Mobility Management and Disconnection Handling for Publish/Subscribe Systems

Gustavo Baptista, Markus Endler
Departamento de Informática, PUC-Rio
gustavo@lac.inf.puc-rio.br, endler@inf.puc-rio.br

Vagner Sacramento
Instituto de Informática, UFG
vagner@inf.ufg.br

Abstract

Advances in computer networks, telecommunications, and portable mobile devices have increased the demand for applications and services that operate well in environments with intermittent connectivity and mobility of devices. A central issue determining the viability of applications in such environments is the employed mobility management, i.e., the automatic maintenance of connectivity between the distributed system when the mobile devices change their IP addresses dynamically while connecting to different network domains. This work presents the development of a SIP Mobility Support Layer, which implements a mobility management mechanism at the application layer based on the SIP protocol. Then, it presents the adaptation of an existing publish/subscribe system to make use of the aforementioned mobility layer. This new publish/subscribe system thus supports the mobility and disconnection of event producers and consumers, enabling seamless asynchronous communication. It also handles NAT and firewall traversal, enabling the system to be used on the entire Internet, and not only within network domains. The referred system has been tested for different scenarios, and its performance has been evaluated for different configurations, and it has been compared to the conventional publish/subscribe system.

1. Introduction

Maintaining the connectivity between components of a distributed application/system in an environment where some devices change their IP addresses entails some challenges, because the Internet was not originally designed to support mobility. Mechanisms to handle mobility in the Internet Architecture are known as mobility management [1]. This work presents the development of a SIP Mobility Support Layer, an implementation of a mobility management mechanism at the application layer using SIP (Session Initiation Protocol) [2] based on the work presented by [3]. This layer has been implemented as an API that can be used by any service or application that requires transparent mobility and connectivity management.

The initial motivation for developing of the SIP Mobility Support Layer was to enhance an existing publish/subscribe system (a.k.a. pub/sub) with the capability to support mobility and disconnection of devices without requiring significant recoding and keeping the simplicity of the API, which can be embedded by many applications or services. This system is called ECI (Event-Based Communication Interface) [4], a content-based publish/subscribe system which is part of the MoCA (Mobile Collaboration Architecture) [5, 6]. In section 4 of this paper we explain how we implemented the aforementioned support in this publish/subscribe system.

Unlike other work about distributed pub/sub systems, which interprets the *mobility of a device* as being the re-association with a new event server (or broker) within the pub/sub overlay network, in our work we consider the device mobility as a predicted or unpredicted change of its IP address. This second form of mobility happens when the event producer/consumer switches between network domains or enters a network protected by firewalls/ NATs, while possibly still being associated with the same event broker. Hence, our notion of mobility support is orthogonal to the common notion of mobility support in distributed pub/sub architectures, and our support can be directly incorporated into distributed pub/sub systems. Thus, the main contribution of our work is this approach to mobility on pub/sub systems with the combination of different concepts and technologies (e.g. application layer mobility management, reliable protocols, disconnection detection) to achieve this goal, which in the best of our knowledge, hasn't been explored on the existing literature.

It is well known that the pub/sub communication model is potentially well suited for mobile distributed systems since the set of communicating peers can be dynamic and peers need not to know each other's ID or address. Moreover, pub/sub consumers and producers don't need to be constantly connected to the pub/sub servers [7]. The pub/sub model is also well suited for the implementation of mobility management at the application layer, because it provides short-lived communication interactions, i.e. it does not need to

maintain long-lived connections active after mobility events.

In addition to supporting mobility, a pub/sub system should also be capable of handling temporary disconnections of event producers and consumers, since this is common in mobile systems. More specifically, in the case of disconnection of a event consumer, the event server should store the consumer's subscriptions and his not-yet delivered notifications, allowing it to receive the notifications as soon as it becomes connected again [8]. Such disconnection support has also been developed and incorporated in our pub/sub system.

The main contributions of this work are the following:

- The design and implementation of an abstraction layer, called JAIN SIP Helper, which encapsulates the use of JAIN SIP API [9] in order to facilitate the development of applications or services that require SIP communication.
- The design and fully functional implementation of a SIP Mobility Support Layer, using the SIP Helper API.
- The incorporation of a SIP-compliant mechanism capable of traversing networks protected by firewalls/NATs;
- The enhancement of a publish/subscribe system (MD-ECI) using the SIP Mobility Support Layer and introducing a customizable disconnection detection and handling mechanism, both for TCP or UDP;
- Performance tests and comparison of the pub/sub systems with and without SIP mobility management and firewall/NAT traversal in a wireless network and using notebooks and HTC's G1 smart phone (running Android O.S.).

The structure of the paper is as follows. In the next section we explain how mobility management is done at the application layer and describe the proposal based on SIP. Section 3 then describes the architecture of our SIP Mobility Support Layer and discusses the approach for firewall/NAT traversal. In section 4 we then explain how this API was used to implement the enhanced pub/sub system and how disconnections are detected and handled. In section 5 we describe the performance tests performed and in section 6 we describe related work. We conclude in section 7 with final remarks and point to future work.

2. Mobility Management at the application layer

Implementing mobility management at the application layer does not require any change on the existing infrastructure of the Internet, and its complexity is quite low since no changes on transport layer protocols are required either. In addition, location management is generally provided at this layer.

The pub/sub communication model provide short-lived communication interactions and the cost to restore them in case of interruption (e.g. in case of a mobility transition) is relatively small, so it can benefit from the simplicity, flexibility and ease of deployment of implementing the mobility management at the application layer [10]. Therefore, we have chosen to implement our mobility management at the application layer, considering the advantages listed above, using the SIP-based mobility management mechanism proposed by [3].

The SIP protocol offers inherent personal mobility, since users use location-independent identifiers to register themselves on Registrars and Location Services and can be found even if they are using different devices such as a notebook, a PDA or theirs PCs, and any request is correctly directed to the device the user is using [11]. The solution proposed by Wedlund and H. Schulzrinne in [3] uses SIP also to support terminal mobility, allowing a user´s device to change its IP address during an active SIP session. First, it is assumed that the user belongs to a domestic network, on which there is a domestic Registrar that receives registrations every time the user´s IP address changes, in a similar way of the Mobile IP´s Home Agent registration (but in SIP the entity being tracked is the user, not the device). When the User Agent of a correspondent host sends an INVITE request to the user, the SIP Proxy Server has current information about the user´s location (the real IP address of the User Agent which the user is using) and redirects the INVITE request to the user. When a device moves (i.e. when it detects a change on its IP address, polling the O.S. or receiving an asynchronous notification from it) during an active SIP session, it must send a new INVITE with the same session id (also known as REINVITE) directly do the other participant´s User Agent, indicating on the Contact field of the INVITE request the new IP address on which the user wants to receive new SIP messages. In addition, on the SDP (Session Description Protocol) [12] part of the session, it must update the addresses related to the exchange of data. The other User Agent must detect that this new INVITE is not intended to initiate a new session, but to update the existing session. So, it must use the received data to update the session with the new IP addresses and then continue to exchange data normally.

3. Implementation of the SIP Mobility Support Layer

This Section presents the implementation of the SIP Mobility Support Layer, which will be referred from this point simply as SIP Mobility. The SIP Mobility is a reusable API that implements a SIP User Agent specialized to provide the aspects necessary to the SIP-

based mobility management. This API encapsulates the details and the complexity of manipulating the SIP protocol, making easy its use by any application which needs a solution like this to support the mobility of devices. The main entities and modules of the SIP Mobility API are presented below.

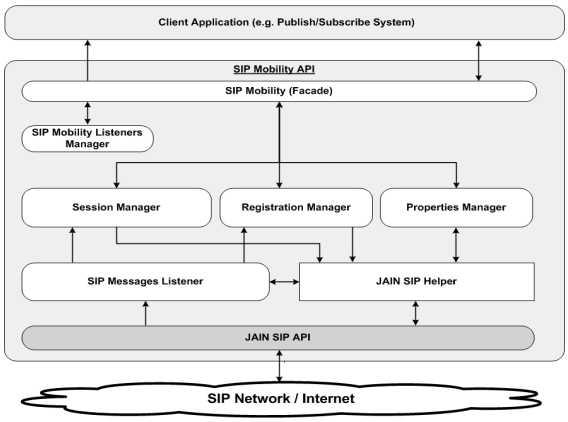

Figure 1 - Modules of the SIP Mobility API

SIPMobility. Is the facade for the API. It provides to applications access to its main functionalities. According to each method called by the application (methods shown on Figure 2), the facade calls the module responsible for its processing (Arrows A on Figure 1).

The facade contains methods for: The initialization and termination of the API execution, through which the application passes all necessary parameters (the calls to these methods are forwarded to the PropertiesManager module).

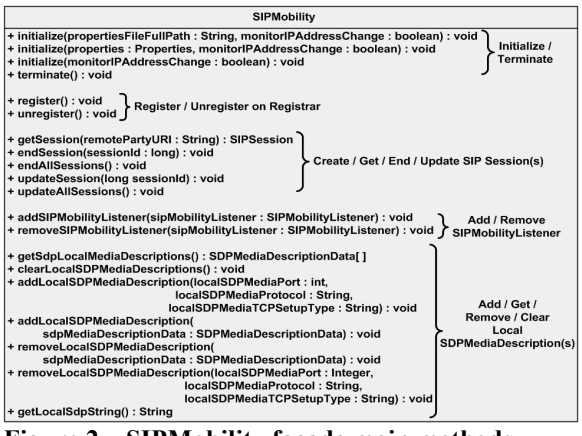

Figure 2 – SIPMobility facade main methods

The inclusion or removal of a user registration into/from a Registrar (the calls to these methods are forwarded to the RegistrationManager module). The creation, termination or update of SIP sessions (the calls to these methods are forwarded to the SessionManager module). The inclusion or removal of Listener objects from the application (the calls to these methods are forwarded to the SIPMobilityListenerManager module). The inclusion, retrieval and removal of local media descriptions, specified in SDP [12] (the calls to these methods are forwarded to the PropertiesManager module).

SIPMobilityListenerManager. As the SIPMobility is an API based on a protocol which exchanges messages asynchronously, the SIPMobilityListenerManager module implements the Observer Design Pattern [13], providing methods for the registration of observer objects (also known as Listeners) that are used to send to the application the API´s main execution events. The application must register objects implementing the SIPMobilityListener interface (shown on Figure 3), which make them able to receive execution events from the API.

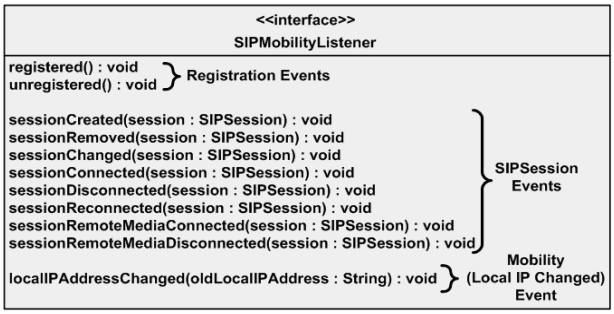

Figure 3 – SIPMobilityListener interface main methods

The SIPMobilityListenerManager is the responsible for managing all Listener objects registered by the application. When some module of the API needs to notify the application about the occurrence of an execution event, it calls this module to notify all registered Listeners (Arrows B on Figure 1). This way, the application can be notified when any of the following execution events occur inside SIPMobility: The registration or removal on/from a Registrar, the creation, removal or change of a SIP session, changes on the communication state of a SIP session, such as when a session is connected, disconnected, re-connected or when the state of the media being used changes between connected or disconnected or the change of the device´s local IP address, possibly due to mobility.

SessionManager: Is the responsible for managing SIP sessions. It sends SIP requests and receives SIP replies for the establishment, update and termination of these sessions. For example, when an application requests the establishment of a session, this module performs the exchange of all necessary messages, such as INVITE, OK, ACK, etc. For the exchange of SIP messages, this module uses the JAINSIPHelper module.

RegistrationManager: Is the responsible for managing registrations on a Registrar. It sends REGISTER messages to a Registrar with the user´s location-independent identifier and the device´s current IP

address. It also contains a thread which periodically sends registration updates to the Registrar with the device´s current IP address. These updates are sent on a rate of a third of the time specified as the registration expiration on the SIP server, or are immediately dispatched in case of a detection of a change on the device´s IP address.

PropertiesManager. Is the module responsible for managing all properties necessary to instantiate the SIPMobility and its configuration. This module also keeps properties related to the use of the SIP protocol, more specifically for the JAIN SIP API (inside the JAINSIPHelper module).

JAIN SIP API: The JAIN SIP [14] is a Java API developed by NIST (National Institute of Standards and Technology) to offer Java developers the capacity to use the SIP protocol. It provides the manipulation of each detail of this protocol, allowing the developer to build SIP User Agents with total personalization. For example, with this API, one can define on the developed SIP User Agent the messages format, the methods for sending these messages and the handlers for the received messages, and also which replies will be sent for each type of received message.

JAINSIPHelper: The abstraction level of the JAIN SIP API for the entities of the SIP protocol is not high enough to make the use of the SIP protocol easy and fast. For facilitating the use of the JAIN SIP on this project (and also for any other project or application), we have developed an abstraction of this API´s main functionalities, allowing the main parameter of each functionality to be specified and to let the JAINSIPHelper interact with the JAIN SIP API transparently. Thus, the difficulties to build SIP messages and to perform operations specific to the JAIN SIP are encapsulated, facilitating the system development, and improving its clarity and its maintainability. Therefore, all modules of the SIPMobility use the JAINSIPHelper to perform their operations. For example, when the SessionManager needs to send a request (e.g. INVITE) to establish a session, or when the RegistrationManager needs to send a REGISTER request to register the user on a Registrar (stream arrows C on Figure 1), these modules can call the JAINSIPHelper module passing all necessary parameters and the module gets responsible for building the SIP message and sending it using the JAIN SIP API. The PropertiesManager module also uses the JAINSIPHelper (stream arrows D on Figure 1) which in turn uses the JAIN SIP API for converting and validating data (e.g. user´s URIs).

SIPMessageListener. This module implements the SIPListener interface (from the JAIN SIP API) and it is added as a listener object to the JAIN SIP´s SIPProvider to receive SIP messages from the lower layer. It processes arriving SIP messages and passes them to the appropriate module for their processing. For example, requests or responses for the establishment of sessions (e.g. INVITE, OK, ACK, etc) are passed to the SessionManager module, and requests or responses related to the registering of the user on a Registrar (e.g. REGISTER, OK, etc) are passed to the RegistrationManager module (stream arrows E on Figure 1).

SIP Servers. For using the SIP protocol, servers which perform the roles of the main entities of the SIP network, such as the SIP Proxy, Registrar and Location Service are necessary. The SER (SIP Express Router) and Open SER are open source implementations of a SIP Server which perform the role of all necessary entities, and also provide many other features, such as user authentication. These implementations are consolidated and used by a large community of SIP users worldwide. Another SIP Server implementation is the SIP Presence Proxy, developed by the creators of the JAIN SIP API. Although this implementation is more intended for experimental purposes, it provides a graphical tool for the visualization of SIP messages that pass through the SIP Proxy, which is very useful for debugging the protocol.

Local SDP description. As explained before, the SIP protocol is used for signaling, and it is independent from the media (exchange of data, voice, etc) being used by the application. When a SIP Session is established, each participant uses the other participant´s SDP specification to sent data to him. Therefore, every SIPMobility instance must be configured with a local SDP description of the communication media to be used for established sessions, so they can be offered to the other participants. So, the SIPMobility offers methods for the application to configure the local SDP description with any desired number of media descriptions to be used. Additionally, to facilitate the manipulation of the SDP protocol, the SIPMobility offers methods that only require the application to specify the media parameters (such as IP addresses, port numbers and communication protocols), and the SDP local description to be automatically generated inside the established SIP Sessions. These parameters can be specified in the SIPMobility´s initialization properties. However, it can be necessary for the application to change the local SDP description after the API´s initialization. For example, in case it needs to change the communication port it is listening on. To enable this, the SIPMobility also offers methods to allow adding, removing or changing media descriptions (SDPMediaDescriptionData objects) on/from the local SDP description used by the SIPMobility (last group of methods on the Figure 3).

SIP Communication Sessions. The main logical entity that is used for the interaction between SIPMobility

instances (which are in fact SIP User Agents) is called SIPSession, which represents a SIP Communication Session, which is a logical binding between users. Each session contains the following data: The location independent identifier (URI) of the user it belongs, the parameters related to the SIP communication, such as IP address, port and communication protocol (TCP or UDP) which the SIPMobility is using to send and receive SIP messages, the parameters related to the exchange of data, such as the IP address, port and communication protocol (TCP or UDP) which the application is using to send and receive data (the term media is used to refer to the exchange of data between session´s participants).

SIP Session's states. This distinction between SIP communication data from the media data is necessary because the SIP protocol is only a signaling protocol, independent from the application´s exchange of data (the communication media). The media data must contain the SDP description of the local SIPMobility and the remote SDP description (i.e. which belongs to the other participant of the SIP session). A device can be connected for the SIP protocol, but at the same time be disconnected from the media specified inside the SDP session description. Thus, to monitor the Session´s state, two state indicators are necessary, one indicating the SIP communication status and other indicating the media status. The Table 1 shows the possible states for the two indicators inside a SIP Session. The SIP Communication Status indicator contains the state of the SIPSession established by the SIPMobility. For example, when the SIPMobility sends an INVITE message for the SIPMobility on the other side of the communication, inviting the other participant for the session establishment, the SIP Communication State remains INVITING until some kind of answer is received. The Remote Media Status indicator contains the communication state, indicating whether data can be sent to the other session´s participant. For example, if the application is not successful on sending data to the addresses specified on the session description (e.g. with the TCP protocol, a failure on the sending occur, or with the UDP protocol an acknowledgement is not received). To represent such situation, the application would set the Media Status indicator to DISCONNECTED, stating that it was not possible to perform the sending of data. It would not make sense to set the SIP Communication State to disconnected, because the SIP communication could still be working normally through an established SIP session. This situation would indicate that although the SIPSession had been established, some error on the other side of the communication is preventing the receiving of data on the IP address and port specified on the SDP session description.

SIP Communication Status	**UNDEFINED, DISCONNECTED, INVITING, RINGING_THERE, BUSY_THERE, RINGING_HERE, REINVITING, CONNECTED**
Remote Media Status	**DISCONNECTED, CONNECTED**

Table 1 – Possible states inside a SIP Session.

SIP Sessions updates and support for terminal mobility. SIP Sessions can be updated by both session participants. For example, when a device changes some data on its local SIP Session representation or in its local SDP description, it must update the sessions it has established so the other participants can have these updates (the participants would have their SIP Sessions and Remote SDP description updated). To update a session, a REINVITE is sent to the other participant updating the session and its states. A REINVITE contains the same session identifier, indicating that the session should be updated. This mechanism is the same used for supporting terminal mobility, as explained in Section 2. In the next Section we illustrate how this feature is used by the client application, in this case our pub/sub middleware.

4. SIP-based Mobility Management applied to a Publish/Subscribe middleware

As already mentioned, in addition to the development of the SIP Mobility Support Layer, this work also presents the extension of an existing pub/sub system to support mobility and disconnection of devices. This Section presents the extension of the ECI middleware, applying the use of the developed mobility layer.

The ECI is a content-based pub/sub system, with a classic centralized architecture. It is composed by a Java API for the event publishers and subscribers called ECIAgent, and a Java API which represents an event broker called ECIBroker. The APIs can be used by any application that need a publish/subscribe content-based communication. On this work, these APIs are restructured and extended for the inclusion of mechanisms to handle the mobility and disconnection. The system product from this extension is called MD-ECI (Mobility and Disconnection Support for the Event-Based Communication Interface). First it is assumed that the ECIBroker must be established on a server at the fixed network, because it is the point of intermediation for all communications, and a disconnection of this component would lead to a total halt of the system [8]. The subscribers and publishers (ECIAgents) are established on portable mobile devices, and can move between different networks changing their IP addresses.

The SIPMobility is used by the MD-ECI to enable support to mobility of event producers and consumers. Therefore, the SIPMobility API is used by both ECIAgent and the ECIBroker, allowing them to establish SIP sessions among them.

Registration of ECIAgent and ECIBroker on user's domestic network's Registrar. On the instantiation of the ECIAgent or ECIBroker, the URI (Uniform Resource Identifier, a location-independent identifier such as a string of the form user@domain) of the user must be informed (on the ECIBroker it must be created a URI of a fictional user representing the event server, such as broker@domain).

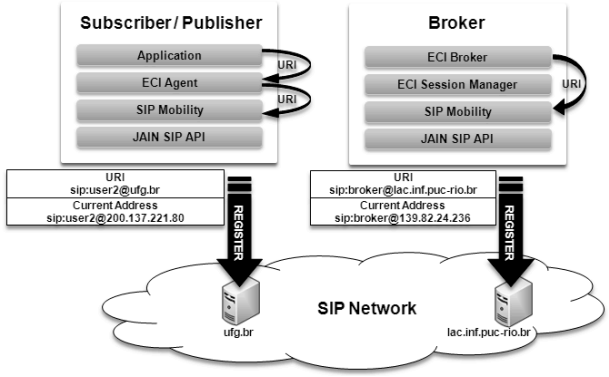

Figure 4 – Registering on the SIP Proxy Server of the user´s domestic network

The SIPMobility API then sends a REGISTER message to the SIP Proxy Server (which contains a Registrar) on the user´s domestic network (which must be configurable) with the mapping of the user´s URI to the real IP address of the device his is using (see Figure 4).

Session establishment between ECIAgent and ECIBroker. At the ECIAgent, the URI of the ECIBroker must be configured, so when the ECIAgent is initialized, its SIPMobility establishes a SIP session with the SIPMobility contained in the ECIBroker, session that will be maintained during all lifetime of these two entities (see step 1 on Figure 5).

Established a session between the ECIAgent and the ECIBroker (see step 2 on Figure 5), the ECIBroker stores the session on its ECI Session Manager, so it can be retrieved when necessary. The ECIAgent only needs to store the unique session established with the ECIBroker. Both sides of the communication can then use the data of the SDP descriptions to send and receive data, such as subscriptions, notifications and any other type of message (see step 3 on Figure 5).

On the original ECI, a user and his subscription were identified by the user´s device IP address and port. Instead, on MD-ECI, the ECIBroker uses the user´s URI and the SIP sessions are used as the logical entity to organize the interaction with the users.

One of ECI´s main logical entities is the Subscription, which contains the expression of interest (i.e. a filter over the content) of events which the client wishes to receive. On the original ECI, a Subscription was also identified by the IP address and port of the device that sent that Subscription, and these data was used for the sending of notifications. With the use of the SIP Mobility Support Layer each Subscription is identified by the URI of the user it belongs. Thus, for sending a notification to the user, the ECIBroker uses the URI contained in the Subscription to find the user´s SIP Session, and to send the notifications to the real network address contained inside the SDP description which the user has specified on the session.

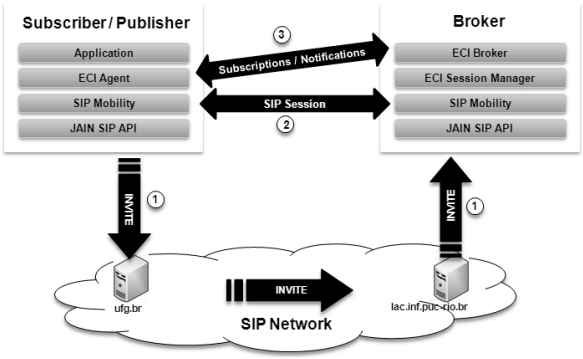

Figure 5 – Session establishment between ECIAgent and ECIBroker

Terminal Mobility Support. As already mentioned, the SIPMobility API provides support for terminal mobility for the MD-ECI. The Figure 6 shows how this functionality is implemented using the SIP protocol. The RegistrationManager module has a thread which monitors the device´s current IP address. When this module detects the change of the device´s address (possibly because of mobility), is dispatches the sending of a new REGISTER message to the Registrar on the user´s domestic network, in order to allow the user´s device to be located on its new network address (step 1 on Figure 6). Any existing active SIP Session is updated by the sending of a REINVITE message to the other participant (step 2 on Figure 6).

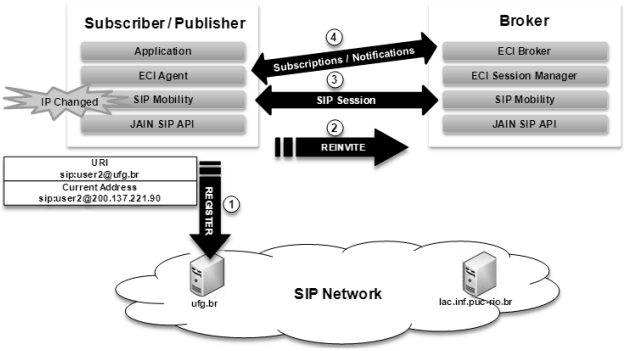

Figure 6 – Recovery of a session after the mobility of a publisher/subscriber

Thus, the SIP and SDP data, together with media data are updated with the new IP address on both participants of the session. When the REINVITE is processed, both the SIP Communication Status and the Media Communication Status states are defined as

CONNECTED, because it is assumed that after a session renegotiation, both participants have their media descriptions compatible with the IP addresses and ports on which they desire to change data (step 3 on Figure 6). Finally, both session participants can restore the normal exchange of data (step 4 on Figure 6).

Disconnection detection and handling by the ECI Agent and ECIBroker. As all steps involved on the session renegotiation caused by mobility require some time, on which ECI event notifications cannot be delivered to the consumer, the service must also provide support to the temporary disconnection of clients (event publishers and subscribers).

To handle disconnections, first it is necessary to define how this state is detected by the system´s components. The Figure 7 shows how this detection takes place. As explained in Section 4, the Media Communication State (Remote Media State) contained inside the SIP Session indicates whether the other participant can be contacted on the addresses it has specified on his SDP description. For example, a communication participant A, to send data to other participant B, gets the remote SDP description from the SIP Session (Remote SDP field), which contains the address on which the participant B is waiting to receive data. If the communication protocol to be used was TCP (which offers reliable data delivery), an exception thrown by the Socket during the sending of data would mean that the participant B is disconnected, because it cannot receive data on the address specified on the session description. However, for publish/subscribe systems the use of the TCP protocol would considerably increase the overhead on the communication between system´s components. The UDP protocol does not provide reliable data delivery, so in order to combine performance with reliability, an implementation of a reliable UDP protocol has been adapted for the MD-ECI (an for the MoCA) called RUDP (Reliable UDP) [15, 16]. This protocol gives to each datagram packet sent by a RUDP client a sequence number, which is used by the receiver (RUDP server) to detect an eventual loss of some UDP packet. The RUDP server sends an acknowledgement (ACK) packet for each packet it receives back to the client. The client has an expiration time to wait for ACKs before a retry on the sending of the packet. This expiration time is tuned with the packet´s roundtrip time, which is measured by a timer that calculates the time a packet takes to be transmitted and to be returned to the client as an ACK. Each sent and successfully confirmed packed is used by the roundtrip timer to define the expiration time.

When a packet is sent and the client does not receive an ACK packet on time, if the maximum number of retries has not been achieved, the expiration time is exponentially increased (with a configurable exponential factor) to avoid overcharging the network with packets. If the maximum number of sending retries is achieved and no ACK packet is received, an exception is thrown to the upper software layer which is using the RUDP protocol. Specifically to this work, when the publish/subscribe system tries to send data through the network with a RUDP client and catches a sending exception, this means that the receiver of the message cannot be contacted and, thus, its Media State will be set as DISCONNECTED.

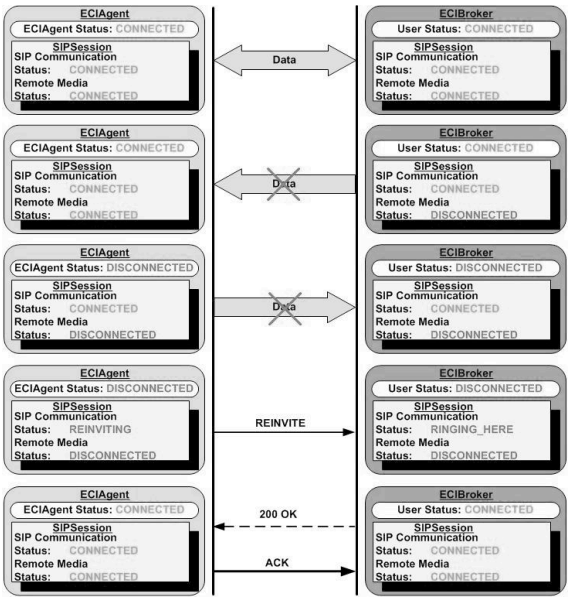

Figure 7 – Detection of disconnection by ECIAgent and ECIBroker

ECIAgent side disconnection detection and handling. As it is assumed that the event broker (ECIBroker) is always connected on a node on the fixed network, when an ECIAgent can´t send data to the ECIBroker, the ECIAgent implicitly infers that there is some problem with it own connectivity. Therefore, it is defined that the ECIAgent can detect is own disconnection when the Remote Media Status of its SIP Session with the ECIBroker changes to DISCONNECTED. The ECIAgent contains an auxiliary indicator (just for organizing purposes) which indicates its connectivity state with the ECIBroker, called ECIAgent Status. This indicator is actually always equal to the current state of the Remote Media Status indicator of the established session with the ECIBroker.

ECIBroker side disconnection detection and handling. The ECIBroker can detect the disconnection of an ECIAgent when some data can't be sent to the ECIAgent. When this happens the ECIBroker changes the Remote Media Status of the session with the correspondent ECIAgent to DISCONNECTED). The ECIBroker also contains an auxiliary state indicator for each user, which indicates the state of the ECIAgent´s connectivity with the system, called User Status. Therefore, for the ECIBroker, a session with

the Remote Media State indicator set to DISCONNECTED represents that the user (session owner) is disconnected from the system (i.e. cannot be contacted by the system) and represent this situation setting the User Status auxiliary indicator do DISCONNECTED. On the other hand, a session with the Remote Media State set as CONNECTED indicates that the user is connected and can be contacted, consequently, the User Status indicator of this user will be defined as CONNECTED.

When a session is established between an ECIAgent and the ECIBroker, the Remote Media State inside the Session is defined as CONNECTED for both sides of the communication, since it is assumed that on the conclusion of a Session negotiation both participants are ready to exchange data through the addresses specified on their Session´s SDP descriptions. This means that when a Session is updated, both participants become immediately reconnected (until some interruption occurs on the data exchange, which would define the state of the side which cannot be contacted as DISCONNECTED).

Management of subscriptions and notifications for disconnected users. The subscriptions are kept on the ECIBroker for some time even for disconnected users. They contain an attribute which specifies the limit of time they remain valid. When the specified time of a subscription expires, the subscription is removed from the ECIBroker. This verification doesn't need to be executed with a high periodicity, because the system doesn't generate notifications for expired subscriptions. If there are queued notifications for a disconnected user, correspondent to a cancelled subscription, they are not removed but are delivered normally when the user reconnects, because they have been generated when the subscription was still valid. On a mobility scenario, a device could be disconnected because it is not on the range of the wireless network or because it has been turned off. On the case of the device being turned off, an ECIAgent running on it would lose all its state (e.g. awareness of subscriptions it has on the ECIBroker), so it would be necessary some persistent storage of subscriptions on the ECIAgent. A mechanism that resolves such question is the capacity of the ECIAgent to query the ECIBroker when needed (e.g. on initialization) to get the list of existing subscriptions.

When a user is disconnected from the system and a published event matches with the user´s subscription, or when the user is still not defined as disconnected, but it is not possible to deliver the generated notification (on this case the user is defined as disconnected when the sending of the notification fails), the notification is stored to be delivered when the user reconnects. This storage is performed on queues inside the ECIBroker, indexed by the users´ URI. The ECIBroker´s configuration properties allow the definition of notifications queues maximum size.

NAT Traversal. For using the MD-ECI through NAT (Network Address Translation), the Media Proxy NAT traversal solution is used. This solution is seamless for SIP User Agents (e.g. SIPMobility) and requires no changes in their internal logic or in their configuration parameters. The re-configuration of the SIP Proxy and the Media Proxy is necessary, requiring changes on the way the requests are processed. The SIP Proxy must be programmed to interact with the Media Proxy and to forward requests to it. The Opensips SIP Proxy provides an internal programming language which was used in this work to define the request processing logic in order to use this NAT mechanism. Details about the Media Proxy solution can be found in [17].

Implementation. The MD-ECI and SIPMobility were developed in Java. The NIST´s JAIN SIP API, also developed in Java and used as base for the development of the SIPMobility API, can be used on JME (Java Mobile Edition) devices with the CDC (Connected Device Configuration) profile. So, the ECI that already could be used with the JME/CDC is still compatible with this platform. In addition, we have ported the MD-ECI and SIPMobility APIs to the Android O.S., and tested them on a G1 device.

5. Evaluation

This Section presents the evaluation of MD-ECI´s performance with different configurations. It also presents a test which shows the system´s throughput for events publications on a mobility scenario.

Overhead introduced by mobility management. To evaluate the overhead introduced by the mobility management, the use of the MDECI with mobility and disconnection support has been compared for different configurations an also with the original ECI. For this evaluation, the metric used was the difference between the time of the publication of an event and the time of arrival of that event on the event consumer. In order to calculate this difference two variables were created inside the event, containing the publication time (set by the event publisher, at the moment of the publication) and the arrival time (set by the event consumer, when the event notification is received). To measure this time with precision (without the need of clock synchronization between different devices), both the event producer and the event consumer were established on the same device (the test considered a variable number of event consumers) connected to a Wi-Fi wireless network, both using the ECIAgent API, while the ECIBroker has been established on a remote device on the fixed network.

The event publisher was programmed to send events periodically, and the event consumers were programmed to calculate the sum and average of the publication time of the events received. The sum of publication times was compared between different configurations: The MDECI with the SIP Mobility Layer and a Media Proxy to allow NAT traversal, subdividing this configuration in two cases, one with the Media Proxy on a remote server to the ECIBroker and other with the Media Proxy co-located with the ECIBroker. The MDECI with SIP Mobility, but without NAT traversal mechanism. The original ECI, without mobility management. The curves show the sum of publication times of events on the system. The events publication rate on this test was 20 events per second and a total of 2000 events have been published for each sample.

The curve representing the original ECI shows the best performance, since there is no overhead of using the RUDP protocol, which requires each packet to be confirmed by the receiver.

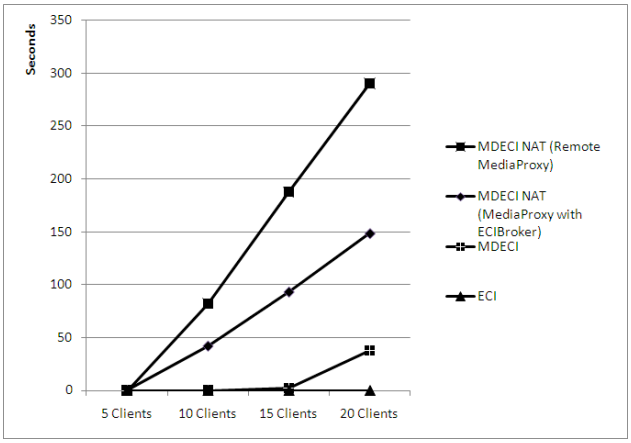

Figure 8 – Evaluation of the overhead caused by M.M.

The curve representing the MDECI with SIP Mobility (without NAT traversal), shows a small increase on the publication time, caused by the use of the RUDP protocol. The curve representing the MDECI with SIP Mobility and NAT traversal shows even higher publication times, since all events must pass through the Media Proxy, which in this case is co-located with the ECIBroker, in order to minimize the traffic of data through a third node on the network. Finally, the curve representing the MDECI with SIP Mobility, NAT traversal, and the Media Proxy remote to the ECIBroker is the one with the worse performance, since all events must pass through a third node on the network, causing an increase on the time of data transmission.

Throughput on a mobility scenario. To evaluate the system throughput on a mobility scenario, that is, the capacity to deliver published events to consumers, located on devices that change their IP addresses, the used metric was the percentage of events loss (event are lost when the queues fill up and events are discarded by the ECIBroker). This percentage was measured for different periodicities of IP address change of the mobile device end, and for different configurations on the size of the buffer used by the ECIBroker to store events that cannot be delivered to consumers. For this test, a publisher was established on a device on the fixed network, while an event consumer was established on a mobile device on the Wi-Fi network. The mobile device with the event consumer was programmed to change its IP address periodically (the test considered a variable time for this periodicity, i.e. it was configured to change the IP address each minute, each half minute and each third of a minute). The events publisher was programmed to send events periodically with a rate of 20 events per second and a total of 2000 events have been published for each sample. To count the events loss, each consumer keeps a counter of received events. The table below shows the results of the throughput test, with the percentages of events loss for the different periodicities of IP address change and for different sizes of the buffer at the ECIBroker.

Events buffer size on ECIBroker	**Events Loss (%)**		
100	**8,38%**	**13,94%**	**27,67%**
150	**4,21%**	**7,77%**	**27,55%**
200	**0,00%**	**0,00%**	**1,22%**
Periodicity of IP Address change:	**1 min**	**1/ 2 min**	**1 / 3 min**

Table 2 – Evaluation of the system throughput on a mobility scenario

The results show that the ECIBroker configuration to keep a buffer of 200 events was enough to prevent any event loss, while the device changes its IP address 2 times per minute, for a publication periodicity of 20 events per second. The main factors that contributed to the results of this test are the event sending periodicity, the time the receivers remain disconnected while they change their IP address and the size of the buffer on the ECIBroker. The time which the device takes to change its IP address (i.e. the time a consumer remains disconnected) influences directly the amount of events which need to be stored on the ECIBroker. On this test, the change of IP address was forced to happen on a determined periodicity and with a pool of known IP addresses, which the device can use (i.e. change from a known IP address to another) in a constant time. However, on a wireless network scenario on which the change of the IP address depends of an automatic configuration service (e.g. DHCP), which doesn't have a constant time for the change of address, the results above can not apply. In this case, the size of the buffer on the ECIBroker may need to be increased or the periodicity of events publication may need to be limited on the publisher.

As mentioned before, in order to validate the solution on portable mobile devices, the MDECI and the SIP Mobility have been ported to the Android O.S. [18], and

tested on a G1 device. Figure 9 shows the average event publication time for an ECIAgent running on a G1 portable mobile device and a traditional notebook device.

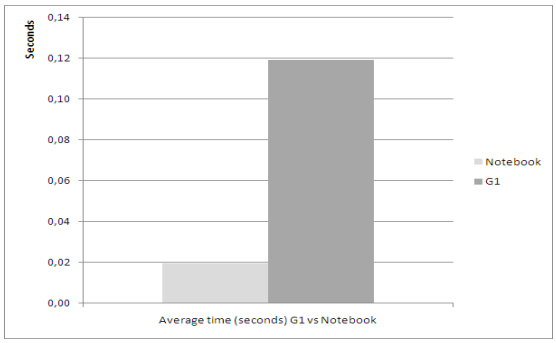

Figure 9 – Evaluation of MDECI performance with G1 device and a notebook

6. Related Work

All existing work about mobility management on the application layer refers directly do the SIP protocol and to our knowledge there is no other alternative standardized protocol on the literature to address this problem specifically on this layer. Since the publications of the original SIP mobility management solution [3, 19], many other works explored aspects like analysis on the handoff latency and optimizations for many types of scenarios [20-25]. However, only one work has been found which presented the actual design and implementation process of a SIP Mobility Layer. The work [26] presents the design and development of a SIP mobility layer very similar to our work. It contains an API which is a SIP User Agent called NCTU SIP UA (National Chiao Tung University SIP User Agent). In addition to the description of the design and the main components of the NCTU SIP UA's architecture, a study of the handoff delay between different sub-networks using IPv4 and IPv6 is presented. Another difference is the focus of the work on the transmission of voice streams using the RTP protocol, while on our work the focus is the use of the implemented layer for supporting mobility and disconnections on publish/subscribe systems, together with a reliable transmission protocol such as the RUDP (Reliable UDP). In addition, the technologies and design of the architecture also presents differences from our work, including the programming language. The NCTU SIP UA API was developed with the C programming language and with auxiliar APIs, e.g. the eXosip (The eXtended osip library) which has similar functionality to the JAIN SIP API, also developed on this programming language. On the other hand, on our work, the SIP Mobility Layer has been implemented with the Java programming language and uses auxiliary APIs (e.g. JAIN SIP API) also developed with this language. The use of the Java language provides more portability, including for mobile devices.

Concerning mobility support for publish/subscribe systems, many other works, e.g. [8, 27-31] address the adaptation of existing pub/sub systems (e.g. adaptations of the Elvin, JEDI, REBECA, Siena, among others) to support mobility. The majority of such systems has a distributed architecture. A pub/sub system with a distributed architecture has the advantage of being scalable for a large number of data publishers and consumers. With the given importance on the distributed aspect of these systems, one can observe on these works that the focus is on the mobility of devices between different event servers (i.e. a device disconnects from a broker and connects to another broker). That is, they propose distributed algorithms which the Brokers must execute in order to synchronize subscriptions, handle the correct routing and delivery of event notifications, among other operations to be executed when a device moves from one broker to another. Therefore, the main difference to our work is the approach to the concept of mobility. In our work, the mobility of a device is considered as the change of its IP address, while it moves between different heterogeneous networks remaining connected to the same event broker.

7. Conclusions and Future Work

This work presented the implementation of a SIP Mobility Layer and its application on the adaptation of a publish/subscribe system to support the mobility and the temporary disconnection of devices.

The mobility layer was implemented as a Java API, reusable by any application which requires a solution like this. Also, we investigated a way to allow the extended pub/sub system, together with the mobility layer, to work even when the system components are executing on different sub-networks protected by firewalls of NATs, using a Media Proxy with the SIP protocol to address this problem.

We believe that the extension of the ECI with the SIP Mobility Layer was an adequate choice, because the ECI is offered to users as Java APIs, and the ease of deployment, flexibility and simplicity of the SIP protocol has allowed these APIs to remain independent of any change on the underlying infra-structure. The elements of the SIP network (e.g. Proxy SIP, Registrar, etc) can be provided as services for the MDECI's users, so they can only download the APIs and use an existing SIP network. In addition, the programmatic interface of the original ECI could remain unchanged, unless by some configuration parameters which can be added through configuration files or passed as arguments.

Therefore, the easy deployment of the SIP protocol allows the MDECI to provide mobility and disconnection to devices and to be used by any interested user in a practical and effective manner, offering the capacity of total mobility between sub-networks protected by NATs.

Regarding performance, it is known that the SIP protocol can be less efficient than Mobile IP, since Mobile IP executes at the network layer of the protocols stack. However, the flexibility and real viability of large scale deployment compensate the eventual loss of performance. Although the use of NAT traversal mechanisms had affected the MDECI´s performance, such degradation was not due to the mobility management itself, but was due to the routing of data through a third network node (Media Proxy).

The choice of RUDP as the protocol for notifications delivery also has a small part on the performance degradation considering the original ECI. The SIP by itself didn't affected the system performance, because after the negotiation of a session, the communication is performed directly between the participants, in this case the ECIAgent and the ECIBroker.

On the present work, the ECI´s architecture has been limited to a single event broker, but as a future work it can be extended to support an interconnected network of Brokers, allowing its use by a much larger number of users, distributed through many geographical locations, served by different event brokers. Also, there should be incorporated algorithms for the synchronization of subscriptions and notifications of clients connected to the network of Brokers.

8. References

[1] T. Henderson and B. Works, "Host mobility for IP networks: a comparison," *Network, IEEE*, vol. 17, no. 6, 2003, pp. 18-26.

[2] J. Rosenberg, et al., "RFC3261: SIP: Session Initiation Protocol," *RFC Editor United States*, 2002.

[3] E. Wedlund and H. Schulzrinne, "Mobility support using SIP," ACM New York, NY, USA, 1999, pp. 76-82.

[4] G. Baptista, et al., "Uma API Pub/Sub para Aplicaçoes Moveis Sensiveis ao Contexto," *Book Uma API Pub/Sub para Aplicaçoes Moveis Sensiveis ao Contexto*, Series Uma API Pub/Sub para Aplicaçoes Moveis Sensiveis ao Contexto, ed., Editor ed.^eds., pp.

[5] J. VITERBO, et al., "A Middleware Architecture for Context-Aware and Location-Based Mobile Applications," *Book A Middleware Architecture for Context-Aware and Location-Based Mobile Applications*, Series A Middleware Architecture for Context-Aware and Location-Based Mobile Applications, ed., Editor ed.^eds., 2008, pp.

[6] V. Sacramento, et al., "MoCA: A Middleware for Developing Collaborative Applications for Mobile Users," *Distributed Systems Online, IEEE*, vol. 5, no. 10, 2004, pp. 2-2.

[7] G. Cugola and H. Jacobsen, "Using publish/subscribe middleware for mobile systems," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 6, no. 4, 2002, pp. 25-33.

[8] Y. Huang and H. Garcia-Molina, "Publish/Subscribe in a Mobile Environment," *Wireless Networks*, vol. 10, no. 6, 2004, pp. 643-652.

[9] P. O'Doherty and M. Ranganathan, "JAIN SIP Tutorial," *At http://java.sun.com/products/jain/JAIN-SIP-Tutorial.pdf*, 2003.

[10] W. Eddy, "At what layer does mobility belong?," *Communications Magazine, IEEE*, vol. 42, no. 10, 2004, pp. 155-159.

[11] L. Popescu, "Supporting Multimedia Session Mobility using SIP," 2003, pp. 15-16.

[12] M. Handley and V. Jacobson, *C. Perkins," SDP: Session Description Protocol*, RFC 4566, July 2006, 2006.

[13] E. Gamma, et al., *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Reading, MA, 1995.

[14] "Java API for SIP Signalling," https://jain-sip.dev.java.net/.

[15] E. Pitt, *Fundamental Networking in Java*, Springer, 2006.

[16] W. Stevens and T. Narten, "Unix network programming," *ACM SIGCOMM Computer Communication Review*, vol. 20, no. 2, 1990, pp. 8-9.

[17] A. Georgescu, "Best practices for SIP NAT traversal," 2004.

[18] Google, "Google Android," code.google.com/android.

[19] H. Schulzrinne and E. Wedlund, "Application-layer mobility using SIP," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 4, no. 3, 2000, pp. 47-57.

[20] N. Banerjee, et al., "Analysis of SIP-based mobility management in 4G wireless networks," *Computer communications*, vol. 27, no. 8, 2004, pp. 697-707.

[21] N. Nakajima, et al., "Handoff delay analysis and measurement for SIP based mobility in IPv6," 2003.

[22] W. Kim, et al., "Link layer sssisted mobility support using SIP for real-time multimedia communications," ACM New York, NY, USA, 2004, pp. 127-129.

[23] E. Ivov and T. Noel, "Optimizing SIP application layer mobility over IPv6 using layer 2 triggers," 2004.

[24] N. Banerjee, et al., "Seamless SIP-based mobility for multimedia applications," *IEEE Network*, vol. 20, no. 2, 2006, pp. 6-13.

[25] N. Banerjee, et al., "SIP-based mobility architecture for next generation wireless networks," 2005, pp. 181-190.

[26] C. Yeh, et al., "SIP Terminal Mobility for both IPv4 and IPv6," 2006, pp. 53-53.

[27] M. Caporuscio, et al., "Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 2003, pp. 1059-1071.

[28] G. Mühl, et al., "Disseminating Information to Mobile Clients Using Publish-Subscribe," *IEEE INTERNET COMPUTING*, 2004, pp. 46-53.

[29] G. Cugola, et al., "The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 2001, pp. 827-850.

[30] A. Zeidler and L. Fiege, "Mobility support with REBECA," 2003, pp. 354-360.

[31] L. Fiege, et al., "Supporting Mobility in Content-Based Publish/Subscribe Middleware," *LECTURE NOTES IN COMPUTER SCIENCE*, 2003, pp. 103-122.