$$

Similarly, formula (2) shows the size calculation for a MD-SSOResponse.

$$SizeOf(MD-SSOResponse) = SizeOf(Header) + SizeOf(Body) = SizeOf(Header) + \sum_{N=1}^{N=NoS} SizeOf\ (SessID)_N + SizeOf(ErrCode)_N = SizeOf(Header) + NoS \times (SizeOf(SessID) + SizeOf(ErrCode)) \quad (2)$$

In (1) we can see that the size of a request has a variable part that depends on two parameters: the number of sessions to move (NoS) and the size of each session file ($SizeOf(File)_N$), which is, in turn, application dependent. On the other hand, formula (2) shows that the size of a response has a variable part that only depends on the number of sessions that the system failed to restore (NoS).

Based on these data, together with the transfer rate of the underlying communication technology (Bluetooth, WiFi, RFID, etc.), the protocol performance can be determined.

Next we explain the state of the implementation, as well as the tests performed to validate the proposed architecture.

VI. IMPLEMENTATION DETAILS

A. Prototype Description

So far, we have formally defined the middleware (i.e. communication primitives, storage formats and operation flow) and partially implemented a prototype that involves communication between a mobile phone and a laptop. Based on it, we tested a use-case where the state of the browsing activity is transferred between these two devices. As mentioned before, the *State* and *Communication Managers* are the key components of the system, and so we focus the *proof-of-concept* implementation on these two blocks.

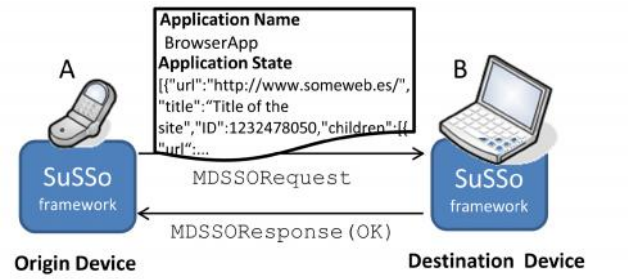

Fig. 8. Prototype conceptual operation: a browser HTTP session is transferred from device A (mobile phone) to device B (laptop)

We have implemented SuSSo in a popular Smartphone OS based on Java and also in a PC. Next, we developed a SuSSo *plug-in* which allows to store and to restore the state of the internet browser (i.e. opened tabs/windows and security sessions with internet services) and conforms to the session data formats defined in section IV. To achieve this, the executables designed for the *plug-in* generate a XML file compliant to the SuSSo format that contains the browser state (tabs, cookies); and, on the other side, launches the browser with the state data read from these XML files.

According to this description, Fig.8 shows a conceptual diagram of the prototype operation.

B. Testing Results

We have executed SuSSo as depicted in Fig.8 to move the browser session between devices. The initial performed tests showed that the state of a running application can be correctly moved and restored at the destination in a reasonable time (~0.5second in average). Furthermore, also in regard with performance, we measured the amount of memory consumption when executing our framework implementation. In Fig.9 we can see that memory consumption is small and it only increases slightly when a transfer is performed.

With the described *proof-of-concept* prototype we aimed to prove that the proposed architecture is easy, generic and feasible. In fact, comparing this solution with existing proposals in [6] and [7], the complexity of our solution is drastically reduced and the implementation is easier, modular and allows interoperability.

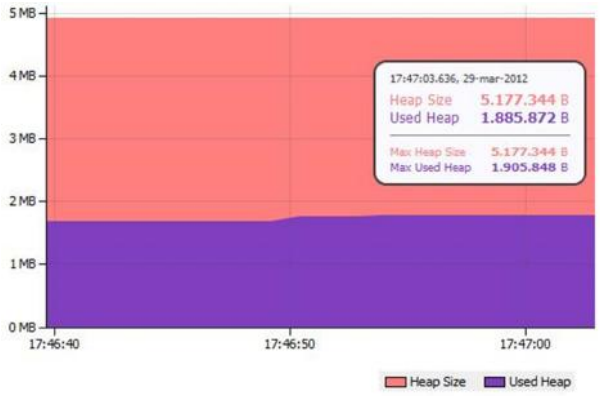

Fig. 9. Memory consumption of the SuSSo prototype executed in a PC with dual core processor (2GHz, 2GHz) and 4,00GB of RAM

Besides this initial test, we carried out a more complex performance analysis, whose results are shown in Fig.10.

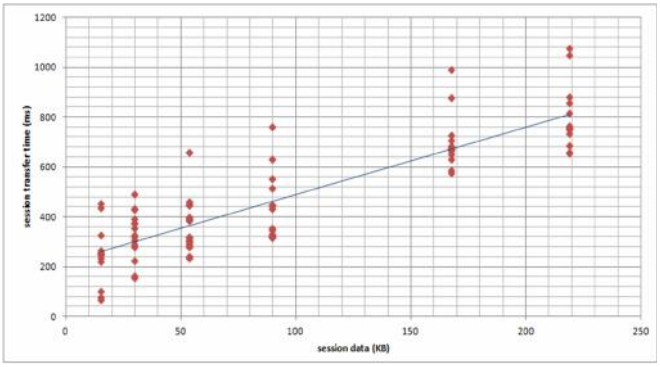

Fig. 10. Time used to transfer sessions from device A to device B, depending on the session size, window number, cookies and history

The graph in Fig.10 shows the time elapsed since the connection is established between the two devices involved in a SuSSo dialogue until the application sessions are restored. We performed 15 repetitions for each experiment and plotted

these values together with the average. The number of sessions is not a useful unit for measures, since there is not a direct correlation between number of sessions and size. In fact, the session size depends on the application directly. For this reason we used kilobytes as unit of measure. In the graph, points for 15, 30 and 54 KB represent measures taken for the transfer of 1, 2 and three sessions respectively; 90, 168 and 220 KB show data for different sessions in different windows, or with browser history. Thus, besides the independency between the number of sessions and the size, we can also derive from these data that time increases linearly with the amount of bytes transferred between devices (as expected from formula (1)). On the other hand, the variability in the time results obtained for the same session sizes is due to the distance between the mobile device and the access point. We observed that there are significant variations depending on the proximity, e.g., a 30Kbytes transfer may vary around ~400ms. These results are very good, because with 8 open sessions (e.g. 220KB), the maximum time obtained has been about 1 second. The minimum time obtained in each measurement was slightly greater in spite of the increase in bytes.

Finally, in order to complement the performance tests, we also provide a qualitative comparison between SuSSo and a synchronization solution developed for a popular web browser. We have installed this latter tool also in a laptop and in a Smartphone.

The working operation for the browser synchronization solution is comprised of the following steps: 1) First, the browser generates a random 128-bit key, called the *sync-key*; 2) this *sync-key*, which is always under control of the user, is used to encrypt the browser bookmarks, history, cached passwords, etc; 3) only the encrypted data is stored at the "cloud", in the browser company server.; 4) to allow the synchronized data to be accessed from a second device, the J-PAKE (*Password Authenticated Key Exchange* by Juggling) algorithm [21], [22] is used to securely transfer the *sync-key* between those devices. Fig. 11 graphically summarizes the described steps.

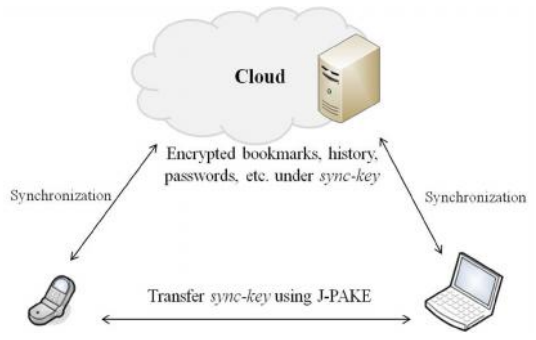

Fig. 11. Web browser synchronization mechanism through of a Cloud used by some commercial solutions

After testing and analyzing the synchronization tool, we compared it with SuSSo regarding several usability criteria:

- **Flexibility:** both approaches can run in different kind on devices, as long as the browser application can be installed. However, SuSSo provides an additional degree of flexibility since it allows users to move sessions from more than one application.
- **Transparency/seamless operation**: Once installed, the synchronization software centrally stores the user's files on the company's servers, automatically tracks the changes, and synchronizes them across the user's devices. All the synchronization process happens in the background, which is transparent to users. Nevertheless, when a user wants to use his data in another device, he has to launch the browser and select a "*use my sync data*" option himself. In this regard, SuSSo avoids this manual step; the session is automatically moved when a change of context occurs (e.g., presence of a device with better capabilities) and the application is also automatically launched by the framework in the destination device.
- **Security:** both approaches take security into consideration. In the case of the synchronization tool the transmitted information is always encrypted. Furthermore, despite this information is stored in the company's servers, privacy is also guaranteed since all the stored data are also encrypted and only the user has the key. In the case of SuSSo, the messages for session transfer are encrypted by means of SSL.
- **Performance:** regarding this criterion, the synchronization tool has a greater impact on performance, i.e., the time elapsed since the user decides to change device until he can use the synchronized data in that device is higher than in SuSSo. This is due to the fact that more manual steps are required, as commented above for the transparency criterion.

According to the above discussion, we can conclude that SuSSo outperforms the analyzed web browser tool in regard to usability, and these advantages would become even more obvious in multi-application scenarios. It is worth noting that the principle of operation followed by each approach is different: SuSSo moves the session data form one device to another and restores the applications, whilst the synchronization approach stores the data in a centralized server where it can be accessed from multiple devices.

To conclude, we aim to stress that the focus of our proposal is in the definition of a generic MD-SSO system; but the *plug-ins* are implemented by vendors and application developers according to the specified formats.

VII. CONCLUSIONS AND FUTURE LINES

Since current solutions for mobility session across devices are proprietary or limited, we maintain that a generic middleware, as the one proposed here, is indispensable to foster healthy progressive adoption by industries and users.

The main contribution of the paper is the design of architecture for multi-device single sign-on that includes the definition of both session storage formats and operation flow for session transfer in an abstract level, so that it can be easily implemented in any consumer electronic device and guarantee interoperability.

We strongly believe that this system constitutes an important step to advance in the field of ubiquitous computing since it clearly constitutes a basis to realize Weiser's well known statement: "*The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it*".

As a suggested extension we consider that the system configuration module can be enhanced in order to provide higher degrees of automation, e.g., by including a prediction component, such as the one in [16], which learns from the user behavior and anticipates his decisions on session transferences. Even, proposals oriented to dynamically adapt to the new environment or device's resources should be addressed, especially in multimedia applications.

REFERENCES

[1] P. Madsen (ed.), "Liberty ID-WSF Multi-Device SSO Deployment Guide", 2008.

[2] M. Satyanarayanan, "Pervasive Computing: Vision and Challenges," *IEEE Personal Communications*, 2001.

[3] M. Weiser, "The Computer for the 21st Century," *Scientific American*, pp. 66–75, September 1991.

[4] Ji-Young Kwak, "Ubiquitous Services System Based on SIP," *IEEE Transactions on Consumer Electronics*, vol.53, no.3, pp.938-944, Aug. 2007.

[5] Ming-Deng Hsieh, Tsan-Pin Wang, Ching-Sung Tsai and Chien-Chao Tseng, "Stateful session handoff for mobile WWW," *Information Sciences*, Volume 176, Issue 9, pp. 1241-1265, May 2006.

[6] Min-Xiou Chen and Fu-Ju, "Session Mobility of SIP over Multiple devices," Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities, TridentCom '08, pp.1-9. 2008.

[7] M. Adeyeye and N. Ventura, "A SIP based web client for HTTP session mobility and multimedia service," *Computer Communication*, pp.954-964, May 2010.

[8] F. Hao and P. Ryan, "How to sync with Alice," Proceedings of the 19th Security Protocols Workshop (SPW), Cambridge, UK, 2011.

[9] C. Soghoian, "An End to Privacy Theater: Exposing and Discouraging Corporate Disclosure of User Data to the Government," Minnesota Journal of Law, Science & Technology, Forthcoming, 2010.

[10] M. Barisch, "Design and Evaluation of an Architecture for Ubiquitous user Authentication based on Identity Management Systems," IEEE International Workshop on Trust and Identity in Mobile Internet, Computing and Communications (TrustID 2011).

[11] L. Rong, I. Burnett, "Application Level Session Hand-off Management in a Ubiquitous Multimedia Environment", e-Business and Telecommunication Networks, part 4, pp. 298-304, 2006.

[12] F. Almenarez, A. Marín, D. Díaz, A. Cortes, C. Campo, and C. García-Rubio, "A Trust-based Middleware for Providing Security to Ad-Hoc Peer-to-Peer Applications", in *Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications* (PERCOM '08), 2008.

[13] M. Abe, Y. Morinishi, A. Maeda, M. Aoki and H. Inagaki, "A life log collector integrated with a remote-controller for enabling user centric services," *IEEE Transactions on Consumer Electronics*, vol.55, no.1, pp.295-302, February 2009.

[14] M. Mulvenna, C. Nugent, G. Xiaoyuan, M. Shapcott, J. Wallace and S. Martin, "Using context prediction for self-management in ubiquitous computing environments," 3rd IEEE Consumer Communications and Networking Conference, 2006 (CCNC 2006), vol.1, no., pp. 600- 604.

[15] U. Christoph, K.H. Krempels, J. von Stülpnagel, and C. Terwelp, "Automatic context detection of a mobile user", Proceedings of Wireless Information Networks and Systems (WINSYS 2010).

[16] P. Arias Cabarcos, R. Sánchez Guerrero, F. Almenárez Mendoza, D. Díaz Sanchez and A. Marín Lopez, "FamTV: An architecture for Presence-Aware Personalized Television," *IEEE Transactions on Consumer Electronics*, vol.57, no.1, pp.6-13, February 2011.

[17] A. Vinciarelli, M. Pantic, and H. Bourlard. "Social signal processing: survey of an emerging domain," *Image and Vision Computing Journal*, vol. 27, no. 12, pp. 1743-1759, 2009.

[18] A. Freier, P. Karlton and P.Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0", RFC 6101, August 2011.

[19] E. Hammer-Lahav, D. Recordon, and D. Hardt, "The OAuth 2.0 Authorization Protocol," IETF draft (work in progress), September 2011.

[20] S. Cantor, J. Kemp, R. Philpott, and E. Maler (Eds.), "Assertions and Protocols for the (OASIS) Security Assertion Markup Language (SAML) V2.0," OASIS Standard, March 2005.

[21] F. Hao and P. Ryan, "J-PAKE: Authenticated Key Exchange Without PKI", Transactions on Computational Science XI, Part II. LNCS, vol. 6480, pp. 192–206. Springer, Heidelberg (2010)

[22] F. Hao and P.Y.A. Ryan, "Password Authenticated Key Exchange by Juggling", Security Protocols 2008. LNCS, vol. 6615, pp. 159–171. Springer, Heidelberg (2011)

BIOGRAPHIES

Arias Cabarcos, Patricia received her Telecom. Eng. degree from Univ. Carlos III of Madrid in 2008 and she obtained the MSc degree in Telematics in 2009. Currently, she is pursuing a PhD at the Department of Telematics Engineering in the Univ. Carlos III of Madrid, working within the Pervasive Computing research group. Her research focuses on the problem of identity management in open and dynamic environments, with special attention to risk analysis and the underlying trust models.

Almenárez Mendoza, Florina (M'07) received her Ph.D. degree from the University Carlos III of Madrid (Spain) in 2006 and is currently an associate professor at UC3M. She received an award-winning as Magna CumLaude in her Computer Science degree. Her research interests include trust management, identity federation, security in ubiquitous computing, and SIM-based applications. She leads the research activities of the PerLab group in advanced trust models, security architectures for open and dynamic spaces, and identity management.

Sánchez Guerrero, Rosa received a Telecom. Eng. degree from Univ. Carlos III de Madrid in 2009 and she obtained the MSc degree in Telematics in 2011. Currently, she works as researcher at the Department of Telematics Eng. in the Univ. Carlos III of Madrid, working within the Pervasive Computing research group. Her research topics include the problem of identity management, security and privacy in healthcare.

Marín López, Andrés (M'07) received a Telecom. Eng. degree and PhD from the Technical Univ. of Madrid in 1992 and 1996 respectively. He lectures in Computer Networks and Ubiquitous Computing in the Univ. Carlos III de Madrid, as an associate professor. His research interests include ubiquitous computing: limited devices, trust, security services, and security in NGN.

Díaz-Sánchez, Daniel (M'07) received a Telecom. Eng. degree from Univ. Carlos III de Madrid in 2002. He graduated as Master Telematic Engineering (2004) and obtained his PhD (2008) from Univ. Carlos III of Madrid. He works as researcher and teacher at Universidad Carlos III. His research topic is distributed authentication, authorization and content protection activities.