An Agent-Based Architecture for Context-Aware Communication

Romelia Plesa[1], Luigi Logrippo[2,1]
[1]*School of Information Technology and Engineering, University of Ottawa, Canada*
[2]*Département d'informatique et ingénierie, Université du Québec en Outaouais, Canada*
rplesa@site.uottawa.ca, luigi.logrippo@uqo.ca

Abstract

We propose an agent-based architecture that allows context-aware communication between users. In seeking a model that is suitable for the design of the required functionalities of our framework, we identify significant overlap between the concepts needed in our domain and those used in the Belief-Desire-Intention (BDI) models of agency. We present a comprehensive attempt to use the BDI model for describing the architecture and protocols of our context-aware communication system, which includes a mechanism for handling conflicting user policies.

1. Introduction

Given the current trends in personal communication, there is a growing need of services tailored to a user's specific needs and preferences. With devices that enable mobile communication becoming more popular, there is an increasing necessity for the users to control and customize their communication, depending on the context in which the communication takes place and influencing the user's availability and reachability for communication. The literature distinguishes between context-free and context-aware communication systems [1]. *Context-aware* services take advantage of knowledge of real-time context regarding the purpose or the circumstances of a call that one is receiving or initiating, thus allowing the user to manage his communication in terms of policies. However, most of today's telephony communication services are *context free*, they do not have such knowledge.

This paper presents the progress to date in an attempt to implement a Belief-Desire-Intention (BDI) agent architecture for communications systems, using AgentSpeak(L), an agent-oriented programming language that implements the BDI architecture. We investigate how the BDI approach may be used to create an agent-based architecture, providing useful, intelligent services to users in realistic settings. Our work tackles different challenges and issues related to the system architecture, design and programming. The elements of the architecture were presented in detail in [2]. Our architecture is applicable to scenarios where users define complex policies on how their communication should be handled, based on their current context, thus offering a very high degree of customization.

2. BDI and AgentSpeak(L)

One of the most successful theoretical models of rational agents is the Belief-Desire-Intention model [3]. BDI agents have proven useful in the theoretical study of rational agents [4] and in practical applications [5]. The BDI framework [6], implemented in several software platforms (e.g. Jack Intelligent Agents [5]) has a wide appreciation, witnessed by the development of a BDI logic [7], the definition of BDI-based languages (AgentTalk [8], AgentSpeak(L) [9]) and the creation of BDI-based development tools such as PRS [10] and dMARS [11]. Providing a convenient terminology for describing intelligent agents, the BDI framework has had many practical applications, making the theory clearer and more generic. BDI agents have been applied to highly dynamic and unpredictable situations. They remain responsive to changes in system state, by only partially expanding alternative plans of actions and also provide ways to recover from failed actions. They describe how to handle conflicting goals, and how to modify already executing actions, all of which are needed in dynamic simulations.

Even though there is a body of research on implementing agents on communication devices [12], [13], no attempt, to our knowledge, has been made at using the BDI model to implement a communication system architecture.

The AgentSpeak(L) programming language was introduced in [9] and it provides an abstract framework for programming BDI agents. The beliefs, desires, and intentions of the agent are not explicitly represented. Instead, they are ascribed to agents written in AgentSpeak(L). An AgentSpeak(L) agent consists of a set of beliefs and a set of plans. There are two types of goals: achievement goals (prefixed with *!* - indicate that the agent wants to achieve a state where the predicate is true) and test goals (prefixed with *?* - test if the agent is in a certain state). Plans refer to the basic actions that an agent can perform. A plan p is written as: p ::= te : ct <- h, where te is the triggering event, followed by a conjunction ct of belief literals representing a context. The remainder h of the plan is a sequence of actions or (sub)goals that the agent has to achieve. A triggering event initiates the execution of a plan and is related to the addition ('+') and deletion ('-') of beliefs or goals. The AgentSpeak(L) interpreter also manages a set of intentions (particular courses of actions to which an agent has committed) and its functioning requires three selection functions: SE selects an event from the set of events; SO selects an applicable plan from a set; SI selects one particular intention from the set of intentions.

The agents test if the events activate any new plans. One plan becomes intended in a single reasoning cycle, and only one intended plan can execute. This can generate new beliefs, goals, or basic action execution. New beliefs and goals serve as triggering events for plans, actions produce changes in the environment.

3. Architecture

In order to provide complex services that are tailored to users' specific desires and preferences the system needs the following functional requirements:

- Collection/dissemination of context information, publishing of user and devices presence information
- description of user policies and preferences
- ubiquitous handling of communication

Fig. 1 presents the overall system architecture.

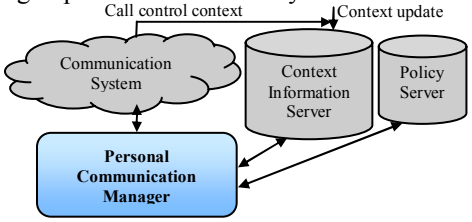

Figure 1. The overall architecture

The communication part is a complex system by itself. One of our goals is to make our solution independent of the underlying communication protocol (SIP [14], H.323 [15]). The requirement we impose on the architecture is that every message that arrives for or is sent by a user must be intercepted in order to extract important information contained inside the message, (i.e. the identity of the caller).

The *Context Information Server* controls the context updates, stores and distributes the context information. The dynamic nature of context information requires a mechanism for keeping up-to-date information in the server, in order to allow services to adapt to the changing context.

The *Policy Server* manages the user's personal policies, including creating, storing, deleting, retrieving and fetching policies. A number of languages exist for specifying policies. The Call Processing Language [16] allows users to define how their calls are handled, but has limited expressiveness for call control [1]. LESS [17] inherits the basic structure from CPL enhancing it with more elements, thus allowing users to program their own communication services. A policy language for policies in the communication domain has been defined in [18]. In Section 5, we will present our approach to represent policies.

A new functional entity is introduced in the architecture: a *Personal Communication Manager (PCM)*, which is a software agent that represents each user and is responsible for deciding the flow of actions for the call, based on personal policies and on information about presence and current context. PCM treats relevant events that occur in the system (invitations to a call, updates in presence or changes in context). PCM is the entity that receives request messages (such as INVITE messages in a SIP-based architecture) and decides the actions that should be taken and how the call should be handled. The components of the PCM are shown in Fig. 2.

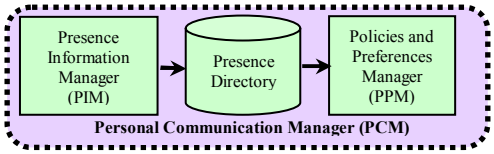

Figure 2. Personal Communication Manager

Presence Information Manager aggregates presence and context information from different sources, manages raw presence data in order to build the "consolidated presence information" [2] for the user, which represents a unified view of an individual's current status. This is achieved by using a rule-based process that takes into account presence and context indicators and their ability to reflect the user's state. Any change in an entity's presence causes the PIM to re-apply the rules and rebuild the user's presence.

The *Presence Directory* is a repository in which all known and deduced presence information is deposited and can be retrieved. The *Policies and Preferences Manager (PPM)* contains the preference logic and rule

based processes that respond to requests to contact an entity. The presence data is interpreted to establish the best method for contacting the user at a particular moment.

4. Context-Aware Call Handling

The call model that we propose will include context update, service selection based on context information and user personal policies as well as service execution. Context update is a process that is done continuously. The feature selection and execution mechanisms will be incorporated into the Personal Communication Manager, more precisely in the Policies and Preferences Manager (PPM) component of it.

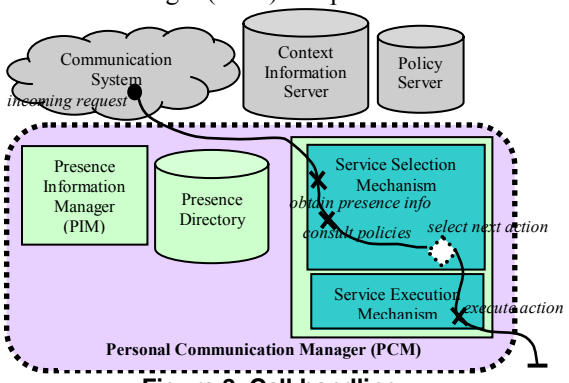

Figure 3. Call handling

PPM, which contains the preference logic and rule-based processes to respond to different requests, consults the user's policies and the information stored in the Presence Directory and decides about the handling and execution of the call or any other request that arrives. From the various options and alternatives available to it at a certain moment in time, PPM needs to select the appropriate actions or procedures to execute. We call this the selection function or the Service Selection Mechanism. In Fig. 3, we show the actions of the PPM using the Use Case Maps notation [19]. We use a dynamic stub to represent the complex mechanism of service selection. After the selection is done, the actual execution of the action is done by the Service Execution Mechanism of the PPM.

5. BDI and Context Aware Communication

5.1. BDI Mapping

By looking at the BDI model as well as at the PPM component of our Personal Communication Manager and the way we want it to function, we found significant overlap between the requirements of our domain and the concepts discussed in the belief-desire-intention model of agency. We can consider that the PPM component that we propose as a part of the system's architecture is a BDI agent (Fig. 4).

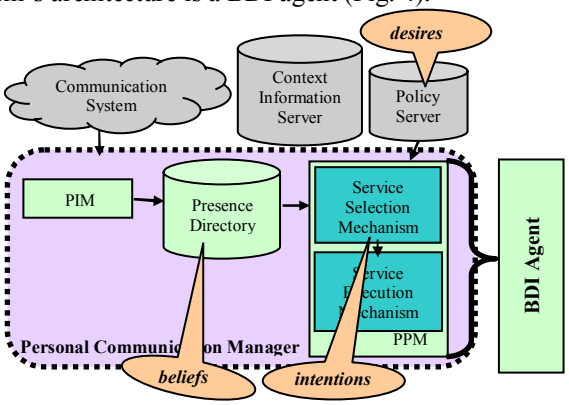

Figure 4. BDI Mapping

The consolidated presence information, stored in the Presence Directory and representing the characteristics of the environment as perceived by the agent, represents the ***beliefs,*** updated appropriately after each sensing action or change in a user's status. The policies, stored on the Policy Server and seen as objectives to be accomplished, can be considered agent's ***desires.*** The output of the selection function, which decides, on the basis of the policies, what action should be executed next, will be the agent's ***intentions***.

BDI agents fulfill the requirements needed for our framework, providing a way to interleave deliberation with responsiveness and limit the amount of forward deliberation required to act rationally. BDI agents can partially search and expand planned actions allowing them to select good alternatives, while avoiding constant deliberation and its associated time penalty.

5.2. Example

In order to further justify the reasoning for the adoption of an agent-based approach to support context aware services, we consider the following situation:

Dr Smith needs to achieve the following tasks on a usual day:

1. *Arrive to the hospital*
2. *Log into the hospital's system to advertise his presence*
3. *Get the schedule for his consultations and surgeries*
4. *Perform his activities, according to his schedule*
5. *Time permitting, assist in any emergency situation*

Within an agent context, these tasks are represented as goals that need to be fulfilled in a given sequence. For each goal, a specific sequence of actions must be executed. There may be a number of plans for achieving the same goal. As a result, the agent has to perform *plan selection*, based on some criteria (for example, to achieve the second goal, the doctor can log in from the computer in his office, or he can use his PDA). Assuming the doctor logs in from his computer, the corresponding plan becomes an intention that might require the execution of other sub-tasks, for which,

again, there might be a number of plans to achieve them. The process can continue, attempting to achieve all goals by executing the appropriate plans, which may trigger other sub-goals, and so on.

While executing the plans, a number of problems may arise so the agent needs to perform *plan failure recovery* (for example, if Dr. Smith can't log from his computer, his agent should find alternative plans, e.g. using another computer, considering the new context).

Another problem the doctor might face is *conflict between different goals*. For example, if the doctor wishes to play golf, this may conflict with his plan to attend a presentation. The agent should be able to *resolve* this conflict by arranging a different time to play golf, or it might have to perform *goal selection*, by choosing the goal that is more important.

It can be seen from the above scenario that as the number of goals and alternative plans increases, the complexity of the reasoning that the agent needs to perform increases significantly. This creates the opportunity for providing automated support for the user, but particular challenges arise due to the dynamic nature of the environment. Our proposed solution consists of agents representing each user, which are capable to perform tasks as the ones described above. They have access to updated information about the context of their users. With a library of plans, they are capable to decide what actions to execute. Some essential features that an agent will provide are:

- **Context-awareness.** The choice of plans must take into account the current context of the user and decisions are made based on the most recent information about the environment.
- **Plan selection.** The agent is able to make a choice between different plans, in case there are a number of alternative plans for achieving the same goal. Appropriate decision procedures must therefore be supplied for supporting plan selection.
- **Plan failure recovery.** If a plan fails, the agent is able to retract properly and select an alternative plan.
- **Conflict resolution.** When the user has a number of goals that cannot be achieved simultaneously, the agent must be able to make a decision about which goals to try to achieve.

5.3. Proof of Concept

Given the mapping we have presented between the BDI model and the functionalities of our architecture, we have implemented the Personal Communication Managers as a set of BDI agents. In particular, the agents are programmed in AgentSpeak(L). The major advantage of this approach is that the BDI mechanism incorporates both the selection mechanism and the execution mechanism that are required by the functionality of the PPM (Fig. 4). The plans that drive the system's behavior are expressed in terms of user context, which makes the definition of new plans more adaptive and flexible.

To illustrate the essential features provided by the system and show how the BDI model can provide the required functionality, we implemented a pilot demonstration of the framework as the first step in implementing a fully functional simulation model based on a real communication system enhanced with context services. The demonstration consists of six agents that share a set of beliefs, stored in a relational database (the Context Information Server in the architecture). Plans, representing user's policies, are defined for each agent. The external events to which an agent must respond are request messages that come from or are sent to the Communication System - Fig. 1 (e.g. INVITE requests on a SIP based architecture). We simulate these requests by feeding the agents with events that correspond to real requests.

We aim to demonstrate that agents will respond to contextual information that comes from the system and will behave accordingly, taking into consideration the user policies, defined as plans. We also want the agents to respond to system changes brought about by other agents and to show that communicating agents can negotiate in order to avoid possible conflicts.

To avoid discrepancies between the agents' understanding of the system and the actions they can perform on it, we need the assumption of a shared data model among all agents. We defined an *Entity-Relationship data model* that provides the means to hold the context, as well as the semantics to describe the simulation domain. With this, the agent's beliefs will be represented as facts against the data model. The agent's desires, in the form of plans, are formulated in terms of entities, attributes and relationships contained in the data model. The advantage of using this data model approach is that each entity class in the data model can be viewed as a finite domain, with the object instances as the elements in that domain. The object's attributes can be used for specifying constraints and reasoning in terms of the data model.

Each agent has a set of plans that models the user's policies. We will describe in detail the plans for handling a specific situation:

> *Dr. Smith is in his office. A colleague calls him from within the hospital. The system will decide, based on the doctor's location, that he is available on several devices: his PDA, his cell phone or his desktop phone. Since the caller is using a phone, and the PDA does not have voice capabilities, the choice is narrowed to two devices. Dr. Smith has a policy that designates his cell phone for calls from his family. Therefore, the system decides to transfer the call to Dr. Smith's desktop phone.*

Before we start discussing the plans to realize this policy, it is worth discussing the initial belief base that is required in the running version of the program. The

beliefs of the agent are based on the data model that we have defined, so it will contain information about the users and the devices available to users, the locations and the activities that users are involved in. Relationships between entities are modeled using databases that contain references to the entities involved in the relationships. (For example, PERSON_PERSON specifies the relationships between two persons identified by their unique ID number). All this information is available in the Presence Directory component of the architecture. From the agent point of view, the information in the database is mapped into predicates. For example:

```
PERSON(john, u001, TRUE, av, call, 562-5800, eng, busy, j, doctor)
DEVICE(fix_phone, dev001, ip, bell, 5010, eng, call, open, 563-2345)
PERSON_PERSON(PERSONID_1,PERSONID_2,RELATIONSHIP)
```

Agents are able to consult, insert or delete values from this database by simply adding, deleting or querying the facts as beliefs.

The policy is enforced using a plan for the situation in which an incoming call arrives, from X, for Dr. Smith. The triggering event for the plan is incoming_call, with a parameter specifying the caller.

```
+incoming_call(X):true <- !get_devices(DeviceList);
  !get_relationship(X,Relationship);
  !get_location(Location);!get_activity(Activity);
  !process_call(X, DeviceList, Relationship, Location, Activity).
```

First, a list of devices where the doctor can be reached is obtained, by adding the subgoal get_devices.

```
+!get_devices(DeviceList) : true
<- .findall(X, device(X,Y,Z,_,_,_,"call",_,_), DeviceList);
   ?name(N);?person(N,ID,_,_,_,_,_,_,_,_);
   !get_user_devices(DeviceList,ID,UserDeviceList).
+!get_user_devices(DeviceList,ID,UserDeviceList) <-
     !get_u_devices(DeviceList,ID,[],UserDeviceList).
+!get_u_devices([],ID,L,L).
+!get_u_devices([D|T],ID,L0,L) : device(D,Did,_,_,_,_,_,_,_,_) &
     person_device(ID,Did)<-!get_u_devices(T,ID,[D|L0],L).
+!get_u_devices([D|T],ID,L0,L)<- !get_u_devices(T,ID,L0,L).
```

A list of all devices having the capability "call" is obtained by querying the device table. After that, the subgoal get_user_devices determines which devices from this list are associated with the user.

Next the relationship between the caller and the user must be determined, by verifying if there is information in the PERSON_PERSON table that contains both the ID of the user and of the caller. If there is no such information, the relationship is set to unknown.

```
+!get_relationship(X,R): name(N)&person(N,ID,_,_,_,_,_,_,_,_)
& person(X,Xid,_,_,_,_,_,_,_,_) & person_person(ID,Xid,R)<-
true.
+!get_relationship(X,R) <- R = "unknown".
```

The current location of the user and the activity also need to be determined.

```
+!get_location(LName): name(N)&person(N,ID,_,_,_,_,_,_,_,_)
   & person_location(ID,L) & location(L,LName,_,_,_,_) <- true.
+!get_location(L) <- L = "unknown".
+!get_activity(AName) : name(N)&person(N,ID,_,_,_,_,_,_,_,_)
   & person_activity(ID,A) & activity(A,AName,_,_) <- true.
+!get_activity(A) <- A = "unknown".
```

Having all the information, the actual routing of the call, is done by the plan triggered by the addition of the subgoal process_call. Certain conditions are verified and the action to be executed depends on this conditions. There is also a default plan, applicable when none of the conditions are true.

```
+!process_call(X, DeviceList, Relationship, Location, Activity) :
   Location=="office"&Relationship=="family"<- ring_mobile.
+!process_call(X, DeviceList, Relationship, Location, Activity) :
   Location=="office"&Relationship=="colleague"<- ring_fixed.
+!process_call(X, DeviceList, Relationship, Location, Activity) :
true  <- ring_mobile.
```

In a similar fashion, a set of plans is defined for each agent in order to cover all the policies for the user it represents.

AgentSpeak(L) includes a mechanism that allows agents to communicate, thus sharing plans and consulting about the content of their beliefs base. We use the power of this mechanism to handle conflicting policies. To illustrate, we use an example with two features, Originating Call Screening (OCS) and Call Forwarding (CF). OCS forbids calling numbers on a screening list. CF forwards incoming calls to another number. A conflict occurs if a user *A*, whose OCS list includes user *X*, calls user *B,* who forwards calls to *X* through CF, thus overruling A's policy. To illustrate how this conflict is solved in our simulator, we assume that Bob has OCS and forbids any calls to Charles and Alice has CF and forwards her call to Charles. Thus, agent bob will have in its beliefs set a line saying:

```
ocs(charles).
```

Similarly, agent alice will have a belief:

```
call_forward(charles).
```

When Bob tries to dial, the event that is generated (dial(X), where X is the person he wants to reach) will trigger the execution of a plan. It is checked if X is on the screening list and only if not the call is completed.

```
+dial(X) : ocs(X) <- .print("You are forbidden to call ", X).
+dial(X): true <- .send(X,tell,incoming_call(bob)).
```

At the other end, when Alice receives an incoming call, her agent checks if the call should be forwarded. If this is affirmative, it will ask the originator of the call if this can be done (it actually asks if the person where the call is about to be forwarded is on the originator's screening list) and upon confirmation, the call will get through.

```
+incoming_call(X) : true <-  ?call_forward(F);
   .send(X,askIf,ocs(F),Answer); !ans(Answer,X,F).
+!ans(true,X,Y,F) : true<-.print ("no connection ", X, " to ", F).
+!ans(false,X,Y,F) : true <-.print("forwarding call from " , X, "
to", F);.send(F,tell,incoming_call(X)).
```

This simple example shows how the mechanism for agent communication can be used in negotiation of

preferences for users and allows for resolution of conflicts. The introduction of context allows for much richer policies that may handle calls depending on presence, availability, role, call type or call content. Conflicts can occur also between these policies and the built-in mechanism for agent communication in AgentSpeak(L) allows for easy resolution. We have applied the approach to other features and combinations of features not described here.

6. Conclusions and Related Work

Several projects have addressed the increasing interest of users to customize their services. The Universal Inbox project [20] defines an architecture for building personal and service mobility features. The Mobile People Architecture [21] is based on a Personal Proxy for achieving person-level routing. Mercury [22] is a system that supports unified communication, allowing a person to initiate a conversation using any available device. The system routes the call based on which device the other party prefers to use in a given context. While Mercury uses SIP as the underlying mechanism for managing sessions, our goal was to make our architecture protocol independent. The functional entities that we established in our design for context acquisition, storage and deployment, for policies storage and for communication handling, have attributions that are independent of the underlying communication. One of the contributions of our work is that it provides an enhanced definition of the term "presence" [2]. Consolidating all the available information for users and their environment provides a unified view of the status of the user at a given time. Our architecture allows real-time use of this information, offering the possibility for a large spectrum of services.

Although our motivation is similar to that of other researchers, as we aim to propose solutions for providing enhanced services for users, we have focused on the use of AgentSpeak(L), following important developments in the area of agent-oriented programming. Communication and telephony are areas where, to the best of our knowledge, the BDI architecture has not been used, yet their complexity require the sophisticated reasoning that AgentSpeak(L) agents display. As opposed to other BDI models, AgentSpeak(L) has an exact notation and a precise logical semantics, which resulted in the successful implementation of its abstract interpreter. It provides an elegant specification of the BDI agents and allows the agents to search and expand planned actions in order to select good alternatives. AgentSpeak(L) is an excellent candidate for the high level design of new context-aware communication systems.

Acknowledgment. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Turner, K., Magill, E., Marples, D. – *Service Provision. Technologies for next generation communications*, Wiley Series in Communications Networking & Distributed Systems.

[2] Plesa, R., Logrippo, L. – Enhanced communication services through context integration, short paper, *MATA* 2005, Montreal

[3] Rao A. S., Georgeff, M. P. - BDI-agents: from theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, San Francisco, USA, 1995.

[4] Wooldridge, M. J. - Reasoning about Rational Agents. MIT Press, 2000

[5] Howden, N., Ronnquist, R., Hodgson, R., Lucas, A. - JACK Intelligent Agents: Summary of an Agent Infrastructure. In *Proceedings of the 5th International Conference on Autonomous Agents*, 2001.

[6] Georgeff, M., Lansky, A. - Procedural Knowledge. *Proceedings of the IEEE Vol 74 (10)*

[7] Rao, A.S., Georgeff, M.: Decision procedures for BDI logics. *Journal of Logic and Computation* ***8*** (1998) 293 – 342

[8] Winikoff, M. - AgentTalk Home Page. http://goanna.cs.rmit.edu.au/ ~winikoff/agenttalk (2001)

[9] Rao, A.S. - AgentSpeak(L): BDI agents speak out in a logical computable language. In de Velde, W.V., Perram, J.W., eds.: *Agents Breaking Away*. Springer Verlag (1996) 42–55 LNAI 1038.

[10] Myers, K.L. - User guide for the procedural reasoning system. Technical report, Artificial Intelligence Center, Menlo Park, 1997.

[11] d'Inverno, M., Kinny, D., Luck, M., Wooldridge, M. - A formal specification of dMARS. In *Proc. of the 4th International ATAL Workshop*, Springer Verlag (1997) 155–176 LNAI 1365

[12] Caire, G., Lhuillier, N., Rimassa, G. - A communication protocol for agents on handheld devices. In *Proceedings of the Workshop on Ubiquitous Agents on Embedded, Wearable and Mobile Devices*, 2002.

[13] Maamar, Z., Mansoor, W., Mahmoud, Q. H. - Software agents to support mobile services. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 666–667, 2002.

[14] Rosenberg, J., Schulzrinne H. - SIP – The Session Initiation Protocol, RFC 2543, 1999

[15] H.323 Site-http://www.packetizer.com/ voip/h323/

[16] Lennox, J., Schulzrinne, H. - Call Processing Language Framework and Requirements, Int. Draft CPL-Framework-02.

[17] Wu, X., Schulzrinne, H., - LESS: Language for End System Services in Internet Telephony, Internet Draft, 2005

[18] Reiff-Marganiec, S., Turner, K. J. - APPEL: The ACCENT project policy environment/language, Technical Report CSM-161, University of Stirling, UK, 2004

[19] Use Case Mps, www.usecasemaps.org

[20] Raman, B., Katz, R., Joseph, A. - Universal Inbox: Providing Extensible Personal Mobility and Service Mobility, in *Proc.of the 3th Workshop on Mobile Computing Systems and Applications*, 2000.

[21] Roussopoulos, M. - Personal-level Routing in the Mobile People Architecture, in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1999.

[22] Lei, H., Ranganathan, A. - Context-Aware Unified Communication, *2004 IEEE International Conference on Mobile Data Management*