CENTRO DE INVESTIGACIÓN CIENTÍFICA Y DE EDUCACIÓN SUPERIOR DE ENSENADA

PROGRAMA DE POSGRADO EN CIENCIAS
EN CIENCIAS DE LA COMPUTACIÓN

"AGENTES AUTÓNOMOS EN AMBIENTES DE CÓMPUTO COLABORATIVOS UBICUOS"

TESIS

que para cubrir parcialmente los requisitos necesarios para obtener el grado de
DOCTOR EN CIENCIAS

Presenta:

MARCELA DEYANIRA RODRIGUEZ URREA

Ensenada, Baja California, México, Agosto del 2005.

RESUMEN de la tesis de **Marcela Deyanira Rodríguez Urrea**, presentada como requisito parcial para la obtención del grado de DOCTOR EN CIENCIAS en CIENCIAS DE LA COMPUTACIÓN. Ensenada, Baja California. Agosto del 2005.

AGENTES AUTONÓMOS EN AMBIENTES DE CÓMPUTO COLABORATIVOS UBICUOS

Resumen aprobado por:

Dr. Jesús Favela Vara
Thesis Advisor

La idea del cómputo ubicuo propone un ambiente físico dotado de dispositivos de diferentes escalas, con capacidades computacionales y de comunicaciones, los cuales se integran de forma natural a nuestras actividades diarias. Es decir, los dispositivos computacionales se convierten en una herramienta de trabajo que solo requiere de nuestra atención periférica. La interacción humano-computadora sucede en forma natural e implícita, lo que permite que los usuarios no requieran estar concientes de la existencia de los dispositivos. El proporcionar las anteriores características a los sistemas de cómputo ubicuo, conlleva a los desarrolladores a abordar varias complejidades de estos sistemas y a enfrentar importantes retos.

Esta tesis describe un middleware que facilita a los desarrolladores manejar algunas de las complejidades asociadas con el desarrollo de los sistemas de cómputo ubicuo por medio del uso de agentes autónomos, los cuales fueron utilizados para permitir que la tecnología de cómputo ubicuo responda a las condiciones y demandas particulares de los usuarios. Los agentes autónomos se utilizaron para implementar las características deseables de estos sistemas de cómputo ubicuo y para mejorar la interacción de los usuarios con el ambiente. Los agentes autónomos fueron identificados como los componentes de software que representan usuarios, dispositivos y servicios y que tienen capacidades para tomar sus propias decisiones acerca de que actividades hacer, cuando hacerlas, que tipo de información debe ser comunicada y a quien y como asimilar la información recibida.

El middleware de agentes (SALSA) fue creado para facilitar la implementación y la evolución progresiva de este tipo de sistemas inteligentes. SALSA principalmente proporciona: una plataforma de comunicación entre agentes que consiste de un Broker que actúa como canal de comunicación entre agentes y usuarios, y un protocolo de comunicación que permite a los agentes intercambiar diferentes tipos de objetos entre agentes, (tal como información de contexto) entre agentes y usuarios (tal como eventos generados por las acciones de los usuarios) y entre agentes y servicios (p.ej. estado de los servicios). SALSA también proporciona un conjunto de clases que facilita la implementación de los componentes de percepción, razonamiento y acción de los agentes. Y finalmente, el middleware SALSA provee de un Directorio de Agentes para que los agentes registren sus servicios y realicen búsquedas de otros agentes.

Las contribuciones de esta tesis se enfocan en presentar los requerimientos funcionales de los agentes autónomos para implementar sistemas de cómputo ubicuo, y el diseño e implementación del middleware SALSA creado para facilitar la implementación y evolución de sistemas de cómputo ubicuo. Finalmente, se proporciona evidencia de la flexibilidad de SALSA para desarrollar progresivamente sistemas de cómputo ubicuo, para lo cual se presenta el diseño e implementación de un sistema de información consciente de contexto para hospitales.

Palabras clave: cómputo ubicuo, agentes autónomos, middleware

ABSTRACT of the thesis presented by **Marcela Deyanira Rodriguez Urrea** as a partial requirement to obtain the DOCTOR of SCIENCE degree in COMPUTER SCIENCE. Ensenada, Baja California, Mexico, August 2005.

AUTONOMOUS AGENTS IN COLLABORATIVE UBIQUITOUS COMPUTING ENVIRONMENTS

Approved by:

Dr. Jesús Favela Vara
Thesis Advisor

The idea of ubiquitous computing (ubicomp) is an environment dominated by computing and communication devices of different scales which are seamlessly integrated to the users activities. Thus, under the ubicomp vision, computing devices become a work tool that demands peripheral attention of users, and the human-computer interaction tends to be more natural and implicit. These features of ubiquitous computing environments require developers to face important challenges in dealing with the complexities associated to the development of ubiquitous computing systems.

This thesis describes a middleware to facilitate developers to manage some of the complexities associated with the development of ubiquitous computing systems by means of the use of autonomous agents, which enable ubiquitous computing technology to respond to users' particular conditions and demands. Autonomous agents were used to implement the desirable features of ubiquitous computing systems and for enhancing the interactions of the users with the environment. In a ubicomp system, autonomous agents were the software components that represent users, devices and services and that are able to make their own decisions about what activities to do, when to do them, what type of information should be communicates and to whom, and how to assimilate the received information.

An agent middleware (SALSA) was created for facilitating the implementation and progressive evolution of this type of intelligent systems. SALSA provides an agent communication platform which is a Broker that acts as a communication channel among agents and users, and a communication protocol that enable agents to convey different types of objects among agents (such as context information), among agents and users (such as events generated by users) and agents and services (such as the state of the services). SALSA provides a set of classes that facilitate the implementation of the agents' components for perceiving, reasoning and acting. And finally, the SALSA middleware provides an Agent Directory in which agents register their services and look for other the services offered by other agents.

The contributions of this thesis focus on presenting the functional requirements of autonomous agents for implementing ubiquitous computing systems and the agent SALSA middleware, which was created with the aim of facilitating the implementation and evolution of ubicomp systems. Finally, this thesis provides evidence of the SALSA flexibility for enabling the progressive development of ubicomp systems by presenting the development of a context-aware hospital information system.

Keywords: ubiquitous computing, autonomous agents, middleware

DEDICATION

To my husband, my son, my parents and sister.

ACKNOLEDGEMENTS

I feel extremely lucky to have worked under the guidance of Jesus Favela, and I am thankful for all the confidence he had in me, and for assigning me important tasks in order to help me to develop as a researcher. He not only gave me the opportunity to work with him, but he advised, encouraged and inspired my research and most importantly became a friend. I also would like to thank my thesis committee, Anind Dey, Leopoldo Moran, Antonio Garcia, and Ana I. Martinez who gave me valuable feedback for finishing my thesis.

At CICESE, I worked closely with many students with an interest in Ubiquitous Computing. In particular I thank Miguel Angel Muñoz, Alfredo Preciado, Pedro C. Santana, and Irma Amaya, and the group of the Fall 2004 Object Oriented Design course who participated in the evaluation of my work. Special thanks to Cecilia Curlango who volunteered to proofread this thesis.

I would like to thank my parents, my sister and my parents-in-law, for their encouragement and all the support I received from them. I would also like to express my gratitude and love to my husband, Angel, for the patience and emotional support he has always provided to me. And to my dear son Angel Roberto, who without knowing it, motivated me over these years.

Thanks to all of you!

Finally, I thank CONACYT and PROMEP for the scholarships granted for the development of the work presented in this thesis.

TABLE OF CONTENT

TABLE OF CONTENT
(Continue)

TABLE OF CONTENT
(Continue)

TABLE OF CONTENT
(Continue)

TABLE OF CONTENT
(Continue)

FIGURE LIST

FIGURE LIST
(Continue)

FIGURE LIST
(Continue)

TABLE LIST

Sinopsis en español (Tabla de contenido)

Sinopsis en español

1 Introducción

1.1 Computación Ubicua

La idea del cómputo ubicuo propone un ambiente físico dotado de dispositivos de diferentes escalas, con capacidades computacionales y de comunicaciones, los cuales se integran de forma natural a nuestras actividades diarias [Weiser, 1991, 1993]. Es decir, los dispositivos computacionales se convierten en una herramienta de trabajo que solo requiere de nuestra atención periférica. La interacción humano-computadora sucede en forma natural e implícita, lo que permite que los usuarios no requieran estar concientes de la existencia de los dispositivos, lo cual es posible mediante el acceso consciente del contexto a información y servicios disponibles en el ambiente [Abowd y Mynatt, 2000]. Por contexto se entiende que es cualquier información relevante para mejorar la interacción entre un usuario y una aplicación, por ejemplo localización, identidad y actividades de los usuarios, [Dey, 2001]. Así, la tecnología conciente del contexto es considerada el corazón de los sistemas de cómputo ubicuo ya que permite que el sistema reaccione ante los cambios de contexto de los usuarios. Debido a que los ambientes de cómputo ubicuo se caracterizan por ser reactivos y pro-activos para proporcionar servicios e información adaptada al contexto de los usuarios, también se denominan ambientes inteligentes. Por lo anterior, recientemente a esta área de la computación se le conoce como Inteligencia Ambiental (AmI por sus siglas en ingles: Ambient Intelligence). Crear sistemas con características como las mencionadas ha llevado a los desarrolladores a enfrentar varios retos tal como se explica en la siguiente sección.

1.2 Retos y complejidades del cómputo ubicuo

Construir sistemas de cómputo ubicuo ha llevado a los desarrolladores a enfrentar retos relacionados con la infraestructura de software cuyo propósito es facilitar que se aborden las complejidades de estos sistemas. Entre estas complejidades están las *fallas* de los sistemas debidas a eventos impredecibles, tal como las desconexiones de los dispositivos móviles; la *heterogeneidad* de los dispositivos computacionales y de comunicación; *la adaptación* de información y servicios para lo cual se debe *considerar el contexto* de los usuarios, que es altamente dinámico; el *descubrimiento de los servicios/dispositivos* disponibles debe requerir mínima o ninguna intervención de los usuarios; finalmente, los sistemas deben ser *escalables* en cuanto al número de usuarios, dispositivos y el tiempo en que el ambiente debe proporcionar sus servicios.

Para abordar algunas de las complejidades de los sistemas de cómputo ubicuo, los desarrolladores han utilizado enfoques de programación (tal como programación orientada a aspectos, y reflexión) y propuesto plataformas de desarrollo (tal como middlewares y librerías de clases) [Popovici *et al.*, 2003; Capra *et al.*, 2003].

1.3 Objetivo de la tesis

El proveer de plataformas que faciliten el desarrollo de sistemas de cómputo ubicuo es otro de los retos de esta área. En este sentido, esta tesis explora el uso de los agentes autónomos como una abstracción para el diseño y construcción de sistemas de cómputo ubicuo y se proporciona un middleware para facilitar la implementación de los agentes. Así, el objetivo de la tesis es:

Diseñar e implementar un middleware que permita a los desarrolladores manejar algunas de las complejidades asociadas con el desarrollo de sistemas de cómputo ubicuo por medio del uso de agentes autónomos.

1.4 Contenido de la tesis

Antes de presentar este middleware llamado SALSA, la sección 2 presenta las plataformas de software existentes que abordan algunas de las complejidades para implementar sistemas de cómputo ubicuo; la sección 3 presenta una introducción a los agentes autónomos; la sección 4 describe la metodología seguida durante esta tesis; en la sección 5 se explican los escenarios de uso de sistemas de cómputo ubicuo que se analizaron para obtener los requerimientos del middleware SALSA; la sección 6 presenta estos requerimientos, y el diseño e implementación de SALSA; en la sección 7 se ilustra como utilizar SALSA para implementar un sistema de cómputo ubicuo; sección 8 describe la evaluación de SALSA; finalmente se presentan algunas conclusiones y trabajo futuro en la sección 9.

2 Trabajo relacionado

Algunas plataformas de software se enfocan en facilitar la implementación de aplicaciones para dispositivos móviles, tal como YCab el cual es un API para crear aplicaciones descentralizadas que permiten a los usuarios colaborar en un ambiente tolerante a fallas ante las desconexiones de los dispositivos móviles [Buzko et al., 2001]. Sin embargo YCab no permite que los usuarios interactúen con otro tipo de dispositivos que podrían ofrecer servicios relevantes para sus actividades. Similarmente, DACIA lidia con las desconexiones de los usuarios móviles permitiendo que una aplicación cliente, residiendo en un servidor, represente al usuario. Este cliente realiza actividades especificas y sencillas, tal como almacenar los mensaje dirigidos al usuario o informar que el usuario no esta disponible. Mientras se reestablece la conexión, el usuario puede continuar utilizando la aplicación en su dispositivo móvil [Litiu y Parkash, 2000]. En [Popovici et al., 2003] se presenta una plataforma que permite a los dispositivos móviles extender o modificar dinámicamente su funcionalidad para adaptar su comportamiento al entorno. Este tipo de adaptación no se realiza tomando en cuenta el contexto de los usuarios, ya que ellos son los que seleccionan la funcionalidad que el dispositivo móvil debe adquirir. CARISMA es un middleware para cómputo móvil que explota el principio de reflexión para adaptar las políticas de asignación de recursos tales como el ancho de banda de la red, memoria y energía de los dispositivos [Capra et al., 2003]. Las anteriores plataformas no toman en cuenta el contexto de los usuarios para realizar tal adaptación o para mejorar la colaboración de los usuarios. Otras plataformas se enfocan en facilitar la implementación de servicios concientes del contexto como las que se presentan a continuación.

RCSM facilita el desarrollo de aplicaciones que requieren comunicación espontánea conciente del contexto mediante un modelo de comunicación descentralizado y un enfoque orientado a mensajes. RCSM permite a los programadores enfocarse en implementar las acciones de las aplicación conciente del contexto, sin preocuparse del monitoreo, detección y análisis del contexto [Yau et al., 2002]. Semantic Space es una infraestructura que mediante tecnologías del Web Semántico facilita que las aplicaciones recuperen información de contexto utilizando consultas declarativas, infieran información de contexto mediante reglas heurísticas, y monitoreen a los dispositivos que dinámicamente se unen al ambiente para abstraer de ellos información de contexto [Wang et al., 2004]. En [Kim et al., 2004] se presenta un middleware que propone un esquema estandarizado para manipular información de contexto. Este esquema llamado HIML (Human Interaction Markup Language) proporciona un lenguaje común para expresar información de conciencia de contexto y facilitar que el usuario modifique o ajuste su propia información contextual. El middleware CAMUS (Context-Aware Middleware for Ubiquitous computing Systems) facilita la composición de contexto y permite que la información de contexto extraída de sensores se represente como tuplas FTS (Feature Tuple Space), el cual es un mecanismo de comunicación y almacenamiento utilizado por el middleware [Hgo et al., 2004].

Otras plataformas de software proporcionan facilidades de programación para abordar los retos de cómputo ubicuo relacionados con el sistema operativo o el middleware sobre el cual se construyen los sistemas. Estas plataformas no solo proporcionan un API que facilita la implementación de los sistemas, sino un conjunto de servicios que liberan a los desarrolladores de abordar algunas complejidades del cómputo ubicuo. Entre estas plataformas está Gaia, la cual proporciona un marco de clases que facilita la construcción de ambientes de cómputo ubicuo y un conjunto de servicios que coordinan las entidades de software y los dispositivos heterogéneos contenidos en el ambiente, facilitan el manejo de eventos o detección de cambios que ocurren en el ambiente, y permiten a las aplicaciones registrar y consultar información de contexto [Román et al., 2002; Ranganathan y Campbell, 2003]. One.world es una arquitectura que permite el desarrollo de aplicaciones de cómputo ubicuo adaptables. [Grimm, 2004]. Entre las facilidades que proporciona está la composición espontánea del ambiente, el descubrimiento de servicios disponibles, y compartir información entre dispositivos utilizando tuplas como medio de almacenamiento. Sin embargo, el mecanismo de descubrimiento de servicios no permite la búsqueda especializada de servicios con características específicas. Finalmente, los componentes del ambiente se comunican por eventos asíncronos, lo cual limita la cantidad de información que puede comunicarse.

Las plataformas de desarrollo presentadas en esta sección proporcionan facilidades para abordar algunas de las complejidades de implementar sistemas de cómputo ubicuo. Por ejemplo, algunas plataformas proponen diferentes técnicas para manipular información de contexto (p. ej. ontologías y lenguajes de marcado). Otros middlewares proponen utilizar diferentes enfoques de programación, tal como programación por aspectos o reflexión para permitir la adaptación de los dispositivos móviles a su entorno [Popovici *et al.*, 2003; Capra *et al.*, 2003]. En esta tesis se explora el uso de agentes autónomos para implementar las características deseables de los sistemas de cómputo ubicuo identificadas de un conjunto de escenarios de uso.

3 Agentes autónomos

Un agente es definido como una entidad de software que funciona continuamente y autónomamente en un ambiente en particular, comúnmente habitado por otros agentes y procesos [Shoham, 1997]. Las características de continuidad y autonomía se derivan del deseo de que un agente realice sus actividades de manera flexible e inteligente para responder a los cambios del ambiente sin requerir la intervención humana. Idealmente, un agente que funciona continuamente por largos periodos de tiempo debe ser capaz de aprender de su experiencia en el ambiente. Finalmente, un agente que habita en un ambiente con otros agentes y procesos debe poder comunicarse y cooperar con ellos [Bradshaw, 1997]. Los agentes de software tienen una o mas de los siguientes atributos: autonomía (para actuar por su cuenta), re-actividad (para responder a los cambios en el ambiente), pro-actividad (para alcanzar sus objetivos); colaboración (con otros agentes), adaptación (para acoplarse al ambiente aprendiendo de su experiencia) y movilidad (para migrar a otros entornos de ejecución) [Bradshaw, 1997; Wooldridge y Jennings, 1995].

4 Metodología

Para explorar el uso de agentes autónomos para diseñar sistemas de cómputo ubicuo y proporcionar un middleware que facilite su implementación, se siguió una metodología que consistió en las etapas ilustradas en la Figura 1. En la primera etapa se seleccionaron los escenarios de uso, los cuales fueron resultado de estudios de campo realizados en el dominio de trabajo de los usuarios. En este caso se seleccionó los hospitales para estudiar como la tecnología de cómputo ubicuo mejora las actividades médicas que involucran coordinación, administración de información, acceso oportuno a información y servicios para permitir una apropiada toma de decisiones. Durante la segunda etapa se identificó como los agentes autónomos pueden ser usados para diseñar sistemas de cómputo ubicuo y se identificaron las características de diseño de estos agentes. En la tercera etapa se obtuvieron los requerimientos para crear un middleware que permitiera crear agentes con tales características.

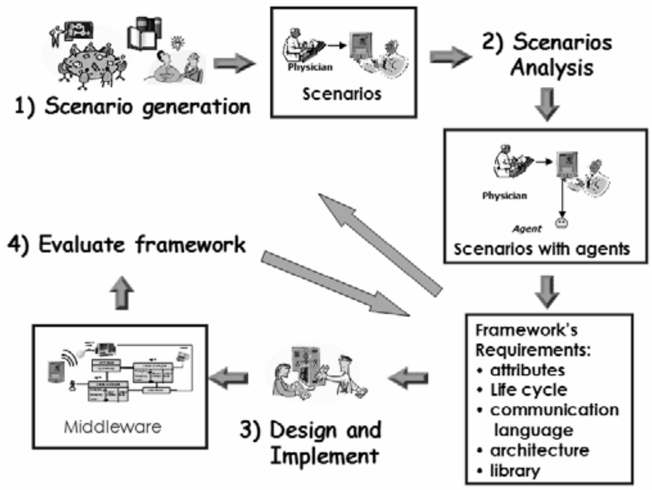


Figura 1. Metodología para crear el middleware SALSA

Finalmente, se evaluó el middleware SALSA y el uso de agentes autónomos para diseñar sistemas de cómputo ubicuo se identificaron los siguientes criterios de evaluación.

- *Utilidad.* Este criterio determina si otros pueden construir aplicaciones de cómputo ubicuo reales utilizando una plataforma de desarrollo.
- *Completo.* Este criterio determina si la plataforma es suficientemente poderosa y extensible para implementar aplicaciones interesantes.
- *Facilidad de uso.* El middleware es evaluado en cuanto a su facilidad de aprenderlo y de uso.

De estos criterios se obtuvieron las hipótesis que permitieron determinar la flexibilidad de los agentes SALSA para implementar y extender la funcionalidad de sistemas de cómputo ubicuo, lo cual se describe en la sección 7.

5 Agentes autónomos para diseñar sistemas de cómputo ubicuo para hospitales

Los siguientes son algunos de los escenarios que se seleccionaron para ilustrar como tecnología de cómputo ubicuo puede mejorar y apoyar las actividades médicas de los hospitales.

5.1 Prácticas médicas concientes del contexto de usuarios móviles

Mediante estudios de campo realizados en el Instituto Mexicano del Seguro Social en Ensenada, B.C., se entendieron algunas de las prácticas médicas, y en base a este entendimiento se propuso como tecnología de cómputo ubicuo conciente del contexto apoya las actividades del personal del hospital. De este estudio surgieron ideas de diseño de un sistema consciente del contexto para hospitales. El siguiente escenario ilustra la funcionalidad de este sistema.

Escenario 1: Acceso conciente del contexto a información médica.

Mientras el Dr. Díaz revisa el estado de un paciente (en la cama 1 del cuarto 222), se da cuenta que es necesario solicitar una prueba de laboratorio para el paciente. Utilizando su PDA, el doctor añade esta solicitud al registro clínico del paciente en el Sistema de Información del Hospital (HIS por sus siglas en inglés: Hospital's Information System). El químico (responsable de tomar las muestras para el análisis) visita el área de medicina *interna cada mañana, entonces su PDA le informa que en la habitación 222 existen tres pacientes que requieren un análisis médico. Cuando el químico está enfrente del paciente, el PDA le muestra el tipo de análisis a realizar. El químico procede a tomar la muestra para el análisis y al final de su ronda la lleva al laboratorio para ser analizada. Los resultados del análisis son agregados al registro clínico del paciente. Cuando el doctor está por terminar su turno de trabajo, y mientras camina por el corredor, su PDA lo alerta que los resultados del paciente de la cama 1 de la habitación 222 están disponibles. El Dr. Díaz regresa al cuarto del paciente, y cuando se detiene cerca de su cama, los resultados del análisis son mostrados en el PDA. El doctor reevalúa al paciente y basándose en los resultados, decide preparar al paciente para intervenirlo quirúrgicamente.*

Escenario 2: Acceso a información médica a través de pantallas públicas

Mientras el Dr. García revisa al paciente de la cama 234, su PDA le alerta que tiene un mensaje nuevo. Para mostrarle este mensaje, el PDA presenta un mapa del piso del área de medicina interna indicando que los resultados de rayos X del paciente en la cama 225 están disponibles. Antes de visitar al paciente, el Dr. García se acerca a la pantalla pública mas cercana, la cual detecta su presencia y le presenta una vista personalizada del Sistema de Información del Hospital (HIS). En particular, le muestra el mapa del piso personalizado, ya que resalta los cambios recientes en los registros clínicos de sus pacientes, los mensajes recibidos y los servicios más relevantes para sus actividades actuales. El Dr. Garcia selecciona el mensaje relacionado a la cama 225, lo que abre una ventana mostrando el registro médico del paciente, la imagen de rayos X recientemente tomada y la guía médica del hospital relacionada con este caso clínico. Mientras el Dr. García analiza la imagen de rayos X, nota en el mapa que un médico residente está cerca y lo llama para mostrarle este caso clínico interesante. El médico residente se da cuenta que es un caso especial, y decide transferir la referencia del caso a su PDA para estudiarlo posteriormente o discutirlo con otros colegas desde cualquier computadora dentro del hospital.

Los componentes principales de los escenarios seleccionados fueron identificados como agentes que tienen capacidades para tomar sus propias decisiones acerca de que actividades hacer, cuando hacerlas, que tipo de información debe ser comunicada y a quien, y como asimilar la información recibida, por ejemplo, la personalizan y adaptan de acuerdo a la localización e identidad de los usuarios.

5.2 Diseño de las características de los agentes autónomos

Los escenarios anteriores fueron utilizados para identificar las siguientes características de diseño de los agentes autónomos en cuanto a su funcionalidad en ambientes de cómputo ubicuo:

- Los agentes autónomos son capaces de tomar decisiones en base al contexto de los usuarios.
- Los agentes autónomos para ser reactivos a información de contexto necesitan mecanismos para percibir, reconocer y diseminar diferentes tipos de información de contexto, tal como localización e identidad de los usuarios, estado de dispositivos, etc.
- Los agentes autónomos pueden ser utilizados para representar usuarios, actuar como proxies a recursos de información (p. ej. el HIS), dispositivos o servicios (tal como la pantalla publica), o implementar una funcionalidad compleja que debe ser transparente a los usuarios (p. ej. estimar la localización del usuario).
- Los agentes autónomos necesitan una plataforma de comunicación que les permita comunicarse con otros agentes o directamente con los usuarios y servicios.
- Los agentes deben estar concientes de la presencia de otros agentes y usuarios disponibles en el ambiente para ofrecerle servicios cuando lo requieran.
- Los agentes necesitan mecanismos para autentificar a los usuarios o a otros agentes que requieren acceder a sus servicios.
- Los agentes requieren comunicar diferentes tipos de mensajes, tal como, mensajes para solicitar información, notificaciones a usuarios, o solicitudes para ejecutar una acción o servicio.

- Para que los agentes decidan como actuar necesitan un algoritmo de razonamiento que puede ser tan simple como un conjunto de reglas o condiciones o un algoritmo más complejo como una red neuronal.

Una vez que se exploró el uso de agentes autónomos para diseñar sistemas de cómputo ubicuo ilustrados a través de escenarios de uso, el siguiente paso consistió en diseñar e implementar un middleware que facilite el desarrollo de ambientes de cómputo ubicuo mediante agentes autónomos. La siguiente sección explica este middleware llamado SALSA (*por sus siglas en inglés: Simple Agent Library for Smart Ambients*).

6 Diseño e implementación del middleware SALSA

6.1 Arquitectura de SALSA

Tal como se ilustra en la Figura 2, el middleware SALSA consiste de los siguientes elementos:

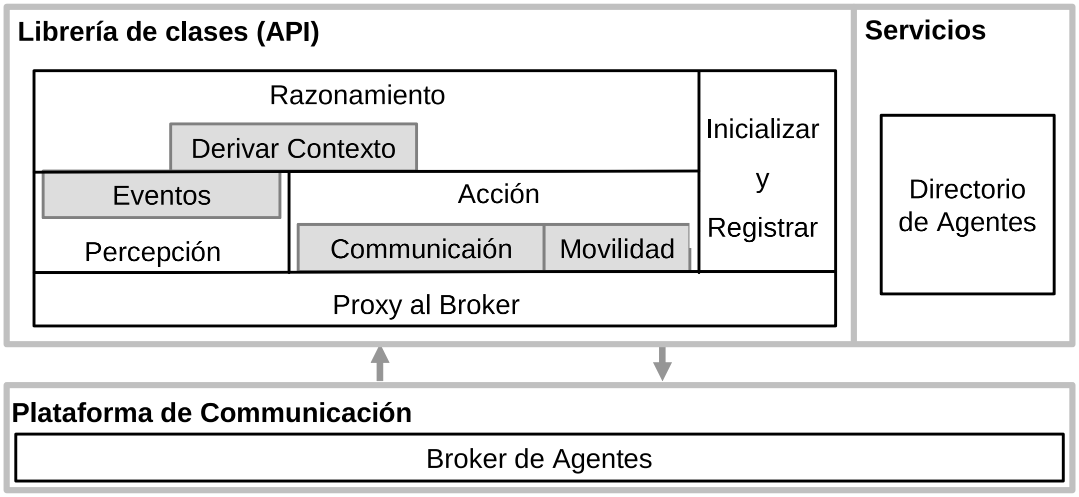


Figura 2. Arquitectura de SALSA

- *Plataforma de comunicación*. Un Broker de Agentes es el canal de comunicación entre agentes y usuarios. La implementación del Broker es un servidor de mensajeria instantánea basado en el protocolo Jabber (www.jabber.org), el cual fue extendido para crear el protocolo de comunicación de SALSA. Este protocolo consiste de un lenguaje que permite a los agentes intercambiar diferentes tipos de objetos entre agentes, (tal como información de contexto, solicitudes de servicio) entre agentes y usuarios (tal como eventos generados por las acciones de los usuarios) y entre agentes y servicios (p.ej. estado de los servicios). Esta información es enviada o recibida por el agente a través de un proxy al Broker, el cual es parte del agente. El proxy al Broker y el conjunto de mensajes que pueden comunicarse entre los agentes son creados mediante la librería de clases de SALSA (API).

- *API de SALSA*. El elemento principal de SALSA es un conjunto de clases (API) que facilita la implementación de los componentes de los agentes. SALSA proporciona una implementación del Proxy al Broker, el cual es un cliente de mensajeria instantánea. A través de este Proxy, el componente de percepción recibe información y genera eventos para comunicar esta información al componente de razonamiento para que sea analizada (p. ej. derivando contexto) o procesada (mediante algún algoritmo de razonamiento) y decida que acción ejecutar. Este componente puede se fácilmente modificado por el programador para implementar o modificar el algoritmo de razonamiento que implemente la funcionalidad deseada del agente. Este razonamiento puede consistir de un simple conjunto de reglas o de un algoritmo más complejo, tal como una red neuronal. Como parte de las acciones, el agente puede requerir comunicarse con otros agentes o usuarios, o actualizar su componente de razonamiento adquiriendo el código del algoritmo de razonamiento de otro agente residiendo en un servidor. Finalmente, el API proporciona un conjunto de clases que permiten al agente registrarse en uno o más Directorios de Agentes y solicitar información de otros agentes con los que requiera interactuar.
- *Servicios*. SALSA provee de un Directorio de Agentes para que los agentes registren sus servicios y realicen búsquedas de otros agentes. Un agente puede buscar información de otro agente con el cual requiera interactuar buscando por el servicio que ofrece. La implementación del Directorio de Agentes consiste de un servidor LDAP (Lightweight Directory Access Protocol y un agente actuando como proxy al Directorio

de Agentes (*AD-proxy agent*). Las solicitudes de información o de registro de agentes son atendidas por el agente Proxy al Directorio de Agentes.

6.2 API de SALSA

La Figura 3 presenta el conjunto de clases de SALSA que facilitan la implementación de la arquitectura interna de los agentes dada por sus componentes de percepción, razonamiento y acción.

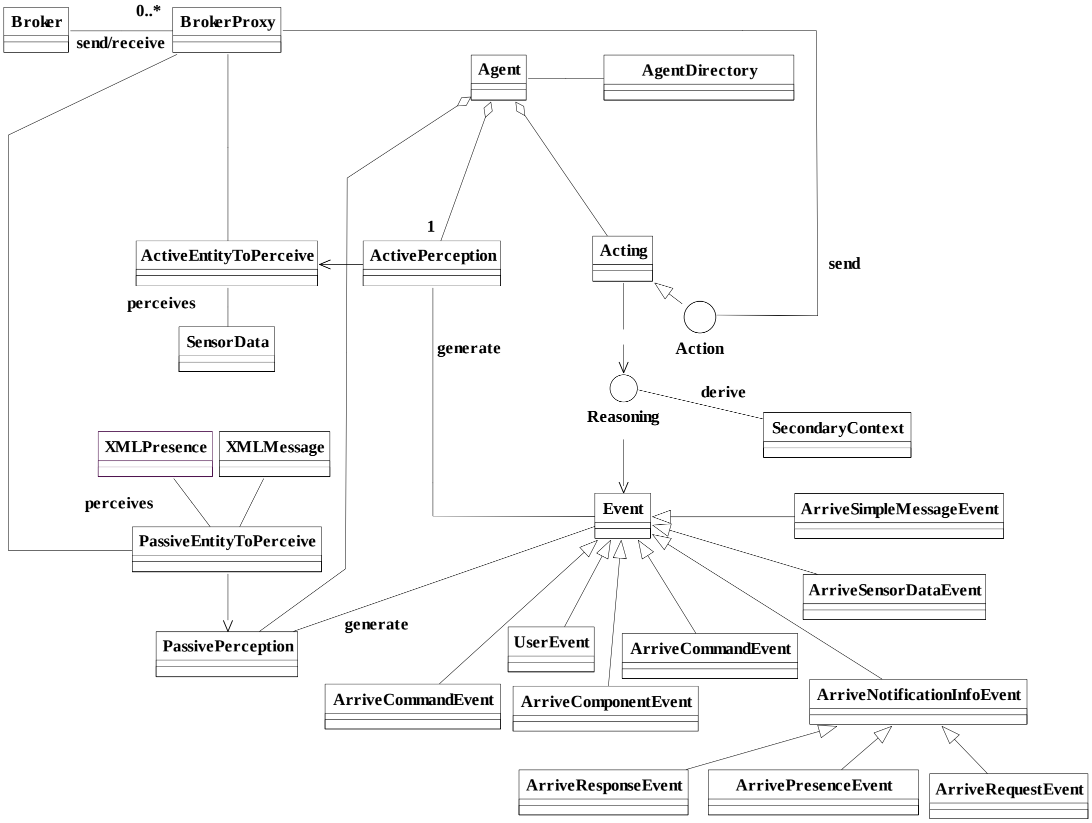


Figura 3. Librería de clases de SALSA

6.2.1 Componente de percepción

Dos tipos de percepción se identificaron para los agentes SALSA: activa y pasiva. En percepción activa un agente decide cuando solicitar información de otro agente o entidad del ambiente, tal como un sensor. En percepción pasiva, el agente está en un modo observador, y recibe información sin solicitarla.

Para implementar la percepción pasiva, el programador especifica mediante la clase `PassiveEntityToPerceive` la entidad (un agente o dispositivo) que el agente requiere observar. A través de esta clase, esta entidad enviará información al componente de percepción del Agente implementado como un objeto de la clase `PassivePerception`, el cual se crea automáticamente cuando se inicializa al agente. El objeto `PassivePerception` genera un evento para indicarle al componente de razonamiento el tipo de información recibida. Cuando el agente requiere comunicarse con otros agentes a través del Broker de Agentes, el Proxy al Broker requiere implementar la clase `PassiveEntityToPerceive` para poder comunicar información.

Similarmente, en la percepción activa el programador requiere implementar un objeto de la clase `ActiveEntityToPerceive` que representa la entidad (dispositivo o agente) que enviará información disponible al componente de percepción del agente (`ActivePerception`).

6.2.2 Componente de razonamiento

La clase `Reasoning` contiene el método abstracto `think` que debe ser especializado por el desarrollador de acuerdo con la lógica del agente. El componente de razonamiento utiliza las facilidades de SALSA para derivar información de contexto a partir de contexto primario percibido por el agente. Para esto, SALSA proporciona la clase `DeriveContext` que permite derivar contexto solo a partir de un conjunto de reglas especificadas por el programador en un filtro XSL eXtensible Style Language (XSL). El contexto derivado es especificado como un mensaje XML.

6.2.3 Componente de acción

Las clase abstracta `Action` permite al desarrollador especificar las acciones que el agente ejecutará. A través del componente de acción los agentes pueden comunicase con otros agentes o usuarios. Los métodos de comunicación de SALSA se presentan en la Tabla

1, los cuales generan mensajes XML (eXtensible Markup Language) que especifican el tipo y la información a comunicar mediante etiquetas definidas por el programador.

Tabla 1. Métodos de comunicación de SALSA

Métodos	Mensajes de SALSA en formato XML
sendCommandRequest()	<message to='**agentA**@server_jabber' from='agentB@server_jabber'> <x xmlns='x:command'> <params><type>**TypeOfCommand** </type> **// TAGS DEFINE BY THE DEVELOPER** </params> </x> </message>
sendResquest()	<message to='**agentA**@server_jabber' from='agentB@server_jabber'> <x xmlns='x:request'> <params><type>**TypeOfRequest** </type> **// TAGS DEFINE BY THE DEVELOPER** </params> </x> </message>
sendResponse()	<message to='**agentA**@server_jabber' from='agentB@server_jabber'> <x xmlns='x:response'> <params><type>**TypeOfResponse** </type> **// TAGS DEFINE BY THE DEVELOPER** </params> </x></message>
sendNotificationInfo()	<message to='**agentA**@server_jabber' from='agentB@server_jabber'> <x xmlns='x:notificationInfo'> <params><type>**TypeOfNotification** </type> **// TAGS DEFINE BY THE DEVELOPER** </params> </x> </message>
sendDataSensor()	<message to='**agentA**@server_jabber' from='agentB@server_jabber'> <x xmlns='x:dataFromSensor'> <params><type>**TypeOfData** </type> **//SENSED DATA** </params> </x> </message>

7 Implementando un sistema de cómputo ubicuo con SALSA

Para ilustrar la flexibilidad de SALSA para desarrollar un sistema de cómputo ubicuo, esta sección explica la implementación de uno de los agentes autónomos del sistema conciente de contexto para hospitales descrito en la sección 5. Este agente reside en el PDA y tiene como objetivo estimar la localización del usuario móvil.

7.1 Estimando la localización del usuario

Para estimar la localización de los usuarios móviles se han utilizado técnicas que toman ventajan de la escalabilidad, y bajos costo de mantenimiento e instalación que ofrecen las redes locales inalámbricas (*WLAN, Wireless Local Area Network*). Estos métodos miden la fuerza de la señal de radio-frecuencia (RF, Radio Frequency) entre el dispositivo móvil y varios puntos de acceso a la red. [Bahl y Padmanabhan, 2000].

Para estimar la localización de los usuarios del sistema descrito en los escenarios de la sección 5, se propuso un agente autónomo residiendo en el PDA cuyo algoritmo de razonamiento es una red neuronal, que una vez entrenada, fue utilizada para clasificar los patrones percibidos de la señal de RF en coordenadas X,Y que indicaban la posición aproximada del usuario. Mas detalles de este trabajo se describe en [Rodriguez *et al.*, 2004].

Figura 4 ilustra el diagrama de clases del agente que estima la localización (LE-a) y un diagrama de secuencia mostrando las interacciones de los componentes del agente. Cada componente son instancias de las clases SALSA. El módulo de percepción del agente recibe la fuerza de la señal (SNR- Signal to Nose Ratio) a través del objeto `PassiveEntityToPerceive`, el cual representa la memoria de la tarjeta de red inalámbrica del PDA. La información percibida se encapsula en un objeto `Input` para ser notificada (`notifying()`) al componente de percepción, el cual genera un evento de tipo `StateChangeEvent` para indicarle al componente de razonamiento que hay información nueva para procesar. Así, el componente de razonamiento obtiene una estimación de la localización del usuario e invoca al componente de acción (*Acting*) para informarle al cliente conciente del contexto (*location-aware client*) las coordenadas X,Y del usuario. Este cliente traduce las coordenadas a una etiqueta indicando la localización del usuario dentro del ambiente físico (p.ej. cuarto 222).

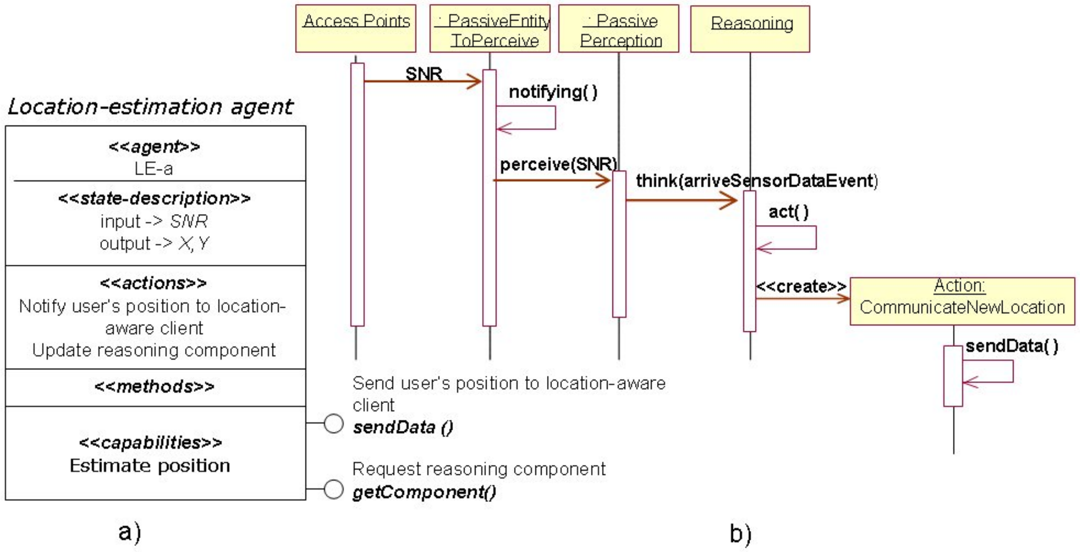
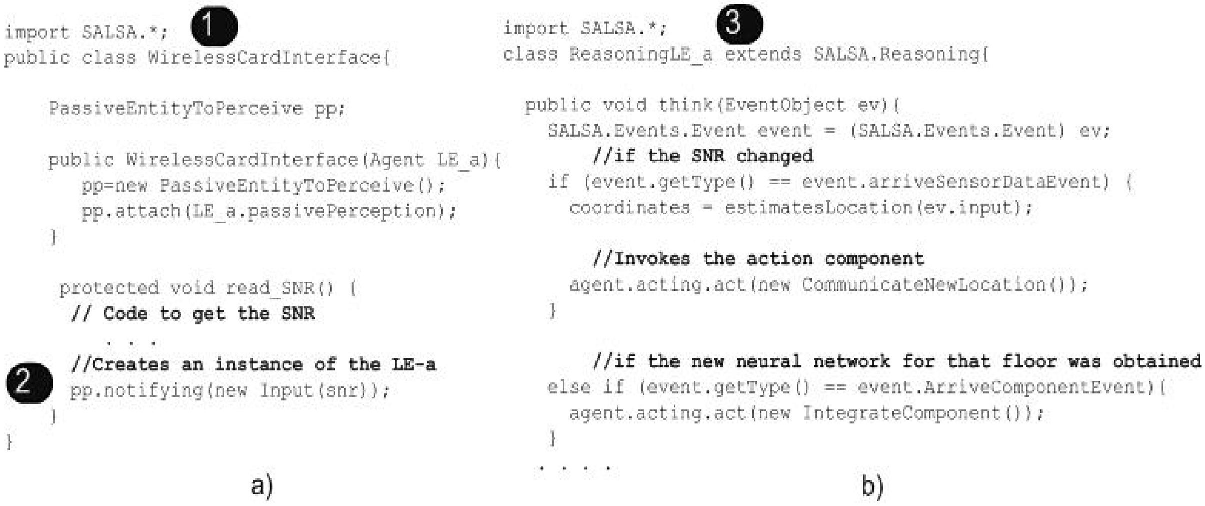

Location-estimation agent
<<agent>>
LE-a
<<state-description>>
input -> SNR
output -> X, Y
<<actions>>
Notify user's position to location-aware client
Update reasoning component
<<methods>>
<<capabilities>>
Estimate position
Send user's position to location-aware client
sendData ()
Request reasoning component
getComponent()
a)
Access Points
: PassiveEntity ToPerceive
: Passive Perception
Reasoning
SNR
notifying( )
perceive(SNR)
think(arriveSensorDataEvent)
act( )
<<create>>
Action: CommunicateNewLocation
sendData( )
b)

Figura 4. Diagrama de clases y de secuencias del LE-a (*location-estimation agent*)

Figura 5a) muestra el pseudo-código del componente de la entidad que percibe información de la tarjeta de red inalámbrica y que la notifica al componente de percepción (`pp.notifying(new Input(SNR))`). Figura 5b) presenta la implementación del componente de razonamiento del LE-a. La clase `ReasoningLE` especializa la clase `Reasoning`. Su método `think()` fue sobre-escrito para implementar el algoritmo de razonamiento y decidir como actuar. Por ejemplo, cuando el razonamiento recibe un evento de tipo `stateChangeEvent`, se invoca al método `estimatesLocation()` el cual implementa la red neuronal entrenada para que estime la nueva localización del usuario. Mediante el componente de acción, implementado por el objeto `communicateNewLocation`, se notifica al cliente al cliente conciente del contexto la nueva posición del usuario.



```
import SALSA.*;
public class WirelessCardInterface{

    PassiveEntityToPerceive pp;

    public WirelessCardInterface(Agent LE_a){
       pp=new PassiveEntityToPerceive();
       pp.attach(LE_a.passivePerception);
    }

     protected void read_SNR() {
      // Code to get the SNR
         . . .
      //Creates an instance of the LE-a
      pp.notifying(new Input(snr));
    }
}
```

```
import SALSA.*;
class ReasoningLE_a extends SALSA.Reasoning{

  public void think(EventObject ev){
    SALSA.Events.Event event = (SALSA.Events.Event) ev;
        //if the SNR changed
    if (event.getType() == event.arriveSensorDataEvent) {
      coordinates = estimatesLocation(ev.input);

        //Invokes the action component
      agent.acting.act(new CommunicateNewLocation());
    }

        //if the new neural network for that floor was obtained
    else if (event.getType() == event.ArriveComponentEvent){
      agent.acting.act(new IntegrateComponent());
    }
 . . . .
```

Figura 5. Pseudo-código del LE-a. a) Implementación de la entidad que lee las señales SNR de la tarjeta de red inalámbrica. b) Implementación del componente de razonamiento

7.2 Sistema consciente del contexto para hospitales

El sistema consciente del contexto para hospitales (*CHIS*, *Context-aware hospital information system*) es un sistema que permite al personal médico localizar documentos relevantes para sus actividades, tal como registros clínicos de los pacientes y resultados de laboratorio; localizar pacientes y colegas; y localizar y conocer la disponibilidad de los dispositivos disponibles en el ambiente, tal como equipo médico u otros recursos computacionales como pantallas públicas de pared [Favela *et al.*, 2004; Muñoz *et al.*, 2003b].

7.2.1 Architecture of CHIS

La Figura 6 presenta la arquitectura del sistema CHIS compuesta por los nodos principales del sistema en donde se ejecutan los agentes quienes se comunican a través del Broker.

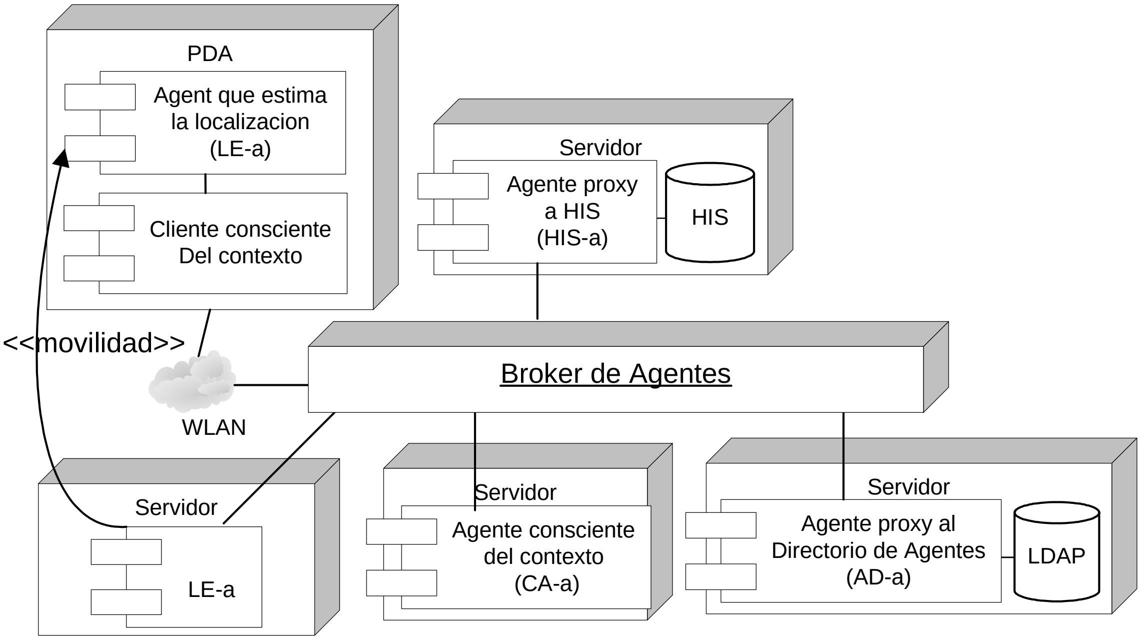

PDA
Agent que estima la localizacion (LE-a)
Cliente consciente Del contexto
Servidor
Agente proxy a HIS (HIS-a)
HIS
<<movilidad>>
Broker de Agentes
WLAN
Servidor
LE-a
Servidor
Agente consciente del contexto (CA-a)
Servidor
Agente proxy al Directorio de Agentes (AD-a)
LDAP

Figura 6. Arquitectura del sistema consciente del contexto para hospitales

En el PDA de los usuarios reside el cliente consciente del contexto el cual notifica la localización y presencia del usuario a otros usuarios y agentes; proporciona información relevante a la localización del usuario, y permite que el usuario se comunique con otros colegas. En el PDA también reside el agente que estima la localización del usuario (LE-a) y la notifica al cliente consciente del contexto. El LE-a actualiza su componente de razonamiento obteniendo de otro agente residiendo en un servidor, la red neuronal entrenada para estimar la localización de usuarios en ese piso o área del hospital en particular. HIS-a es el agente que actúa como proxy al sistema de información del hospital, ya que administra y almacena la información médica de los pacientes. Este agente permite el acceso a la información de los pacientes contenida en el sistema de información del hospital y monitorea los cambios a esta información. Finalmente, el agente conciente del contexto (Ca-a) es el componente del sistema que envía los mensajes que dependen de variables contextuales para su envío, tal como localización del usuario e identidad.

7.2.2 Implementation of CHIS

La Figura 7 ilustra la funcionalidad del sistema de información consciente del contexto y como los agentes interactúan utilizando el protocolo de comunicación de SALSA.

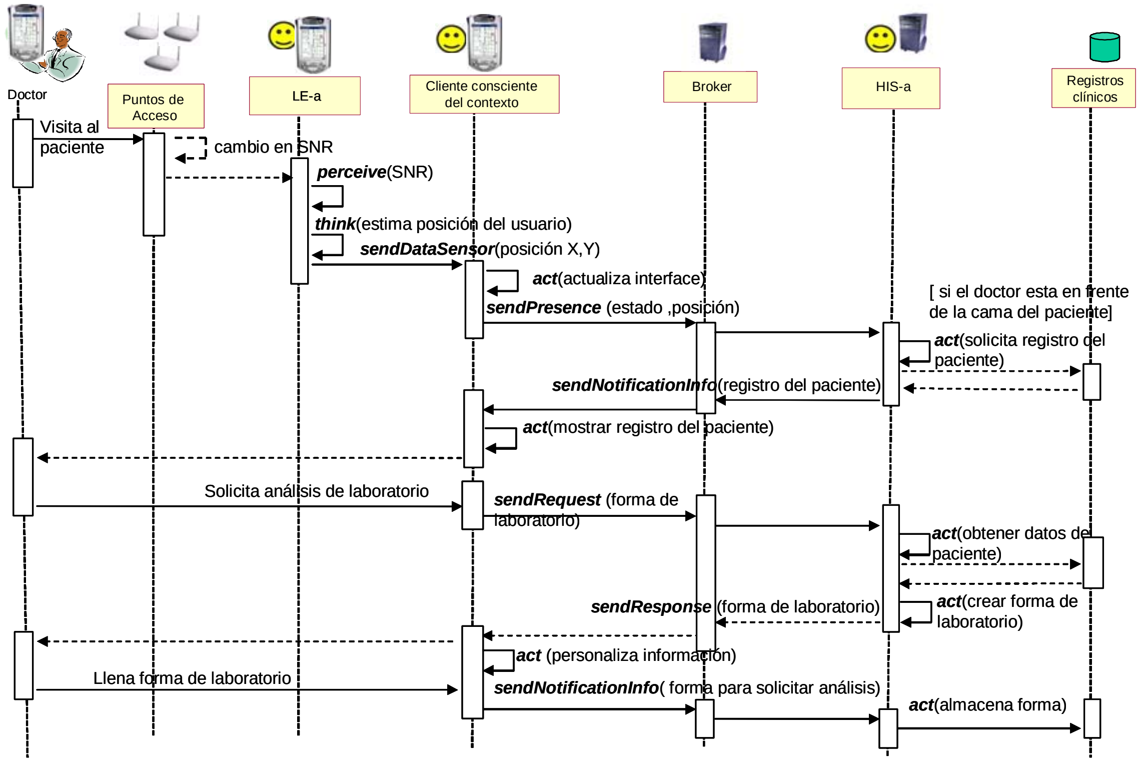

Doctor
Puntos de Acceso
LE-a
Cliente consciente del contexto
Broker
HIS-a
Registros clínicos
Visita al paciente
cambio en SNR
perceive(SNR)
think(estima posición del usuario)
sendDataSensor(posición X,Y)
act(actualiza interface)
sendPresence (estado ,posición)
[si el doctor esta en frente de la cama del paciente]
act(solicita registro del paciente)
sendNotificationInfo(registro del paciente)
act(mostrar registro del paciente)
Solicita análisis de laboratorio
sendRequest (forma de laboratorio)
act(obtener datos del paciente)
sendResponse (forma de laboratorio)
act(crear forma de laboratorio)
act (personaliza información)
Llena forma de laboratorio
sendNotificationInfo(forma para solicitar análisis)
act(almacena forma)

Figure 7. Interacciones de los agentes de CHIS para proporcionarle al médico acceso al expediente electrónico del paciente

A continuación se describen las interacciones entre los agentes de CHIS para ofrecer la funcionalidad descrita en el primer escenario de la sección 5.1. Cuando el Dr. Díaz está en frente de uno de sus pacientes, el LE-a percibe (`perceive(SNR)`) el cambio de SNR (señal de ruido) y estima la posición del usuario mediante la red neuronal implementada en el metodo `think(estima posición del usuario)`, y la comunica al cliente consciente del contexto (`sendDataSensor(posición X,Y)`). El cliente traduce las coordenadas X,Y a un identificador del lugar (p. ej. cuarto 222), actualiza su interfaz de usuario para mostrar la localización del usuario (`act(actualiza interfaz)`) y la comunica como parte de la presencia del usuario a los agentes y usuarios suscritos a ésta. (`sendPresence()`). El HIS-a procede a verificar si las condiciones contextuales concuerdan con el contexto percibido (p.ej. role y posición del usuario), y como el Dr. Díaz está en frente de una de las camas de sus pacientes, el HIS-a recupera los registros médicos del paciente (`act(solicita registro del paciente)`) y los envía al cliente consciente del contexto (`sendNotificationInfo()`) para que sean analizados por el médico. Después de consultar al paciente, Dr. Díaz decide solicitar un análisis de laboratorio, para lo cual el cliente consciente del contexto se comunica con el HIS-a (`sendRequest(forma de laboratorio)`). Este agente crea la forma de laboratorio la cual incluye los datos generales del paciente (`sendResponse(forma de laboratorio)`). El doctor llena la forma y la envía nuevamente al sistema de información del hospital para ser almacenada (`sendNotificationInfo(forma para solicitar análisis)`). Así, cuando el químico encargado de tomar las muestras de análisis ha llegado al hospital para iniciar su ronda, el HIS-a detecte le notificará de los análisis pendientes.

En la siguiente sección se presenta la discusión de los resultados de la evaluación del mddleware SALSA.

8 Discusión

Para evaluar SALSA se identificaron que los aspectos relevantes de un middleware son utilidad, qué tan completo es, y facilidad de uso [Klemmer *et al.*, 2004] [Grimm, 2004]. Así, la evaluación de SALSA se enfocó en determinar como SALSA cumplía con estos aspectos. Se aplicaron varios métodos de evaluación que incluyó evaluar los escenarios identificados, utilizar el API de SALSA en un experimento de laboratorio y un ejercicio de diseño de un sistema de cómputo ubicuo en donde participaron alumnos del curso de Análisis y Diseño de Software del periodo Septiembre-Diciembre del 2004, y finalmente un estudio comparativo entre el middleware SALSA y JADE el cual también permite implementar agentes autónomos.

El criterio para determinar la utilidad de SALSA establecía que si éste permite crear aplicaciones reales de cómputo ubicuo, entonces el middleware es útil. La evaluación de los escenarios seleccionados de cómputo ubicuo, determinó que fueron percibidos como fácil de utilizar por el personal del hospital, por lo que SALSA también fue considerado útil para crear este tipo de sistemas [Favela *et al.*, 2004; Muñoz *et al.*, 2003a; Muñoz *et al.*, 2003b].

Para evaluar que tan completo es SALSA, se llevó a cabo un experimento donde se determinó que SALSA es lo suficientemente flexible para permitir la implementación y evolución de sistemas de cómputo ubicuo mediante agentes autónomos. Evidencia de lo anterior se presenta en el documento extendido de la tesis y en la sección 7 de este resumen.

Los resultados del ejercicio de programación y de diseño proporcionan evidencia de que el modelo de ejecución de SALSA (percepción, razonamiento y acción) y las facilidades para implementar agentes son comprensibles. Para algunos de los participantes el uso de agentes autónomos como abstracción para desarrollar un sistema de cómputo ubicuo no fue sencillo, ya que los participantes no entendieron varios de los conceptos relacionados con agentes y las facilidades proporcionadas por SALSA, tal como el protocolo de comunicación de agentes y el uso de XSL para derivar contexto.

Finalmente, para complementar la evaluación de SALSA, se realizo una comparación con el middleware JADE-LEAP. Esta parte de la evaluación consistió en un ejercicio de diseño de un mismo sistema utilizando ambos middlewares. Como resultado de la evaluación se concluyó que el protocolo de comunicación de SALSA es más apropiado para crear ambientes de cómputo ubicuo que el proporcionado por JADE-LEAP.

9 Conclusiones

Esta tesis ha presentado un middleware que facilita la creación de sistemas de cómputo ubicuo mediante el uso de agentes autónomos. En estos sistemas los agentes autónomos se utilizaron para implementar las características deseables de los sistemas y para mejorar la interacción de los usuarios con el ambiente. Los agentes autónomos fueron identificados como los componentes de software que representan usuarios, dispositivos y servicios.

Las contribuciones de esta tesis proporcionan evidencia de como los agentes autónomos pueden ser utilizadas como una herramienta de abstracción para diseñar e implementar sistemas de cómputo ubicuo y presenta un middleware que facilita la implementación de estos agentes. Estas contribuciones se resumen a continuación.

- Se identificó un conjunto de escenarios de sistemas de cómputo ubicuo que reflejaban las actividades reales del personal médico en hospitales.
- Se identificó los aspectos de diseño que los agentes autónomos deben cumplir para permitir el desarrollo de sistemas de cómputo ubicuo.
- Alguno de los aspectos de diseño identificados para agentes autónomos, fueron abordados mediante el diseño e implementación de un middleware de agentes que facilita el desarrollo de sistemas de cómputo ubicuo.
- Uso del middleware propuesto en otros trabajos de investigación, a través de los cuales se han identificado aspectos de SALSA que podrían mejorarse o abordarse como trabajo futuro, tal como se explica a continuación.

Durante las diferentes etapas de esta tesis se identificaron varios aspectos de funcionalidad de los agentes y de sistemas de cómputo ubicuo, algunos de los cuales no fueron abordados por el middleware:

Soporte a las desconexiones de los usuarios móviles. Siendo este un aspecto relevante que afecta la interacción natural e implícita que caracteriza a los ambientes de cómputo ubicuo, es necesario identificar escenarios reales en los cuales las actividades de los usuarios son afectadas, proponer como los agentes autónomos pueden abordar este problema, y finalmente identificar los mecanismos a incorporar al middleware.

Autentificación de agentes. Los agentes deben ser capaces de identificar si el usuario o agente que solicita sus servicios tiene privilegios de accederlos. Para integrar un mecanismo de autentificación a SALSA, es necesario primero analizar la información a tomar en cuenta para la autentificación, además de la identificación o rol de los usuarios.

Por ejemplo, en un hospital, los médicos modifican los registros de los pacientes para dar un diagnóstico, pero las enfermeras solo pueden accederlos para leer las indicaciones de los médicos. Considerando que en los ambientes de cómputo ubicuo las interacciones suceden de forma natural e implícita, o requiriendo pocas interacciones, se debe explorar cuales elementos de privacidad y seguridad de la información deben ser considerados para realizar esta autentificación.

Proporcionar canales de comunicación alternos para los agentes. Se identificó que el proveer de un Broker de Agentes como el de SALSA no resulta apropiado para algunas interacciones entre agentes, como aquellas que requieren transferir grandes cantidades de información. Se propone explorar si SALSA debe permitir que los agentes decidan que canal de comunicación elegir de acuerdo al tipo de interacción requerida y al contexto de la interacción del usuario.

Derivar información de contexto secundario. Proporcionar mecanismos para derivar contexto secundario con SALSA, es un paso inicial para explorar otros mecanismos complejos que permitan que los agentes infieran contexto.

Esta tesis exploró el uso de agentes autónomos para abordar algunas de las complejidades abstraídas de escenarios de cómputo ubicuo. Estos escenarios fueron utilizados como una herramienta para definir la funcionalidad de los agentes autónomos, identificar como pueden apoyar y mejorar las actividades de los usuarios, e ilustrar como pueden ser utilizados como abstracciones de diseño de los sistemas de cómputo ubicuo. Se presentaron los requerimientos funcionales de los agentes autónomos para implementar sistemas de cómputo ubicuo, y se presentó el diseñó e implementación del middleware SALSA creado para facilitar la implementación y evolución de sistemas de cómputo ubicuos mediante agentes autónomos. Y finalmente, se proporcionó evidencia de cómo SALSA facilita la creación de sistemas de cómputo ubicuo.

Chapter 1

Introduction and motivation

The idea of ubiquitous computing, also known as pervasive computing, first arose from contemplating the place of today's computer in actual activities of everyday life. Anthropological studies of human daily activities teach us that people primarily work in a world of shared situations in which the computer is too often the focus of attention, rather than being a tool through which we work by disappearing from our awareness [Weiser, 1991, 1993]. The term ubiquitous computing was coined by Mark Weiser in 1988 to define an environment in which computers are embedded in the objects we use everyday [Weiser, 1991].

The goal of ubiquitous computing is to enhance computer use by making many computers available throughout the physical environment which are effectively invisible to the user [Weiser, 1993]. This environment is furnished with computational resources of all scales that provide information and services when and where desired, such as, digital tablets, wall-sized electronic whiteboards, laptops, handhelds and personal digital assistants (PDAs), some of which allow greater user mobility in pervasive environments. Undoubtedly, the development and deployment of the necessary infrastructure to support continuous mobile computation is arriving [Abowd and Mynatt, 2000]. However, ubicomp promises more than just infrastructure, it suggests new paradigms of interaction and collaboration between users and/or services, inspired by widespread context-aware access to information and computational capabilities [Abowd and Mynatt, 2000], which have led developers of ubicomp systems to cope with several challenges for creating these systems. Such challenges have major implications for the software infrastructure that must facilitate the progressive development of a ubicomp system.

1.1 Ubiquitous computing challenges

Building a ubicomp system requires developers to address several challenges in order to cope with the complexities associated to the development of ubiquitous computing systems. Among these complexities are the *routine failures* that happen due to unpredictable events, for instance those related to disconnections of mobile users; *heterogeneity* in computing, communication and sensing devices embedded in the physical environment which lead to challenges related to the adaptation of information in more than one computing device as required by the user. From this, new challenges for ubiquitous systems arise because *adaptation* must often take place without human intervention to achieve what Weiser calls calm computing [Weiser and Brown, 1996] since it should involve taking into account the user's context which is highly dynamic. Other challenges have surfaced in relation to the *gathering and inferring of context information*. An additional complexity is the *discovery of service/devices* which should be provided by the ubicomp infrastructure by requiring minimal or no configuration at all from the user. Finally, issues regarding *scalability* are implicit in the ubicomp paradigm that reinforce the desire to break the human away from desktop-bound interaction thereby opening up opportunities for other users and new devices to join the environment at any time, which implies scaling with respect to devices, people, and time.

1.1.1 Providing a development platform: a major challenge

The ubicomp research community has emphasized the importance of facilitating the development of ubicomp systems, providing support in order for applications to be easily reused by other developers as well as integrated into larger applications, and that ubicomp system should be implemented in an open and extensible manner to enable their reuse and the system evolution [Banavar and Bernstein, 2002; Davies and Gellersen; 2002, Kindberg and Fox, 2002]. Thus, providing such software support for building ubicomp systems is a major challenge that has been addressed by other research projects. However, the existing development architectures have not addressed how they facilitate the evolution of a pervasive system. Instead they provide support for dealing with some of the complexities of ubiquitous computing systems. Developers have adopted programming techniques already used for developing complex distributed software systems and have also proposed new development platforms, such as middlewares, toolkits and development frameworks, that facilitate the building of ubicomp systems. For instance, some of the middlewares use non-traditional programming approaches, such as aspect oriented programming or reflection, to enable the adaptation of mobile computing devices in a ubicomp environment [Capra *et al.*, 2003, Popovici *et al.*, 2003].

1.1.2 Selection of a programming approach for developing ubicomp systems

There are several programming approaches that may be used for developing a system. This section presents a review of the most used approaches for developing complex systems, and a discussion of what characteristics they offer for developing ubiquitous computing systems in particular.

Object Oriented Programming (OOP), Component based software development (CBSD), Agent Oriented Software Development (AOSD) and more recently, aspect-oriented software development (AOSD) have been proposed to tackle problems experienced during the software engineering process.

Object-oriented programming (OOP) is a technology that can fundamentally aid software engineering, because the underlying object model provides a better fit with real domain problems. *Objects* provide a high-level primitive notion of modularity for directly modeling applications [Wegner, 1990]. In CBSD, a full-fledged software-system is developed by assembling a set of pre-manufactured *components* which are the main building blocks. Each *component* is a black-box entity, which can be deployed independently and is able to deliver specific services through a public interface [Suvee et al., 2003]. The aim of this paradigm is to improve the speed of development and the quality of the produced software. Since the previous approaches were not sufficient to clearly capture all the important design decisions the program must implement, a new programming approach was proposed: Aspect Oriented Software Development (AOSD). Contrary to the other approaches, *aspects* tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways. Some aspects are so common that they can easily be thought about without reference to any particular domain. One example is failure handling. Many performance-related issues can be *aspects,* because performance optimizations often exploit information about the execution context that components span [Suvee et al., 2003].

The granularity of these building blocks affects component reusability, productivity and maintenance. In object-oriented programming, the basic building blocks are objects, which are implemented as classes. The reuse of individual classes will not bring significant productivity leaps because the granularity is small. Object-Oriented libraries enable developers to compose an application from a set of classes from which objects may be created. There are two levels of composition in OOP: declarative composition of classes (by inheritance) and execution-time composition of objects. Objects are composed by specifying module interconnections in a module interconnection formalism. Composition of objects requires declarative composition of interfaces, specification of module interconnections, and redrawing the boundary between public and private information [Wegner, 1990]. The aim of CBSD is to find for each application the joints that allow components to be designed for assembly into products with minimal “glue”, in which the glue is a programming language. Thus, the properties of components, the notion of composition, and the nature of glue are very different between OOP and the CBSD approach. The reusable components in CBSD should have larger granularity than traditional objects and packages [Wegner, 1990].

Some applications are inherently more decomposable than others, and there is no guarantee that such joints will necessarily exist. [Wegner, 1990]. As mentioned at the beginning of Section 1.1, ubiquitous computing (ubicomp) systems are characterized by the heterogeneity and distribution of their entities or components (software and hardware) that may need to interact for providing information and services whenever users need them. Thus, these systems may have a large number of parts with many interactions. For dealing with these systems’ characteristics, the OOP approach does not address the problem of communication among heterogeneous components. An object’s message may request only one operation, and that operation may only be requested via a message formatted in a very exacting way [Odell, 2002]. Adding a new component to the ubicomp environment may require the newly added object(s) to know a-priori the objects with which they can interact and the exact format of their messages for invoking operations. Thus, the OOP is not a flexible paradigm for implementing the spontaneous integration of new services or easily adding an extra functionality to the ubicomp environment. Similarly, in the CBSD approach, the *components* interact through their published interface which also has to be known a-priori by the other systems' *components*. Adding new system functionality or a service to the ubicomp system may be easy from the point of view of implementation and integration. However, changing the behavior of an existing *component* may require creating a new component for replacing the old one, and if its communication interface has changed, replacing other system's *components* will also be required. Finally, since ubicomp environments are highly dynamic (i.e. the context of the users) and unpredictable events occur (i.e. disconnections, integration of new devices), identifying the system's behavior or functionality in order to encapsulate it in a component could not be feasible.

On the other hand, software agents have a more "opaque" notion of encapsulation which means that agent behavior can be unpredictable. Agents are commonly designed to determine their behavior based on individual goals and states, as well as the states of ongoing conversations with other agents [Odell, 2002]. For this, agents need to be reactive to external events, and autonomously initiate internal or external behavior at anytime (when it receives a message o when it perceives information from the environment). These characteristics of reactivity and autonomy are also desirable characteristics of the components of a ubicomp system to achieve "calm computing". The underlying agent communication model is usually asynchronous. This means that there is no predefined flow of control from one agent to another. Finally, agents are naturally interactive since they communicate through a common and rich communication language, as humans do. This enables agents to negotiate (for services, information, or for making a decision), coordinate (for accessing a resource) or simply for transmitting information. This characteristic of collaboration of agents can be used for enabling the transparent interaction of the ubicomp system's components required for providing opportunistic services to the user.

Thus, this thesis proposes using autonomous agents to deal with the complexities abstracted from ubicomp application scenarios which can be related to the challenges mentioned above. The following sections explain what an autonomous agent is and why they are more appropriate for conceiving ubiquitous computing systems.

1.2 Autonomous agents

A software agent is a software entity that acts on behalf of someone to carry out a particular task which has been delegated to it. To do this, an agent might be able to infer users' preferences and/or needs by taking into account the peculiarities of users and situation. This definition is based on the notion of agenthood as an *ascription* made by some person [Bradshaw, 1997] .

Other researchers provide a definition of a software agent based on a *description* of the attributes that an agent may need to act on behalf of someone or something else. Under this approach, each agent might possess a greater or lesser degree of attributes which have to be consistent with the requirements of a particular problem. Some of these attributes are the following: autonomy (to act on their own), re-activity (to respond to changes in the environment), pro-activity (to reach goals), cooperation (with other agents to efficiently and effectively solve tasks), adaptation (to learn from experience) and mobility (to migrate to new places) [Bradshaw, 1997; Wooldridge and Jennings, 1995].

For some researchers, particularly those working in Artificial Intelligence (AI), the term "agent" has a stronger meaning than the one presented above. By agent they mean a computer system that in addition to the properties identified above, is conceptualized using terms that are usually applied to humans. Thus, it is quite common in AI to characterize an agent using mentalist notions, such as knowledge, beliefs, desires, intentions or obligations [Wooldridge and Jennings, 1995].

As the aim of this thesis is to determine if autonomous agents are an appropriate metaphor for designing and implementing ubiquitous computing systems, the following section presents an analyses of the advantages of using autonomous agents from the perspective of different areas involved in the building of ubicomp systems, such as Software Engineering, Distributed Systems and Human-Computer Interaction.

1.3 Advantages of using software agents for building ubicomp systems

Software agents have been introduced in many fields of computer science. This makes the term elusive, since some authors emphasize their distributed nature while others think about agents from the perspective of being able to exhibit intelligent behavior. Being ubicomp a multidisciplinary field, it can take advantage of the use of agents from these different perspectives. We describe these perspectives of software agents and their relevance to the design of ubiquitous computing systems.

1.3.1 Software Engineering and the Agent-oriented approach

From the software engineering perspective, an agent is seen as a computer system situated in some environment and capable of flexible, autonomous action in that environment in order to meet its design objectives [Wooldridge, 1997]. The role of software engineering is to provide structures and techniques that make it easier to handle complexity. One approach for doing this is to adopt an agent-oriented approach which means decomposing a problem into multiple, autonomous software components that can act and interact in flexible ways to achieve their objectives. Autonomous agents are software components that offer greater flexibility and adaptability than traditional components [Griss and Pour, 2001]. In the rest of this document, the term “components” will be used to refer to the main building blocks that form part of a ubiquitous computing system.

1.3.2 Distributed systems and multi-agent systems

From the Distributed Systems stance, the technology of autonomous agents appears appropriate for building systems in which data, control, expertise, or resources are distributed; agents provide a natural metaphor for delivering system functionality [Wooldridge and Jennings, 1999]. Agents and multi-agent systems have been used as a metaphor to model complex distributed processes. While the area of multi-agent systems addresses distributed tasks, distributed systems support distributed information and processes. In short, multi-agent systems are often distributed systems, and distributed systems are platforms to support multi-agent systems.

One characteristic of ubicomp systems is that they should provide services and information whenever users need them. For this, implementing a ubicomp environment as a multi-agent system hides the fact that devices, services and information are disseminated all over the physical environment and makes it possible to create an environment with autonomous components that provide largely invisible support for tasks performed by users, for which the components may require to interact to achieve their objectives. However, in a ubicomp environment it is impossible to a priori know about all potential interactions that may occur at unpredictable times, for unpredictable reasons, between unpredictable components. As agents are components with the ability to initiate and respond to interactions in a flexible manner [Jennings, 2001], the agent oriented approach can be the natural way to deal with unpredictable associations and interaction among ubicomp systems' components.

1.3.3 Agents in Human-Computer Interaction

Finally, autonomous agents have been used to change the way people interact with computers which has been referred to as indirect management [Kay, 1990]. In this approach, the agent is a personal assistant that gradually becomes more effective as it learns about the user's interests [Maes, 1994]. Thus, from the Human-Computer Interaction (HCI) viewpoint, autonomous agents can be used to implement a complementary style of interaction [Maes, 1994]. Agents can also radically change the style of human-computer interaction and enhance collaboration in ubiquitous computing environments. The metaphor used is that of a personal assistant who is collaborating with the user in the same work environment. Autonomous agents can assist users in a variety of different ways: they hide the complexity of difficult tasks, they perform tasks on the user's behalf, they can train or teach the user, they help different users collaborate, and they monitor events and procedures. One of the major research directions for Human-Computer Interaction (HCI) has been exploring the novel forms of interaction that can achieve Mark Weiser's vision of naturally integrating computer technology with our daily activities. In this sense, autonomous agents can seamlessly assist users in their interactions with the ubicomp environment. For this, autonomous agents can learn of the users' interactions to infer their preferences and continuously be aware of their contexts in order to be responsiveness to their activities.

Many projects are under way to enable multi-agent systems on ubicomp environments, especially on mobile devices with tight memory constraints, limited computing power, and limited user interface peripherals. The section presents several projects that have integrated agents in ubiquitous computing systems and the problems they have tackled.

1.4 Agents as an alternative to deal with the challenges of developing ubicomp systems

There is a migration of technologies originally developed for PC's to the realm of handhelds and wireless networks, and agent technology is following this downsizing trend [Caire *et al.*, 2002]. Furthermore, agent technology is being considered of great help in the development of pervasive computing environments [Campo, 2002]. Thus, several research projects propose using multi-agent systems to cope with some of the challenges and difficulties of building pervasive environments, especially those that include mobile devices.

The rest of this section will present some of the projects that use an agent based paradigm to build pervasive environments and that deal with some of the complexities faced when building these environments.

Service/Device Discovery. Several projects, such as, Jini, Salutation, UPnP, SLP and Bluetooth [Golden, 2002] have focused on proposing protocols and technologies for discovering devices or services in ad-hoc networks. With service discovery technologies, devices may automatically discover network services including their properties; and services advertise their existence dynamically. Of the above projects, SLP [Bettstetter and Renner, 2000] uses agents to advertise the location and characteristics of services, on behalf of services; and agents to perform service discovery, on behalf of the client.

The PDP protocol uses a Service Discovery Agent to help other agents search for services offered by other agents or other systems in the network [Campo, 2002]. In each device of the environment resides a Service Discovery Agent, which uses the PDP protocol to maintain the list of the services offered in the environment. The difference between this project and the classical service discovery protocols previously mentioned is that it does not use a centralized server that listens for broadcast or multicast announcements of available services. Then, this solution adapts better to the case of spontaneous or ad-hoc networks. In the above mentioned protocols, agents were used to allocate the main functionality of a system that is inherently distributed. Thus, a ubicomp system using these protocols may have several agents requesting services and others registering services provided by the devices they represent.

Content adaptation based on context-awareness. To enhance the user's interaction with computational entities and other peers of a ubicomp environment, it is necessary to take into account the context of the interaction. Improving the computer's access to context increases the richness of communication in human-computer interaction and makes it possible to provide more useful computational services [Dey, 2001]. Context awareness becomes a key feature for ensuring an appropriate response by the application to the user's requests. Several works have focused on using the context in which the interaction takes place in order to adapt and personalize the presentation of information to the user, which is one of the challenges explained above. These projects use a multi-agent paradigm, where agents wrap functionality to obtain context information (such as, user's profile, preferences, activity, location and used device), which influence the way a service is accessed or service results are presented to the user [Carolis and Pizzutilo, 2002; Laukkanen *et al.*, 2002]. For this, these projects use a multi-agent system approach, in which agents at different layers need to coordinate their decisions to adapt information and services for the user.

Robustness. There are issues related to an application's robustness and security that have to be taken into account when trying to deploy ubicomp applications. Some of these issues are the disconnection of mobile devices, dynamic IP-address assignment, and the difficulties of pushing information to other wireless devices. Mobile user's disconnection limits the ubiquitous interaction with other entities (users, resources or services) of the environment. Several projects consider that a disconnection is not a failure event; rather it is a behavior that is natural to a mobile environment. Thus, these works provide methods that support, in different ways, the disconnection of mobile users. For instance, the CRUMPET project considers that agents are in charge of monitoring network connections and buffering the messages when they occur, or advertising the user if he is moving away from the coverage of the wireless network's [Laukkanen *et al.*, 2002]. Since mobile devices are more vulnerable and fragile than stationary ones, the ANTARCTICA system uses agent technology to manage and keep mobile user's data in a secure and safe space (called locker) in a network's fixed node. In this approach, agents are autonomous, in the sense that the data in the lockers are managed by agents working on behalf of the users, representing them in the network even while the users are actually disconnected [Villate *et al.*, 2002].

As stated above, agents have been used as a technological alternative to deal with some of the challenges in implementing ubicomp environments. Some of the above projects identified as autonomous agents the software elements that "act on behalf of" users or services. While others use the paradigm of agents to speed-up concurrent processing, and provide more reliability because of the lack of a single point of failure and improve the responsiveness of the system. However, they do not use autonomous agents as an abstraction tool for the design and construction of these systems.

1.5 Autonomous agents for developing Ambient Intelligence systems

Ubiquitous computing (ubicomp) environments are spaces where computational artifacts are invisible, become present whenever we need them, are adaptive to mobile users, can be enabled by simple and effortless interactions, and act autonomously to support users' activities and goals. Because of these features, ubicomp environments have also been referred to as smart spaces or Ambient Intelligence (AmI) environments.

The development of ambient intelligence systems demands an interdisciplinary approach, borrowing methods and techniques from several computing fields: 1) Ubiquitous Computing: which refers to a physical environment furnished with computational communication devices of all scales naturally integrated into human activity. 2) Context-aware Computing: to empower the system with technology that enables it to be responsive to users' activities and needs. 3) Human-Computer Interaction (HCI): to provide specific interfaces that are responsive to users needs in an unobtrusive and often invisible way, and finally, 4) Artificial Intelligence (AI): AmI systems need intelligent capabilities to be adaptive to users, reactive to context, and learn from user's behavior in order to provide high quality services based on their preferences. To provide these intelligent capabilities to ubicomp environments, we may use Artificial Intelligence (AI) techniques such as learning algorithms, pattern matchers and agent technology.

Ubicomp or AmI environments possess the characteristics of distribution, reactivity, collaboration and adaptation of their artifacts, thus sharing several characteristics with agents. These agents have autonomy in order to make their own decisions about what activities to perform, when to do them, what type of information should be communicated and to whom, and how to assimilate the received information. This provided motivation for this thesis to explore the use of autonomous agents as an abstraction tool for the design and implementation of ubiquitous computing systems. From this motivation, the following thesis objective was stated.

1.6 Research Objective

Design and develop a middleware to allow developers to manage some of the complexities associated with the development of ubicomp systems by means of the use of autonomous agents.

1.7 Research questions

Several research questions were the foundation of this thesis. The first and most important is related with the objective of the thesis: *Are autonomous agents appropriate constructs for developers to deal with the challenges associated to the development of ambient intelligence environments?* In answering this question, other issues came up:

The main actors of a ubicomp environment are mobile users that may need to be assisted opportunistically by the environment. Thus, the ubicomp infrastructure should provide an environment to enable mobile users to collaborate with other users, and access information and available services relevant for their activities whenever they need them. From this, other issues were*: How can autonomous agents enhance the activities of mobile users in a ubicomp environment? How can agent technology solve some of the problems associated with the opportunistic and spontaneous interactions of mobile users? How can autonomous agents enable users to opportunistically access the services available in the environment? What are the ubicomp systems' features that can be addressed by using autonomous agents? And, what are the challenges faced by developers when implementing these systems' features?*

To address the above questions it was necessary to identify settings in which the nature of the involved work is highly mobile such that information and services may need to be accessed from different users and locations and through different devices. This led to, as explained in the following section, the first part of this thesis in which the goal was to identify a setting where these characteristics were present and envision ubicomp scenarios such that ubiquitous computing technology combined with autonomous agents enhances the users' activities. This in turn led to the following research questions: *What are the ubicomp scenarios in which these characteristics are evident and relevant to the activities of users? Of the selected scenarios, what is the functionality that can be supported by autonomous agents?*

Once the way in which autonomous agents support users' activities in ubicomp settings was identified, it was necessary to answer the following questions: *What characteristics must an agent have in order to act autonomously on behalf of the user, and to represent services available in a ubicomp environment?*

Answering these questions permitted the identification of the requirements of an agent middleware for implementing ubicomp systems which in turn raised the following questions: *What are the minimum services that must be supported by a development framework in order to implement these kinds of ubicomp systems? What are the appropriate protocols of interaction and communication for these agents? Will this development framework allow programmers to design and build ubiquitous computing* *systems with little difficulty?* And finally, *what is the development process that a programmer must follow in order to build these kinds of systems?*

1.8 Methodology

To achieve the thesis objective, a methodology was followed to aid in identifying design scenarios that illustrate how ubiquitous computing technology enhances users' activities, and to explore the design of these systems by means of autonomous agents. From these system designs, the requirements for a middleware that facilitated the implementation of autonomous agents for ubiquitous computing systems were abstracted, and from them, an agent middleware was developed. The proposed methodology consisted of several iterative phases as illustrated in Figure 1. The first phase consisted of selecting scenarios that were then analyzed to identify how autonomous agents can be used for designing ubicomp systems. In the next phase the requirements of a middleware to support the development these ubicomp systems were identified. Based on these requirements the SALSA middleware was designed and implemented. Finally, the use of SALSA agents for designing and implementing ubicomp systems was evaluated.

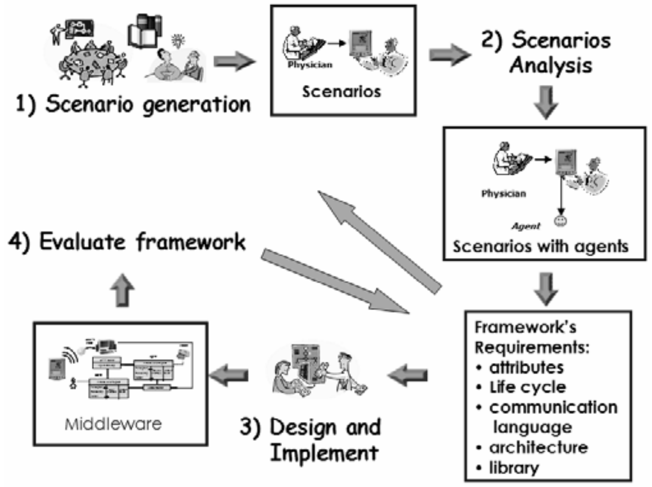


Figure 1. Methodology followed to create and evaluate the agent middleware for ubicomp systems

1.8.1 Scenario generation

Scenario building is a powerful mechanism to generate design ideas for new systems and to identify the possible uses and contexts of use of products by predicting how people could act in a particular situation. That is why it is well suited for the design of new product concepts and for the design of consumer products in the early phase of the development cycle, where the context of use is not well-defined [Carrol, 1995]. One benefit of scenarios is that they treat technology quite flexibly: they can either be described in detail or the focus can be more on the context of use. Under this approach typical and significant activities are explicitly envisioned and documented early and continuously in the development process of a system. The value of scenarios is that they make ideas more concrete for the purpose of analysis and communication. The concreteness enables designers and users to deal with complicated and rich situations and behaviors in meaningful terms, and to better understand the implications of particular design solutions for performing realistic tasks [Carrol, 1995].

This stage consisted of identifying scenarios with the purpose of discovering how ubiquitous computing systems can support the activities of users in their working environment. As a result of this stage, the scenarios were described in textual or graphic form. The scenarios briefly sketched user’s activities without committing to details of precisely how the tasks are carried out or how the system enables the functionality for those tasks [Carrol, 2000].

1.8.2 Analysis of scenarios and applications

The selected scenarios were analyzed with the purpose of discovering how autonomous agents can enhance the activities of users. Then, the scenarios were transcribed in order to express the situations in which it is relevant for an autonomous agent to act on behalf of a user and/or allow the opportunistic interaction with a device or a service. The scenarios were constructed in terms of hardware and software components that could implement the envisioned functionality in order to provide a system view. Thus, representing the use of a system or application with a set of interaction scenarios makes their use explicit and meaningful to people using the system to achieve real goals [Carrol, 2000]. Thus, during this stage it was possible to explore the way in which autonomous agents can be used to design AmI systems, in addition to how they allow developers to integrate new functionality into a system. Finally, this analysis phase enabled us to abstract the requirements for the middleware from a desired set of scenarios.

1.8.3 Design and implementation of an agent middleware for ubicomp systems

From these application scenarios, the initial requirements related to the behavior of the agents and their communication protocol were identified. To support these requirements the following core features of the agent-based ubicomp middleware were designed and implemented:

- The agents' components for perceiving, reasoning and acting.
- The agents' life cycle which was defined based on the agents components and the agents behavior identified from the scenarios.
- The communication platform, which should enable the collaboration among agents, users and services/devices.
- A library of abstract classes that provides the methods to create the components of autonomous agents and control their life cycle.
- A service that enables agents to register and search information regarding available agents with whom they need to interact in order to achieve their goals.
- Finally, it was possible to identify the attributes that an agent living in a ubiquitous computing environment must have in order to support the functionality sketched in the scenarios.

1.8.4 Evaluation

The aim of this evaluation phase was to assess whether the thesis objective was reached. In order to do this, the following properties were identified as being relevant and desirable for evaluating in a middleware:

- *Utility.* This criterion determines whether others can build real pervasive applications atop the infrastructure.

- *Completeness.* This criterion determines whether the architecture is sufficiently powerful and extensible to support interesting user-space programs.
- *Ease of use.* Development frameworks should be evaluated on how readable programs using the programming language are by other programmers, how learnable it is, how convenient it is for expressing certain algorithms, and how comprehensible it is to novice users.

These desirable features were used to state the hypothesis that permitted an evaluation of how flexible SALSA agents are in enabling the progressive development of ubicomp systems.

1.9 Thesis contributions

The following are the main contributions of the thesis:

- *A set of realistic scenarios of ubiquitous computing systems.* A set of selected scenarios that represent real users' activities in a hospital setting. They illustrate how users' activities are enhanced by using ubiquitous computing and agent technologies.
- *An analysis of the complexities that can be addressed by using autonomous agents.* During the thesis, the complexities associated to the development of ubicomp that can be addressed thru the use of autonomous agents were identified, and this was illustrated by the scenarios of use of ubicomp systems, and their design and implementation.
- *The design issues regarding autonomous agents for developing ubiquitous computing systems.* These design issues describe the characteristics that autonomous agents must have in order to provide the functionality needed by ubiquitous computing systems. They were the foundation from which the requirements of a middleware for implementing ubicomp systems were identified.
- *An agent middleware, named SALSA, for developing ubiquitous computing systems.* This middleware had the goal of facilitating the building and evolution of ubicomp systems by means of autonomous agents. SALSA provides an appropriate agent communication language for enabling agents to convey information in a ubicomp system; and defines an agent life cycle with different states through which agents

perceive information and decide how to act. The usefulness of SALSA was illustrated by describing how ubiquitous computing systems are developed with this middleware, and the facilities provided by SALSA for creating autonomous agents was evaluated by a group of developers.

- *Use of SALSA as a research test-bed*. SALSA was used for building ubiquitous computing systems in which autonomous agents were used as its main abstraction. The creation of other applications enables the identification of the strengths and weaknesses of SALSA agents.

1.10 Thesis outline

CHAPTER 2 *(Background and related work)* introduces the field of ubiquitous computing, and then explains the complexities that developers have to address when implementing ubicomp systems. Then, this chapter briefly introduces agent theory and explains why agents are attractive for facilitating the development of ubiquitous computing systems. Finally, it presents an analysis of the facilities provided by existing development platforms for ubiquitous computing from which it is concluded that these approaches do not address the progressive evolution of ubiquitous computing systems.

CHAPTER 3 *(Scenarios envisioned for the healthcare)* explains why the healthcare domain was choose for analyzing the activities of users and propose a ubicomp system to enhance them. It presents related research in which agents were used to cope with some of the complexities of ubiquitous computing systems development. And finally, it presents the scenarios envisioned from observing users in a healthcare domain. These scenarios were analyzed to find design issues which were addressed by using autonomous agents. Thus, this chapter illustrates how autonomous agents can be used as design abstractions for ubiquitous computing systems.

CHAPTER 4 *(The SALSA development framework)* presents the functional requirements of the SALSA agent middleware which were identified from the scenarios described in CHAPTER 3. Then, it presents the design of the architecture of SALSA and finally, the API and services provided by SALSA to facilitate the development of ubicomp systems.

CHAPTER 5 (*Creation and evolution of ubicomp systems with SALSA)* illustrates the implementation of an autonomous agent and its integration into a ubicomp system, which demonstrates how SALSA can easily enable the integration of a new functionality into a pervasive system.

CHAPTER 6 *(Evaluation)* describes the experiments carried out for evaluating a set of features of the middleware, and also presents the results of such experiments.

CHAPTER 7 *(Conclusions)* presents the conclusions and reflections of this work and new research questions rose by this work.

Chapter 2

Background and related work

Beyond the era of personal computing, the era of ubiquitous computing begins with the vision of decentralizing computing power: The computer is omnipresent becoming a part of everyday life and an inevitable component when performing a variety of private and business related tasks [Hansmann *et al.*, 2001]. Devices of many forms and sizes enable users to exchange and retrieve information they need quickly, efficiently, and effortlessly from everywhere at any time [Hansmann *et al.*, 2001]. This was the vision of Mark Weiser in 1991 which led to diverse unresolved issues that must be addressed before ubiquitous computing truly reaches its goal of improving our everyday lives. For coping with some of these unresolved issues or challenges faced in creating ubicomp environments, several research works have proposed development platforms, such as frameworks, toolkits or middlewares to facilitate the implementation of ubicomp systems. Before presenting some of these development platforms, this section first introduces the ubiquitous computing area, then it exposes the challenges for implementing ubicomp environments and finally, it presents the software systems that facilitate the development of these environments by providing mechanisms that address some of the ubicomp challenges.

2.1 Ubiquitous computing

Ubiquitous computing involves a new way of thinking about computers, one that takes into account the human world and allows computers themselves to vanish into the background. The vision of a future ubiquitous computing landscape is dominated by the pervasiveness of a vast manifold of heterogeneous computing devices, the autonomy of their programmed behavior, the dynamicity and context-awareness of services and applications they offer, the ad-hoc interoperability of services and the different modes of user interaction upon those services [Fersha, 2002]. Today a variety of terms –like Ubiquitous Computing (ubicomp), Pervasive Computing, Calm Computing, Invisible Computing, Ambient Intelligence (AmI), Sentient Computing and Post-PC Computing –

refer to new paradigms for interaction among users and mobile and embedded computing devices [Banavar, 2000]. Thus, ubiquitous computing is the attempt to modify the traditional human-computer interaction paradigm not only by distributing computers, of all scales, into the environment surrounding users, but by augmenting work practices, knowledge sharing, and communication of users. For this, computers should be able to use implicit situational information, or context, to provide useful services and relevant information whenever users need them [Dey, 2001]. Context-aware computing refers to an application's ability to adapt to changing circumstances and respond based on the context of use. A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task [Dey, 2001]. Dey defined context as *"any information that can be used to characterize the situation of an entity, which can be a person, place, or object that is considered relevant between a user and the application, including the user and application themselves"[Dey, 2001]*. Among the main types of contextual information considered relevant are identity, time, activity, and location which are known as primary context [Dey, 2001; Schilit and Theimer, 1994]. This information answers the questions of who, when, what, and where, which can be used to identify whether a specific piece of information is relevant to establish context. Thus, the environment should be aware of the user's context to provide information and services whenever users need them, in a proactive fashion and anticipating user's needs. Furthermore, the services provided by the environment have to be accessible to diverse and non-specialist users through simple and effortless interactions. For this, human computer interaction promises to support more sophisticated and natural input and output, to enable users to perform potentially complex tasks more quickly, with greater accuracy, and to improve user satisfaction.

From the previous explanation is can be stated that ubiquitous computing environments are characterized by:

1) the *distribution* of their devices and services,
2) the high *mobility* of users

3) the need to *opportunistically access* information, services and devices available in the environment;
4) and finally, the *implicit and natural interactions* of users with the ubicomp environment.

Ubiquitous computing environments require software components that enable the above characteristics. These software components need to seamlessly establish connections and *communicate* among themselves in a transparent way to provide the services required by the users; the software components should be *reactive* to the environment and the users' context in order to enable them to opportunistically access pervasive computing resources. For this, the components need to be perceptive to context changes which are highly dynamic due the mobility of users; through *adaptive* components the environment can learn from the user's interaction in order to adapt its behavior to the users' needs; and finally the components need to act *autonomously* freeing users to explicitly access the ubiquitous computing resources and decide how to act in order to enhance the users' activities. Thus ubiquitous computing environments are characterized by the *collaboration, reactivity, adaptability and autonomy* of their software components and in this sense, they share many characteristics with agents. For this reason, this thesis proposes using software agents as the constructs to deal with the challenges in realizing the ubiquitous computing vision. The following sections explain what challenges were identified for developing ubicomp systems and present existing development platforms that facilitate programmers' work as they face some of these challenges.

2.2 The challenge of developing ubiquitous computing systems

The research community has identified research challenges for designing and implementing ubicomp systems.

In [Banavar and Bernstein, 2002] it is stated that three characteristics have to be addressed when implementing a ubicomp system: task dynamism, device heterogeneity and resource constraints, and the impact of ubicomp in a social environment. From them, several research challenges for coping with these characteristics were identified and organized into four broad categories: 1) *Semantic modeling.* In order for the computing environment to be adaptable and easy to compose, high-level semantic models are needed to represent the users' preferences and the relevant characteristics of computing components. 2) *Building the software infrastructure*. A ubicomp infrastructure must be capable of finding, adapting and delivering the appropriate applications to the user's computing environment based on his context. 3) *Developing and configuring applications*. Configuring services and applications so they can be easily and reliably reused by other developers and composed into larger applications will be a major challenge. 4) *Validating the user experience*. Effective methods are needed for testing and evaluating the usage scenarios enabled by pervasive applications.

Mark Weiser expounded that one important feature of ubicomp environments is the physical integration and spontaneous interoperation of devices that arrive and leave routinely. In order to address these features, Kindberg and Fox identified several challenges that were classified in the following areas: *discovery* (of devices/services), *adaptation* (presenting data and interfaces in heterogeneous devices), *integration* (of the computing environment with the physical world), *robustness* (managing of failures, i.e. going out of network range), *security* (user authentication, privacy, trust), and *programming frameworks* (that facilitate building ubicomp systems) [Kindberg and Fox, 2002].

After analyzing the experiences and problems faced in implementing manifold ubicomp prototypes, Davies and Gellersen affirmed that deploying systems beyond the limited existing prototypes will require significant progress toward integration. This poses major research challenges that must be overcome in supporting the requirements that integration places on components that form a ubiquitous computing system. Some of these challenges are: *component interaction* (components should be designed and implemented in an open and extensible manner), *adaptation and contextual sensitivity* (components adapt internally), *appropriate management mechanisms and policies* (zero or low configuration of components, and support for rapid reconfiguration), *components association and task analysis* (system's ability to develop associations between components to assist the user in his activities), *user interface integration* (as the number of applications increases, the ubicomp environment should ensure the provision of a reasonable user interface), *social, legal and technical solutions to privacy and security concerns* (designers have to empower users to evaluate the tradeoff between protection of privacy and access to services) [Davies and Gellersen, 2002].

Even when the above challenges where identified from different research works, they coincided on various aspects: For instance, all of them agree that providing support for adaptation (of applications, system's components, data or interfaces) is a major issue. The most important similarity among the previous works is that all of them, implicitly or explicitly, present as a challenge providing facilities for developing ubicomp systems, such as programming frameworks as exposed in [Kindberg and Fox, 2002]; while in [Banavar and Bernstein, 2002] the emphasis is on the importance of providing support in order for applications to be easily reused by other developers and composed into larger applications; and in similar way, Davies and Gellersen emphasizes that ubicomp system's components should be implemented in an open and extensible manner to enable their reuse and the evolution of systems [Davies and Gellersen, 2002].

2.2.1 Complexities of ubiquitous computing systems

Providing a development platform that eases the development of ubiquitous computing systems is a challenge because it has to address other challenges faced in developing ubicomp systems. From the challenges previously presented and the analysis of different development platforms (presented in Section 2.3) the complexities that these development platforms tackle by providing facilities for creating a ubicomp system were identified, as well as the challenges faced in addressing these complexities without using a development platform. What I obtained is the following list of complexities of ubicomp systems and an explanation of the challenges faced in addressing them.

a) *Robustness and routine failures.* In a ubiquitous system we can see an increase in the frequency of failures related to unpredictable events that happen in this type of systems. For instance: a ubicomp system uses wired and wireless networks in which unpredictable disconnections of mobile devices may happen. Coping with disconnections of mobile devices has become a major challenge for developers of mobile and ubiquitous computing systems. The heterogeneity of the devices that compose ubicomp systems may also generate other “failures”. For instance, a user may need to access the same information from a public display and later from his PDA which can be considered an unpredictable event if the system does not provide the mechanisms to adapt the information to be presented in different devices. Heterogeneity and adaptation are other complexities of ubicomp systems, and dealing with them presents manifold challenges for developers.

b) *Heterogeneity.* In a ubicomp system we find devices of many scales, forms and with different computing and communication capabilities. This is the major complexity of a ubicomp system since it presents many challenges related with associating different devices for providing a service to the user, and adapting information in any of the computing devices as required by the user.

c) *Adaptation.* Adaptation in ubicomp is quantitatively hard. Instead of adapting a small fixed set of content types to a variety of devices types (1-to-n), we must potentially adapt content among n heterogeneous device types (n-to-n) [Kindberg and Fox, 2002]. A new challenge for ubiquitous systems arises because adaptation must often take place without human intervention, to achieve what Weiser calls calm computing [Weiser and Brown, 1996]. Thus, adaptation not only involves displaying information, interfaces and services according to many devices with different computing capabilities, but taking into account the user’s needs, and what his preferences are. Then, the challenge is applying adaptation by using mechanisms invisible to the user. For this, a ubicomp system needs capabilities for taking into account the user’s context.

d) *Dealing with context information.* Enhancing the human-computer interaction is a major issue in ubicomp [Dourish, 2004]; and providing information and services based on the user’s context is a way to contribute to the achievement of this goal. However, dealing with context information is a complex problem that has deserved enough attention to give birth to a new research area: Context-aware Computing. The aim of this area is to provide the technology that will enable ubiquitous computing systems to be sensitive and responsive to their setting.

Context is difficult to use for several reasons [Dey, 2000]. First, capturing primary context information requires the use of sensors and computing devices. Context must be abstracted to make sense to the application, for instance, the ID of a mobile user must be abstracted into the user's name or role. Finally, context is dynamic, i.e., a mobile tour guide must update its display as the user moves, which require tracking the user's location by gathering information from multiple sensors, and using techniques that estimate the user's location or guess the route that a user will follow, which may introduce uncertainty. Thus, in order to provide context-aware services, various issues and challenges are faced in order to cope with the gathering, interpretation, representation and dissemination of context information within dynamic and frequently changing computing environments.

e) *Discovery of services/devices and interaction.* In a ubicomp environment, new devices may unpredictably join the environment. The system should provide the mechanisms needed in order for this to happen with low or no configuration from the user. As a new device or service joins the environment, it must be visible or available for the other system's components and vice versa. However, several implications related with the discovery and interaction of devices/services are involved. For instance: what are the policies for allowing new devices to interact with the ubicomp environment? What are the appropriate protocols for mutual discovery of devices in ubiquitous computing, since many of the devices may have different computing and communication capabilities? Answering these questions poses significant challenges for ubiquitous computing.

f) *Scalability*. Weiser defined the notion of scale as incorporating a broad space of computational devices [Weiser, 1991]. However, other issues of scale are implicit in the definition of ubicomp: The ubicomp paradigm reinforces the desire to break the human away from desktop-bound interaction, which implies users may collaborate with other users or interact with the ubicomp environment from any other device. This opens up opportunities for other users not attached to a desktop to join the environment, which implies scaling with respect to people. A final dimension, time, presents new challenges for scaling a system. Pushing the availability of interaction to a "24/7" basis uncovers another class of unexplored interactions. To address scaling with respect to time, the term "everyday computing" (synonymous with ubiquitous computing) was introduced to refer to informal and unstructured activities that are continuous in time, since they do not have a clear starting or ending point [Abowd and Mynatt, 2000].

2.3 Development platforms for ubiquitous computing systems

As one of the objectives of this thesis is to propose a middleware that facilitates the development of ubicomp systems, this section presents a review of several development platforms that deal in different ways with one or more of the ubiquitous computing complexities previously presented. These development platforms were categorized as follows: 1) those that emphasize the support provided for creating applications for supporting the mobility of users (mobile computing) and their collaboration with other users or accessing of services available in the ubicomp environment; 2) those that support the implementation of context-aware features of the ubicomp system; 3) and finally, platforms that provide some support for creating ubiquitous computing systems, which may involve support for mobile computing and context-aware computing. These are middlewares that not only provide a development framework or API, but an infrastructure on which a ubicomp system is executed.

2.3.1 Software architectures for mobile computing

Several of the middlewares for creating ubicomp environments, focus on facilitating the development of applications for mobile devices that need to interact with other devices of the ubicomp environment.

2.3.1.1 YCab

YCab is a lightweight and flexible API that developers use to rapidly create small and highly optimized applications that allow mobile users to collaborate in ad-hoc wireless networks [Buzko *et al.*, 2001]. This framework focuses on providing a fault-tolerant environment by allowing collaborators to float in and out of an ad-hoc network without causing disruption to the collaborative session. Thus, the collaborative system is entirely decentralized with all the clients running exactly the same software. Users can use the YCab application provided with the API, or they can implement their own application using the services provided to fully customize the application for the target device. The YCab framework provides default implementations for common “services”, such as, state recovery (restoring the state information for a given service), optimization of the network resources (such as, high and low bandwidth) or service initialization. Developers can specify which services are to be included in the application, and how they are to be arranged. In addition, they can enable or disable certain features of those services without the need to redesign each service. Thus, the YCab framework allows implementing modular collaborative applications that do not require extensive programming expertise to design. Since the applications that can be developed with YCab are specifically for mobile devices of ad-hoc networks, it only allows interactions between mobile users, restricting the user’s opportunity of interacting spontaneously with any other device that could offer a relevant service that may enhance the collaborative activities of the user. Furthermore, this framework is not suitable for building applications for pervasive environments, where heterogeneous computing devices can arrive and leave routinely.

2.3.1.2 DACIA

DACIA is a development framework that provides mechanisms for building groupware applications that adapt to available resources and support user mobility [Litiu and Parkash, 2000]. Using DACIA, components of a groupware application can be moved to different hosts during execution, while maintaining connectivity with groupware services and other users. A mobile application can be parked while its user is disconnected or idle. A parked application, also called client agent, can continue to interact, with some limitations, with other parties on behalf of the user. It can reside on the same computing device the user had been connected from, or it can move to a fixed host if the user’s device is disconnected. When the user reconnects, eventually from a different place, he can take over control from the parked application. The temporary failure of a connection between two components of the application is made transparent. When a network connection is broken, the messages are cached until the connection is re-established, assuming that the disconnection is temporary. However, the support that DACIA provides for intermittent connectivity is weak in the sense that it does not allow the client agent to continue interacting on behalf of the user when the disconnection was produced due to a failure on the network. The activities that a client agent can carry out while the user is disconnected are too specific and simple, such as, save messages; inform other parties that the user is not active or forward notifications. This agent neither acts on its own nor make decisions that enhance collaboration if the user is inactive for a long period of time.

2.3.1.3 MIDAS-PROSE

In [Popovici *et al.*, 2003] a platform is presented to enable mobile computing devices to dynamically extend or modify the functionality of an application. The idea is to let the environment proactively adapt the application rather than forcing the application to adapt itself to every possible environment. Thus, this kind of adaptation is not based on sensing the environment in order to adapt its behavior to the current context. Through this platform, mobile devices acquire at execution time any functionality extension they may need to work properly in a given environment, or they can provide extensions to other devices. For this, the platform uses Jini technology to advertise to a base-station the presence of the services that need to be adapted in a mobile device. The platform provides a layer of security to assure that the extension comes from a trusted party and that the extension does not access non-authorized resources. By using this kind of adaptation, users of the application in the mobile device should explicitly select the functionality extension available in the environment. This is a different approach to adaptation from the way in which it is usually addressed in ubiquitous computing, in which the devices of the environment are intelligent enough to adapt themselves to user's context in order to enhance his interaction with the ubicomp environment.

2.3.1.4 CARISMA

CARISMA is a mobile computing middleware that exploits the principle of reflection to enhance the construction of mobile applications that have to adapt to changes in context, such as variations in network bandwidth, memory or battery power [Capra *et al.*, 2003]. The model followed by CARISMA, assumes that the behavior of the middleware with respect to a particular service is determined, at any time, by only one policy; that is, a service cannot be delivered using a combination of different policies. These policies are specified by means of application profiles which can be dynamically changed through a reflective API. Applications, however, may not be smart enough to cope with these changes, which may lead to conflicts. A conflict may exist when different policies are used in the same context to deliver a service, so that the middleware does not know which one to apply. When a conflict occurs, a resolution mechanism is run to solve the conflict and find out which policy to use to deliver the service. The conflict resolution mechanism is based on microeconomic techniques, in which the middleware plays the role of an auctioneer; and the applications are agents competing for goods, which are the execution of the policies they value most among a set of alternatives that correspond to the policies that can be applied in a particular context to deliver a service. The aim of the middleware is to select the policy that satisfies the largest number of applications involved in the conflict. In this way, this middleware focuses on providing a protocol to enable mobile devices to negotiate for services available in the environment by adapting the access policies to these services. Similar to the middleware previously presented, the adaptation provided by CARISMA is not based on the context or preferences of users.

2.3.2 Architectures for context-aware systems

The above mentioned middlewares are projects that provide a high level development API and services to allow the building of collaborative applications for mobile users, or the adaptation of the functionality of mobile devices to interact with the ubiquitous computing environment. However, these architectures do not take into account the users' context to enable the provision of adapted services or for enhancing the collaboration among users. To address these issues, other middlewares focus on supporting the challenges associated with the provision of context-aware services, such as, the gathering, representation and dissemination of context information. As explained in Section 2.1, context-aware technology is regarded as a core technology of ubiquitous computing. The following sections present several middlewares that provide support to facilitate the implementation of different aspects of context-aware systems, such as: extracting context information of the environment, transforming this information to a language understood by the environment's components, reasoning about the perceived context, and automatically executing or adapting services.

2.3.2.1 RCSM

RCSM is a middleware that provides development and runtime support to allow applications to exploit their context sensitivity to communicate with other devices [Yau *et al.*, 2002]. RCSM was designed to facilitate the development of applications which require context aware ad hoc communication. By using RCSM, a context-sensitive application is modeled as context-sensitive objects, which consist of two parts: an interface that encapsulates the application’s context sensitivity (which include the list of relevant context elements, actions and mapping between both of them), and the implementation of the actions that the application software must provide. Thus, the developer focuses on implementing the actions, in any language, without worrying about context monitoring, detection, and analysis. To support spontaneous communications between remote objects, RCSM uses a decentralized communication model, and a message-oriented (CORBA, ORB-oriented) approach, which allows the middleware to discover new devices and functionalities and establish new communications links.

2.3.2.2 Semantic Space

Semantic Space is a pervasive computing infrastructure that exploits Semantic Web technologies to support explicit representation, expressive querying, and flexible reasoning of context in smart spaces [Wang *et al.*, 2004]. An ontology is proposed to represent context information that characterizes an environment. This context information was identified as three classes of objects (user, location, computing entity) and one class of conceptual object (activity) from which a higher-level context can be inferred. The infrastructure of Semantic Space provides mechanisms to let applications retrieve context information using declarative queries, infer higher-level contexts by using heuristic rules, enable devices to dynamically join the environment, and monitor these devices in order to abstract context information from them.

2.3.2.3 HIML

In [Kim *et al.*, 2004] a context-awareness middleware is presented, which utilizes a standardized context scheme called Human Interaction Markup Language (HIML) to manipulate context information. It was created with the aim of providing a common language for expressing context-awareness for any kind of device, and for facilitating the user's interaction with the context aware system by modifying his current contextual information. This middleware enables the components of the ubicomp system to describe and construct context messages and store them in a database for later use. The middleware lets a user explicitly give or adjust contextual parameters through his terminal, thus new context information is generated from the terminal and formed into HIML format in order to be stored in a context database. The accumulated context knowledge is fused and analyzed through data mining to form new contexts that can be used to activate the operation of terminal devices.

2.3.2.4 CAMUS

Context-Aware Middleware for Ubiquitous computing Systems (CAMUS) is a middleware that focuses on providing context composition and an efficient separation of concerns between different sensing techniques and context formation processes [Hgo *et al.*, 2004]. The data extracted from sensors, called features, are represented as a Feature Tuple Space (FTS), which is the communication and storage mechanism used by the middleware. Tuples are mapped to convert a given feature into context, which is saved in an ontology repository. The ontology contains the domain concepts and properties with formal semantics which enables the categorization of context entities into agents, devices, environment, location and time. Several reasoning mechanisms can be incorporated in CAMUS as pluggable services. For instance, fuzzy logic and Bayesian networks can be used to produce composite context which is derived from context information gathered from multiple sensors.

2.3.3 Middleware for pervasive environments

This section presents software platforms that provide programming support for addressing ubicomp challenges related with the operating system or middleware over which a ubicomp environment is built. These platforms not only provide an API for facilitating the implementation of ubicomp systems, but a set of services to cope with some of the complexities for implementing ubicomp systems. Table III briefly presents the features of these middleware.

2.3.3.1 Gaia OS

The ActiveSpaces project developed the Gaia meta-operating system, which is a distributed middleware infrastructure that coordinates software entities and heterogeneous networked devices contained in a physical space [Román *et al.*, 2002]. This middleware abstracts a space and all the resources it contains as a single programmable entity, called active space. Thus, Gaia OS supports the development and execution of portable applications for active spaces. Its OS kernel provides a set of basic services to ubicomp applications, such as, managing events or changes that occur in the active space, letting applications register and query for particular context information, or managing users' data automatically. Gaia also provides an application framework, which lets users construct, run, or adapt existing applications to active spaces, and includes an infrastructure to add context-awareness features to systems [Ranganathan and Campbell, 2003]. It involved various agents, each one with a defined functionality, such as providing context to other agents, or deducing high-level context. A key feature of this infrastructure is that it endows agents with a variety of reasoning or learning mechanisms to help them reason about context appropriately, for instance, one of the agents (User Mood Context Provider) uses the Na Bayes algorithm for predicting user mood.

2.3.3.2 One.world

The aim of this architecture is to enable the development of adaptable pervasive applications [Grimm, 2004]. To create this architecture, three requirements were identified: System support must embrace contextual change of mobile users either carrying their own portable devices or switching between devices, encourage a dynamic computing environment by enabling ad-hoc composition, and allow user collaboration by facilitating sharing between applications and between devices. Among the facilities supported by one.world is the discovery of resources. For this, one device acts as the discovery server which provides information (a descriptor and an event handler) of the available resources. To advertise services, individual devices have to export their discoverable resources to all visible servers. However, this mechanism of service discovery does not facilitate the specialized look up, such as searching for a service with specific features, or searching for the services within a sub-area of the physical environment. Related with the communication among components, one.world expresses all communication, through asynchronous events. Developers have to create their own event handlers according to the information that want to be communicated. Under this approach, the quantity of data to be communicated may be limited, since the focus of the event model is just to notify to listener objects of changes in their runtime context. To simplify the sharing of data, one.world represents all data as tuples, which are expressed in the form of code. This limits the interaction of one.world components with other platforms. Finally, one.world provides the ability to migrate an environment, including all execution state and all stored tuples to a remote node, leaving no implicit back-references to the originating node.

2.4 Discussion

The development platforms presented in this Chapter provide support for addressing some of the complexities associated to the development of ubiquitous computing systems. Table I, II and III summarize how these development platforms cope in different ways with the complexities identified in Section 2.2.1. The first row of these tables contains the name of the development platforms and the type of application they support in parentheses; the remaining rows give a brief explanation of how they tackle some of the complexities for implementing ubicomp systems. The last row presents the programming techniques used by the development platform for dealing with some of the complexities of ubicomp systems. Among these techniques are software programming paradigms and languages (i.e. Aspect Oriented Programming), models for representing data (i.e. ontologies) and mechanisms to handle the system’s behavior (i.e. reflection) [Capra *et al.*, 2003; Popovici *et al.*, 2003]. For instance, some of the development platforms propose using different implementation techniques for providing a common language to represent and manipulate context information (ontologies, or a new markup language such as HIML as described in Table I.

Others middlewares propose using different programming approaches, such as aspect oriented programming or reflection, to enable the adaptation of mobile computing devices in a ubicomp environment (see Table II).

Due to the characteristics of software agents, explained in Section 2.1, this thesis explores the use of software agents as the key computer-based components of a ubiquitous computing system. Agents can be software elements with *autonomy* that make their own decisions about what activities to perform, when to do them, what type of information should be communicated and to whom, and how to assimilate the received information; with *reactivity* for perceiving the context of the users and acting based on it; with a communication language that enables agents to convey different kinds of objects needed for enabling not only the interoperability or *collaboration* among the software components of the environment, but also among the components with users and devices; finally the *adaptability* attribute of agents can be used for enabling the components to adapt their services to users’ needs. Under this approach, autonomous agents are the software entities that can represent the devices and services available in the environment, act on behalf of users, or wrap a system functionality with which the user seamlessly interacts.

Table I. Platforms for supporting mobile computing in ubicomp environments.

DEVELOPMENT PLATFORMS **CHALLENGES**	**YCab** (Collaborative applications)	**DACIA** (Collaborative applications)	**MIDAS/PROSE** (Proactive adaptation of applications)	**CARISMA** (Adaptive context-aware applications)
Robustness and routine failures	Disconnections: State recovery, Decentralized	Disconnections: An agent represents the user	Secure access to resources/services of the environment	Resolution of conflicts through policies to determine devices interaction
Deal with context information	N/A	N/A	N/A	Application profiles specify a set of conditions to use a service
Adaptation	N/A	N/A	Adaptation of application functionality	Adaptation of the policies to apply for using a service
Discovery of services/devices and interaction	N/A	N/A	Advertisement of the available services (Jini)	N/A
Heterogeneity	Mobile devices	Mobile devices	Mobile devices (PDAs, laptops, lego-robots) and Servers	Mobile devices (lap-tops)
Scalability	N/A	N/A	A mobile device can adapt itself to a wide range of settings	A scalability limit given by: Profiles with 10 policies, each with 5 or more associated contexts, and 10 or more resources for each of these contexts By parallelizing the auction protocol, it was feasible to increment the number of devices
Implementation techniques	N/A	N/A	Aspect Oriented Programming	Reflection

N/A = Not Available

Table II. Architectures for creating context-aware systems.

DEVELOPMENT PLATFORMS **CHALLENGES**	**RCSM** **(**Context aware ad-hoc applications)	**Semantic Space** (Context-aware applications)	**HIML** (Access and sharing of Context information)	**CAMUS** (Context composition)
Robustness and routine failures	N/A	N/A	N/A	N/A
Deal with context information	Analyses of context data and trigger actions	Representation, querying, reasoning of context information, and inferring of high-level context	Representation, and generation of new context information Users have direct access to context information	Unification interface for abstracting data of sensors Reasoning modules to produce composite context
Adaptation	N/A	N/A	Present context information in several formats (WML, HTML, XHTML)	N/A
Discovery of services/devices and interaction	Publish/subscribe RCSM objects	Devices dynamically join to the environment (UPnP)	N/A	N/A
Heterogeneity	Heterogeneous devices	Any device, such as sensors, mobile devices, PCs.	Terminals (PCs and mobile devices)	Any sensor or device that provide context information
Scalability	N/A	N/A	N/A	N/A
Implementation **Techniques**	N/A	Semantic Web technologies, Ontology	Markup language	Tuples, Ontology

N/A = Not Available

One of the features of autonomous agents is that they are appropriate abstractions for implementing loosely-coupled distributed systems since agents have an asynchronous communication language.

One of the challenges of providing a middleware for ubicomp is to facilitate the creation and evolution of a ubicomp system. However, the middlewares analyzed do not address how they facilitate the evolution of a pervasive system. Through autonomous agents a ubicomp system can be created and extended if they use appropriate communication mechanisms that enable them to seamlessly interact.

This thesis proposes a middleware that facilitates the implementation of autonomous agents which are the main components of a ubiquitous computing system with the above characteristics. Before presenting this middleware, the following Chapter introduces how agent technology has been used for implementing ubiquitous computing systems.

Table III. Middlewares for ubicomp systems.

DEVELOPMENT PLATFORMS **CHALLENGES**	**Gaia** (Pervasive application)	**One.world** **(**Pervasive applications)
Robustness and routine failures	Error detection of software and hardware	Resumes an application after a failure
Deal with context information	Register and query for particular context information	N/A
Adaptation	N/A	N/A
Discovery of services/devices and interaction	N/A	Automatic registration and proactive discovery of services Communication trough asynchronous events and share data as tuples
Heterogeneity	Heterogeneous devices	Heterogeneous devices
Scalability	N/A	N/A
Implementation approach	It includes agents for the context-awareness infrastructure	Tuples for data, and events for communication

N/A = Not Available

Chapter 3

Autonomous agents for designing ubiquitous computing systems for hospitals

For analyzing how autonomous agents can enable developers to deal with the complexities of ubiquitous computing systems, we decided to study the medical activities in Hospitals, which are settings characterized by the need for coordination and collaboration among specialists with different areas of expertise, an intense information exchange, and the mobility of hospital staff, patients, documents and equipment.

3.1 Characteristics of hospitals settings

Information management and communication in a hospital setting is characterized by a high degree of collaborative work, mobility, and the integration of data from many devices or artifacts [Reddy and Dourish, 2002]. Exchanges of information are intense, and demands from participants to promptly extract from the artifact useful pieces of data to perform their job. In contrast with other settings such a control rooms [Theureau, ans Filippi, 2000], information in hospitals is not generally concentrated in a single place but distributed among a collection of artifacts in different locations. For instance, patients’ records are maintained and used in coordination with data on whiteboards, computers, or binders located in rooms, labs, common areas or offices. Following Bossen we might say that for practical purposes the whole hospital becomes the information space and it is by "navigating" this space that hospital's staff can get the data to perform effectively [Bossen 2002].

Given the high distribution of information together with the intensive nature of the work, it results clear that tremendous coordination efforts are required from all members of the hospital staff to properly manage the information to attend and take care of patients. The right information has to be in the right place, whenever it is needed by whoever needs it, in whatever format (representation) that they need it. Hence the characteristics of artifacts containing information play a fundamental role to achieve this coordination. For instance, patient’s records are easily moved from place to place and filled, checked, read and consulted in many locations like nurses' room, analysis labs, or the actual bed where the patient is being attended; nurses, physicians and other workers interact with those records and use them to support their work or to transmit instructions to be followed by others. To have the patient's records at the right place is what in part makes them successful to support coordination, as well as the fact that the information contained in them is clear, complete, accurate, and updated. Unfortunately those conditions are not always achieved. Documents get lost, instructions are not clear, or the data is not complete to support decisions. We therefore need to understand how each information item gets integrated successfully into artifacts to support the coordination required in hospitals. From this understanding we can derive adequate information technology. To provide adequate support for managing information in hospital settings it is needed technological designs that are based on a proper assimilation of the context where the hospital's staff performs their job. We can assimilate the context of work done at a hospital by an active engagement of researchers in daily work through which we can capture the routines, procedures, and working practices of individuals. Consequently our approach is based on workplace studies to achieve those understandings [Muñoz et al., 2003a]. Thus, from workplace studies conducted in a local public hospital real scenarios were obtained that reflect how the medical practice can be enhanced by ubiquitous computing technology.

Before presenting the scenarios that were developed based on the understanding of the medical practices, the next section presents related research of how agent technology has been used in systems that support medical activities.

3.2 Autonomous agents in healthcare

As the aim of this thesis is to explore the use of autonomous agents in real ubiquitous computing scenarios, we selected the healthcare domain to study how ubicomp technology in combination with autonomous agents can enhance users’ activities. For this reason, this section presents

Agent technology has been used to create systems with a specific focus on improving the management of clinical information within a hospital. For instance, PalliaSys is a multi-agent system that collects information about the status of palliative patients and reports it to a doctor. This system enables physicians to consult medical information through different communication technologies, such as mobile phones, PDAs and computers [Moreno et al., 2004]. In [Greenwood et al., 2003] an adaptable agent-based system is proposed to aid the medical staff in analyzing the data of diabetic patients which could be previously downloaded from the patient's PDA.

Other research projects have focused on providing support for the management of clinical information that may be distributed among the different hospitals and departments within a hospital. In [Castro-Oliveira et al., 2000] a multi-agent system is presented that physicians may use to access multiple and heterogeneous medical data sources available at the hospital, in a transparent way. This system enables the proactive construction of unified medical repositories which may contain wide range of data type, such as video, images, and text. A multi-agent community works autonomously, gathering data from the operational systems, and applying procedures for transforming the data which is then written to a patient record repository. In [Singh et al., 2004] the use of agents along with grid technology is proposed to merge organizational knowledge from various hospitals. While agents deal with the management of information, such as retrieving and presenting information, grid technology copes with issues of security, such as authentication and authorization services. In [Kirn, 2003] the concept of ubiquitous healthcare is introduced that refers to the disposition of any type of health service such that individual consumers can access them through mobile computing devices. This work presents the OnkoNet initiative which proposes developing a multi-agent system for cancer treatments. This system will enable authorized persons/institutions to store, maintain, retrieve and access medical data by using local and mobile devices in order to integrate all patients' information related to diagnosis, therapy and care.

Agents have also been used for dealing with other challenges in providing better healthcare services, such as monitoring the state of a patient, providing assistance services to remote patients, and the care of disabled and senior people. These projects selected agents as the main system's components that exhibit dynamic changes and autonomous behavior. For instance, the e-Tools project uses sensors, wireless communications devices and agent technology to provide supportive services to people with disabilities. One such e-Tool is an electric-powered wheelchair in which a multi-agent system autonomously controls its behavior, monitors the state of the patient, interacts with him/her through an interface that provides assistance in navigation, and sends messages indicating the state of the patient to the PDA of the patient's caregivers and relatives [Cortés et al., 2003]. The AINGERU project proposes a multi-agent system for offering tele-assistance services to elderly people. This system consists of agents running on the patient's PDA, in charge of monitoring the patient's health condition and communicating with other agents residing in the healthcare center which perform other tasks, such as appointment negotiation [Tablado et al., 2003].

Most of the projects presented support the management of patient information by means of the use of software agents that have well defined objectives and that must collaborate among themselves to support different activities related to the management of medical information. Thus through a multi-agent system these information management tasks are transparent to the users. To do this, these systems provide agents at different system's layers. For instance, all of them provide agents at the user interface layer, that interprets the user's request or that present the information to them. Then, agents at a higher layer are in charge of extracting this information and passing it to the interface agents in order to be presented to the user. For instance, in the case of the OnkoNet project, agents residing on a PDA can receive information requests from users, and then these agents move to a desktop to contact the agents that locate and extract the requested information from the hospital's databases. The use of agents enabled these projects to cope with the distribution and heterogeneity of clinical information. In spite of the fact that some of the above projects introduce the concept of AmI or ubiquitous computing healthcare, these systems do not address other challenges related to the implementation of ubiquitous computing environments, such as taking into account the context of users for presenting and adapting the clinical information, neither do they address the need to access the medical information from multiple and heterogeneous computing devices. However, providing systems such as the one previously presented is a step towards creating a ubiquitous healthcare computing system. An infrastructure that supports the integration and retrieval of clinical information facilitates the introduction of pervasive and context-aware systems that enable the opportunistic access of medical information.

The next section presents a set of ubiquitous computing scenarios within the health-care domain that were developed as part of this thesis. These scenarios show how a hospital can become a ubiquitous computing environment that supports the medical activities of the hospital staff and serve as a basis for identifying the type of support that an agent-based infrastructure should provide.

3.3 Context-aware medical practices of mobile users

The first workplace study was conducted in the internal medicine area of the IMSS General Hospital at Ensenada, B.C. with the goal of finding out how handheld computers along with context-aware technology could support medical activities . The focus was on capturing the essential contextual elements that support the management of information and influence hospital staff interactions with colleagues and with medical services and devices available in the environment. In this study, scenarios were used as a way to frame our understanding of medical practices and also our vision of how their work could be augmented with mobile computing devices and a context-aware system. Given that such technologies are not commonly used by the subjects in our study, scenario building enabled to generate design ideas for a context-aware hospital system, and to identify the possible users and contexts of use for the system. These scenarios are presented in the following sections.

3.3.1 Scenario: Context-aware access to medical information

While Dr. Diaz is checking the status of a patient (in bed number 1 of room 222), he realizes that he needs to request an ordinary laboratory test for her. Through his PDA, he adds this request to the patient's clinical record of the Hospital's Information System. The *chemist (responsible for taking the samples for the analysis) visits the internal medicine area every morning. His PDA informs him that inside room 222 there are three patients that require a medical analysis. When the chemist stands in front of the patient, his PDA shows him the samples that have to be taken and the type of analysis to be performed. He labels the samples and at the end of his round he takes them to the lab to perform the analyses. The results are added to the patient's clinical record. When the doctor is about to finish his shift and while walking through the corridor, his PDA alerts him that the test results of the patient in bed number 1 in room 222 are available. Dr. Diaz goes back to the patient's room and when he stands near the patient's bed the results of the analysis are displayed on his PDA. At that point, the doctor revaluates the patient and based upon the results just received, decides to prepare him for surgery.*

3.3.2 Scenario: Context-aware communication

Rita is a doctor in a local hospital. As she makes her final round, she notices that a patient, Theresa, is not responding well to her medication. Rita wishes to leave a note to the doctor who will be reviewing Theresa in the afternoon shift. She doesn't know who that will be, so she writes a message to the first doctor to check the patient after her.

3.3.3 Desirable features of the ubicomp system to support context-aware medical practices

Our study of IMSS General Hospital revealed four critical contextual elements that a context-aware system would have to consider in supporting the hospital's information management and activity coordination . The selected scenarios also illustrated how users can communicate with other users and interact with services through messages that are delivered when certain contextual conditions are met.

The scenarios present how a context-aware system for mobile users can enhance medical practices by directly addressing those contextual elements which characterize them:

1. **Location.** Where hospital staff members are at a particular time determines in part the type of information they require. For example, access to a patient's medical records is most relevant when the doctor or nurse is with that patient.
2. **Delivery timing.** Communication exchanges in a hospital tend to be time sensitive, which means that a message might be relevant for only a certain period. For example, in scenario 3.3.2 a doctor might leave a message that describes recommendations for treatment to any nurse on the next shift.
3. **Role reliance.** In hospitals, parties who might be strangers or rarely meet must communicate with each other. A user often addresses messages not to particular individuals but to "the nurse on the afternoon shift," or "the next doctor to visit the patient." Thus, the system must be able to recognize roles as well as particular individuals.
4. **Artifact location and state.** An artifact, particularly a device, can have many states. The state of devices (temperature reading) and other artifacts (availability of lab results) can be important triggers for appropriate actions, including information exchanges. Medical staff might need to communicate directly with documents or devices. For example, a doctor might want to display the patient's lab analysis on her office desktop as soon as results become available.

3.3.4 Design issues regarding autonomous agents

From the above scenarios and the desirable features identified for the system, several design issues arose regarding the functionality of autonomous agents for creating ubicomp systems.

- *Autonomous agents are decisions makers*. First, the need for third-party decision makers, which can be autonomous agents, was identified. The agents would review the context and make decisions about what activities to do, when to do them, and what type of information to communicate to whom.
- *Autonomous agents are reactive to the contextual elements of the environment.* As explained in the previous section, these contextual elements are: location, delivery time,

users' role, and location and state of artifacts and users that agents may need to monitor for opportunistically providing information and services to users. For instance, an autonomous agent can be aware of changes to medical information to be aware of when a patient's medical analyses are available and then notify it to the doctor. Another agent can monitor the environment to make sure that the contextual requirements are satisfied before delivering a message. For this, agents need mechanisms to perceive, recognize and disseminate different types of context information: role, location (of users and devices), state (of documents, services, devices, and users) and time.

- *Autonomous agents can represent users, act as proxies to information resources of the environment or to wrap a complex system's functionality.* In the scenarios, users require access to information resources such as the hospital information system from which a doctor access patient's clinical records. For this, the doctor can interact with her personal agent on the PDA that will request, on her behalf, medical information from an agent acting as proxy to the hospital information server. Finally, agents can be wrappers of complex system's functionality, such as the location-estimation agent that may need an intelligent mechanism for estimating the user's position.
- *Autonomous agents should be able to communicate with other agents, or directly to users and services.* For this, agents need a platform of communication that enables them to convey information to other agents, users, and information resources (such as the hospital information system) by using the same protocol of communication. This platform and protocol of communication should enable agents to seamlessly interact with users in order to enhance their interaction with the ubicomp environment.

3.3.5 Using autonomous agents to support context-aware communication

The above scenarios were analyzed to identify how autonomous agents can be used to design the system illustrated in the presented scenarios which was named as context-aware hospital information system. As described below, the design of this system included several agents :

- **Location-aware client.** The location-aware client is an agent with whom the user can interact in order to compose a message and specify its delivery context. The agent should include an interface that requires only peripheral attention through which users are aware of the presence of other users and the available services in the environment, and with which he can interact with them.
- **Context-aware agent (CA-a).** All context-aware messages go to this agent, which monitors the environment to determine whether conditions are such that the system can deliver the message. It has mechanisms for monitoring the environment, configuring the environment (devices available, groups of users, site map, and so on) and detecting changes in contextual information, such as the device state and user position. When the agent perceives changes in the environment context, it determines if the message's delivery context matches current conditions.
- **Location-estimation agent (LE-a).** This agent resides in all users' PDAs and estimates each user's position which is communicated to the Location-aware client which notifies it to other users and agents subscribed to its presence [Rodriguez *et al.*, 2004].
- **Hospital Information System agent (HIS-a)**. This agent provides access to, and monitors, the state of, the hospital's information system, which manages all artifacts other than devices in the form of digital documents. For example, when it detects that a user has updated the IS with the results of a laboratory analysis, the agent notifies the physician. It also provides patient information to the medical staff according to their role and location.

As part of the design of the context-aware hospital information system, scenarios 3.3.1 and 3.3.2 were rewritten to identify how the autonomous agents interact to enable the context-aware communication of hospital's personnel as described in the following sections.

3.3.5.1 Scenario: Context-aware access to medical information

The sequence diagram of Figure 2 illustrates how the agents of the system interact for the first scenario:

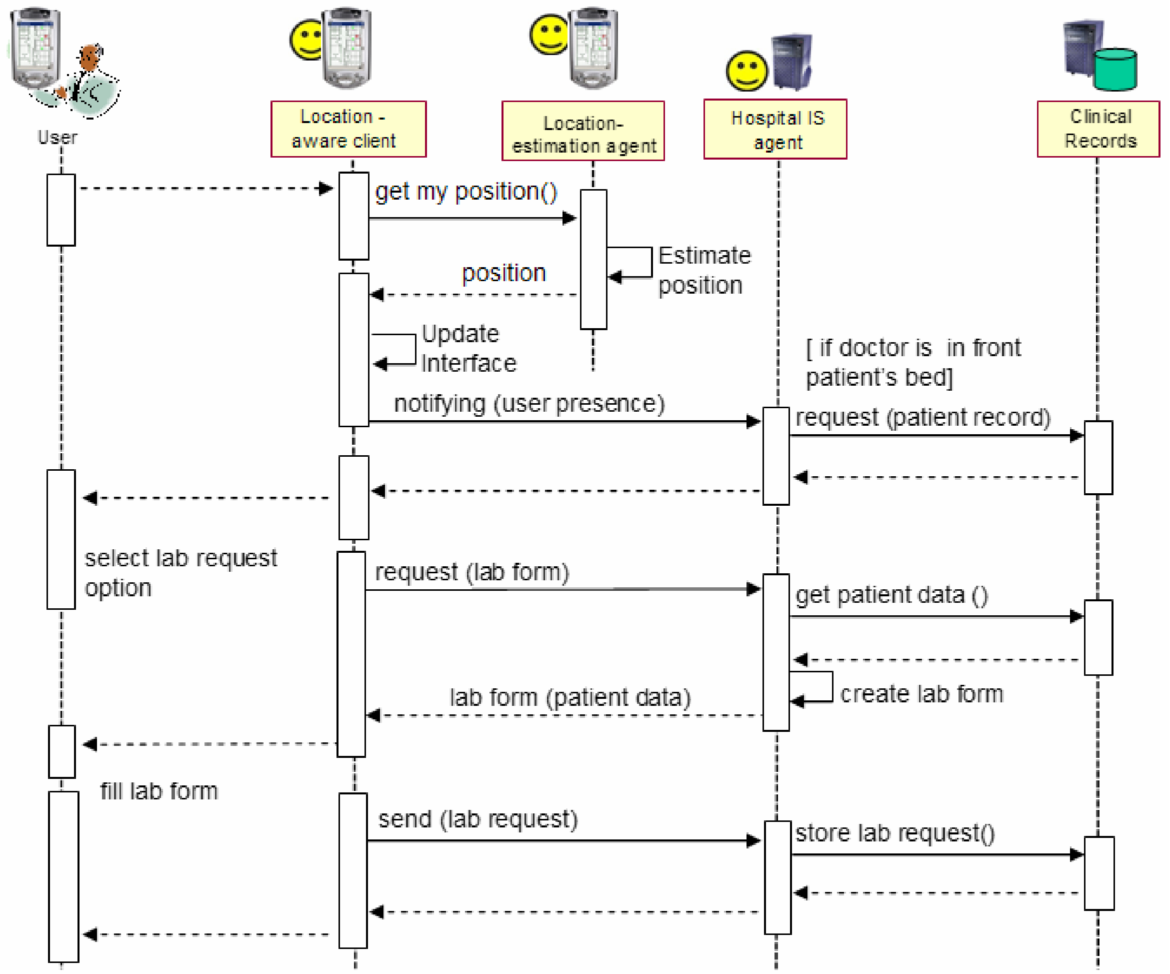


Figure 2. Autonomous agents interacting for accessing patient's clinical records

Dr. Díaz begins his daily routine by visiting each one of his patients. While he moves around the patient's rooms, the Location-aware client in his PDA communicates with the Location-estimation agent (LE-a), to constantly update his position. When the doctor's location changes, the Location-aware client notifies his position to all users and agents. Then the HIS-a verifies if its contextual conditions match the new context, i.e. the user's role and location, in order to send him a message through which the physician can directly retrieve the patient's clinical record.

After consulting both the record and the patient, Dr. Diaz decides to request a lab analysis which is customized for him based on his identity (Dr. Diaz) role (physician) and current location (bed 222). The HIS-a creates the lab form, which includes some of the patient's data such as his name and bed number, and sends it to the Location-aware client to be display for Dr. Diaz. Then, he fills the lab form and sends it to the Hospital IS to be added to the patient's clinical record.

The HIS-a will inform the chemist of the analysis to be performed and the medical samples to be taken. The chemist performs the analyses and the results are added to the patient's clinical record through the HIS-a. As Dr Diaz is still at the hospital, the HIS-a decides to notify the doctor that the test results of patient in bed number 1 in room 222 are available. Thus, the Location-aware client receives the notification and alerts him. Dr. Diaz goes back to the patient's room and when he stands near the patient's bed, the HIS-a sends the results of the analysis to his Location-aware client to be displayed on the PDA.

3.3.5.2 Scenario: Context-aware communication

Figure 3 illustrates how the autonomous agents of the system interact to support the activities of the scenario that describes context-aware communication:

Rita is a doctor in a local hospital. As she makes her final round, she notices that a patient, Theresa, is not responding well to her medication. Rita wishes to leave a note to the doctor who will be reviewing Theresa in the afternoon shift. She doesn't know who that will be, so by using the interface of the Location-aware client, she writes a message to the first doctor to check the patient after her. For this, Rita also specifies information of the user that should read it, such as his role (a doctor), and location (in front of Theresa's bed). Rita sends the message and this is received by a Context-aware agent which will be checking when the contextual conditions met to deliver the message.

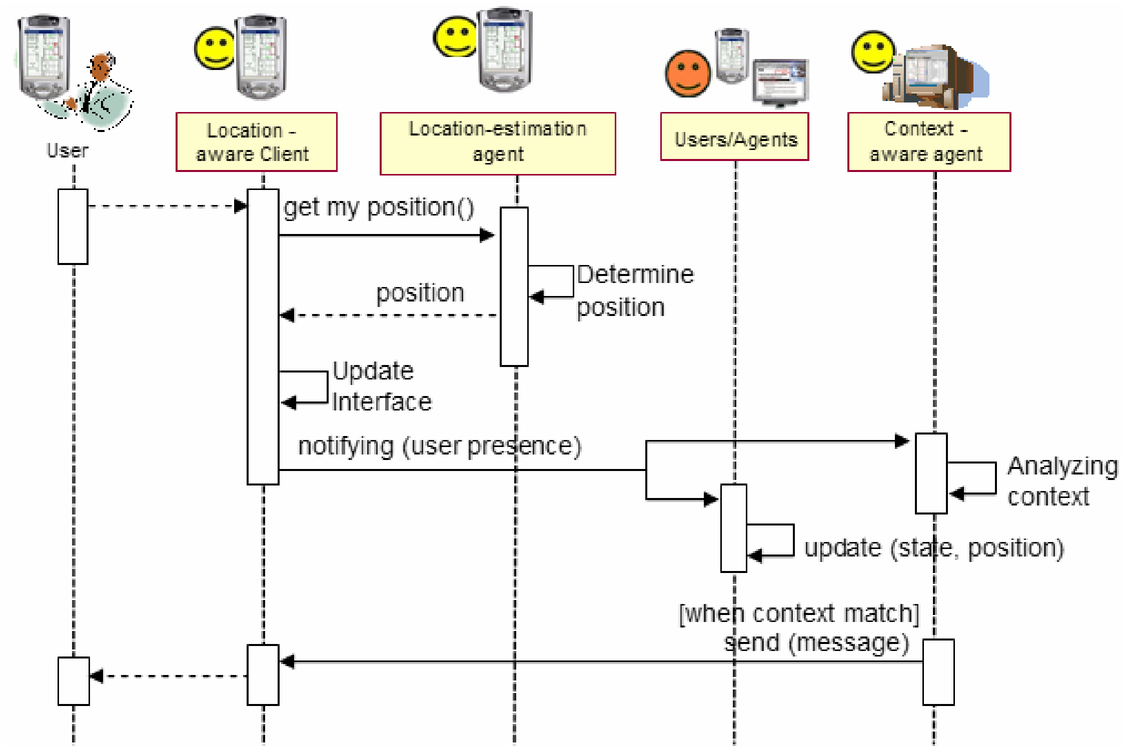


Figure 3. Autonomous agents interacting to deliver a contextual-message

Later, Doctor Gómez, the physician in that afternoon's shift, begins his daily routine by visiting each one of his patients. As illustrated in Figure 3, while he moves around the patient's rooms, the Location-aware client in his PDA communicates with the Location-estimation agent, to constantly update his position. When his location changes, the location-aware client sends the doctor's position to all authorized users and agents. When doctor Gómez enters Theresa's room, the Location-aware client updates his presence and notifies this to the context-aware agent. As the message delivery conditions match the new context, the context-aware agent sends to doctor Gómez the message written by Rita.

3.3.6 Conclusions

Conceiving the context-aware hospital information system was the result of the first iteration of the methodology proposed for exploring the use of autonomous agents for designing ubicomp systems and for making some of the preliminary design decisions for a middleware that facilitated the development of these autonomous agents. Mainly, during this phase it was identified that agents in a ubicomp system may act on the user's behalf, represent information resources or services, or wrap a complex system's functionality; these agents might run in a user's PDA (such as the Location-aware client and LE-a), a desktop computer or a trusted server (HIS-a and CA-a); finally, in order for agents to communicate with other agents and users, they need a common protocol and channel of communication.

Thus, the design issues raised by the scenarios presented were the basis for defining the functional requirements (described in Chapter 4) for designing a middleware that supports the implementation of autonomous agents for ubiquitous computing systems. In order to continue refining these design issues of autonomous agents and finding new ones, scenarios 3.3.1 and 3.3.2 were evaluated with the aim of validating that the scenarios supported real users' activities and for identifying other opportunities for proposing the use of ubicomp technology and autonomous agents for enhancing the medical activities. The results obtained from this evaluation enable this thesis to continue exploring other alternatives and propose new ubicomp scenarios. One such result showed that the participants considered that it would be useful if the system supported the visualization of X-Ray images. However, a discussion among the participants made it clear that visualizing medical images through a handheld computer had significant limitations. This suggested the use of tablet computers or public displays. As other works have reported, whiteboards hung on walls play a crucial role for coordination [Bardram *et al.*, 2003; Bardram and Bossen, 2003]. So, it was decided to further explore the second option in a second case study, not only for exploring the use of public displays for visualizing medical images, but as a means of coordination and communication.

3.4 Public displays to support coordination and communication

A second case study was performed, this one directed specifically at understanding the current uses of whiteboards, corkboards, and X-Ray viewers with the aim of integrating public displays to support coordination and communication of the hospital staff. Public displays by themselves would offer only limited services, thus we explored their integration into the context-aware mobile computing system envisioned as a result of the first case study.

The workplace study was conducted in the trauma area of the hospital with the goal of proposing innovative uses of interactive public displays in a hospital [Favela *et al.*, 2004]. The study was intertwined with the design phase. While design scenarios were being developed, issues were raised by the designers, which often led to one of the researchers going back the next day to the hospital to confirm assumptions or gather a missing piece of information. The scenarios thus combine our vision of the support that can be provided by interactive public displays with actual, and in some cases very concrete, issues faced everyday by hospital workers. The findings from the study were used to inspire the creation of design scenarios that depict innovative uses of interactive public displays in a hospital [Favela *et al.*, 2004]:

3.4.1 Scenario: Accessing medical information through public displays

While Dr. Garcia is checking the patient in bed 234, his PDA alerts him that a new message has arrived. His handheld displays a hospital floor map indicating to him that the X-ray results of patient in bed 225 are available. Before Dr. Garcia visits this patient, he approaches the nearest public display that detects the physician's presence and provides him with a personalized view of the Hospital Information System. In particular, it shows a personalized floor map highlighting recent additions to clinical records of patients he is in charge of, messages addressed to him, and the services most relevant to his current work activities. Dr. Garcia selects the message on bed 225, which opens windows displaying the patient's medical record, the X-ray image recently taken and the hospital's medical guide related with this case. While Dr. Garcia is analyzing the X-ray image, he notices in the map, that a resident physician is nearby and calls him up to show him this interesting clinical case. The resident physician notices that this is indeed a special case and decides to make a short note on his handheld computer by linking both the X-ray image and the medical guide. He can later on use these links to study the case in more detail or discuss it with other colleagues from any computer within the hospital.

3.4.2 Scenario: Physicians collaborating through ubiquitous devices

The cardiologist is notified on his PDA that the electrocardiogram he requested for the patient in bed 233 is available. The doctor approaches the nearest public display that detects his presence and provides him with a personalized view of the Hospital Information System. The doctor selects bed 233, which opens windows displaying the patient's medical record and the electrocardiogram recently taken. While he is analyzing the information, he notices on the map, that the traumatologist assigned to this patient, is walking down the corridor in the next floor. By selecting the icon representing the doctor, the cardiologist can invite him to a collaborative session. The traumatologist receives a message indicating that the cardiologist would like to discuss a case with him and specifying the location of the nearest display available. He accepts the invitation by tapping the "accept" button on the message, and moves to the nearest display. When the display recognizes his presence, it opens an application sharing and audio-conference session with the cardiologist. Both doctors can now browse the patient's medical record and discuss whether it is appropriate for the scheduled operation to take place considering the patient's cardiac condition.

3.4.3 Management of medical knowledge through public displays

Scenario 3.4.1 illustrates how pervasive technology, that integrates handhelds computers and large public displays, can be used to provide timely access to medical knowledge thru context-aware information retrieval.

Physicians, who are in a continuous learning process through their daily practice, are motivated to seek information to reduce the uncertainty of the route they ought to follow for the patient's treatment when faced with a complex clinical case [Casebeer *et al.*, 2003; Schoen, 1983]. During the workplace study it was observed that the hospital provides medical guides to be consulted by the physicians. However, given their current workloads, they seldom have the time to search for information on the local medical guide or in medical digital libraries. On the other hand, doctors often use information from previous clinical cases in their decision-making. For this, they might rely on their own experience or consult other colleagues who are more experienced on a given subject, or just to have a second opinion. Physicians, however, seldom consult the clinical records of previous patients, to a large extent because they are difficult to locate, or because the data that could be used to retrieve them, such as the patient's name or the date of a given case, are difficult to recall. The development of hospital information systems that provide access to electronic patient records is a step in the direction of providing accurate and timely information to hospital staff in support for adequate decision-making.

Traditional Web information retrieval systems are isolated from the context in which a request occurs. Information requests occur for a reason (related to the current user's activity), and that reason grounds the query in contextual information necessary to interpret and process it. Without access to this context, requests become highly ambiguous, resulting in incoherent results, and unsatisfied users [Budzik and Hammond, 2000]. This is evident in the health-care domain, in which vast quantities of medical information are now available through the web and can easily lead to information overload [Budzik and Hammond, 2000]. One way to overcome such a problem is to provide an environment that proactively retrieves and presents information based on the hospital professionals' context, thus providing Context-Aware information Retrieval (CAR) [Brown and Jones, 2001]. Context-aware computing technology is a key element to construct this new generation of Web retrieval systems by sensing the changes on the users' activities, to predict users' interests and then, retrieve information based on them.

Ubiquitous and context-aware computing technology may provide support to healthcare professionals for opportunistically acquiring, managing, and sharing knowledge, which are issues faced everyday by hospital workers. The aim of the following scenario is to bridge the gap between the medical knowledge management practices and an idealized pervasive hospital environment. This scenario addresses current sources of misshapes or look to simplify time consuming tasks, such as searching in digital libraries from the Web through the use of ubiquitous computing technology .

3.4.4 Scenario: Context-aware presentation of medical knowledge

While Dr. Garcia is evaluating the patient in bed 234, her PDA alerts her that a new message has arrived. Her handheld displays a hospital floor map indicating her that the X-ray results of patient in bed 225 are available. Before Dr. Garcia visits this patient, she *approaches the nearest public display that detects the physician's presence and provides her with a personalized view of the Hospital Information System. In particular, it shows a personalized floor map highlighting recent additions to clinical records of patients she is in charge of, messages addressed to her, and the services most relevant to her current work activities. Dr. Garcia selects the message on bed 225, which opens windows displaying the patient's medical record and the X-ray image recently taken. Aware of the context of the situation (patient's original diagnosis, the fact that X-rays where just taken from the patient's hand, etc.), the system automatically opens a window with the hospital's medical guide that relates to the patient's current diagnosis, and shows an icon which displays references to previous similar patient's cases. Dr. Garcia analyses the X-ray image and writes on it to emphasize the clinical problem. As she is interested in knowing about the efficiency of the treatment received by other patients with similar problems, she selects a reference to a previous patient's case. By the doctor's interaction with the information on the public display, the system infers she is still interested in learning more about this case, in particular, in alternative treatments for this medical condition. Then the application suggests links to medical digital libraries from the Web that are considered by the hospital's clinicians as reliable sources of medical information. She selects the first recommended Web link which presents information about alternative treatments. Thus, Dr. Garcia uses this trusted medical information combined with their medical experience to make a more accurate decision about the treatment to follow.*

3.4.5 Desirable features of context-aware public displays

Based on the findings and scenarios such as the one presented above, we identified the following aspects to be addressed by context-aware interactive public displays [Favela *et al.*, 2004]:

- **User's location and authentication.** Hospitals are characterized by the mobility of the professionals that work there, that of the artifacts they use, such as clinical records or medical equipment, and even the patients, who are moved from one hospital area to another as required. When a doctor examines a patient, he needs to move to obtain the patient's clinical records and other documents. Hospital workers also need to move to locate information displayed in whiteboards. For instance, the schedule of patient's operations for the current day is displayed on the whiteboard in the office of the chief surgeon, and the activities and working area assigned to nurses are advertised in different boards throughout the hospital. Boards help to communicate information regarding patients' condition and location, and hospital staff often visits the boards to find this information. Thus, hospital workers require access to information from anywhere within the hospital. In addition, as exemplified by the scenarios, the public display should be able to recognize the user as he approaches it and give him access to relevant clinical data without cumbersome login procedures.

- **Content adaptation and personalization based on contextual information.** Contextual information such as, location, role, and identity should be taken into account to adapt and personalize the presentation of information to the user. For instance, in scenario 3.4.1 when Dr. Garcia is in front of the display, the pervasive environment displays the messages addressed to him and highlights the beds of the patients he is examining. A traumatologist attends to mostly a small and well defined number of patients assigned to him. He might be required to occasionally access the medical record of another patient, but this is seldom the case. Thus, information overload is prevented by personalizing the display to provide immediate access to the clinical records of those patients assigned to the doctor.
- **Information transfer between heterogeneous devices.** In the hospital users frequently transfer information from public spaces to personal spaces. For instance, the chief nurse might leave a note on a public board in order to advertise the date of the next meeting; then, another nurse would write this information into her personal agenda. A few physicians actually carry PDA's and reported using them to record information displayed on whiteboards or corkboards. A pervasive environment furnished with devices of all scales, should support the simple and safe transfer of information between devices. Some of these devices are public, such as large displays, while others are personal, such as PDAs. In scenario 3.4.1, the resident physician decides to keep a personal record of the clinical case presented on the display. Thus, the physician

transfers the relevant information from the large display (public space) to his PDA (personal space). A doctor may also want to transfer information from his PDA to a public display in order to discuss it with a colleague such as illustrated by scenario 3.4.2. Information is also transferred from both devices when the physician approaches the board and is authenticated, the display will use information stored in the user's PDA to personalize the applications running in the public display. For instance in the scenario, the message received by the physician indicating that the X-rays are available and the dates stored in the PDA of the physicians' calendar are transferred to the display to personalize the map and calendar applications.

- **Context-aware retrieval of medical information.** The electronic patient record offers the opportunity to retrieve relevant medical cases with little effort from the user. To retrieve this information, the pervasive environment has to take into account contextual information, such as, type of clinical problem, and if the device where it will be displayed is a public display that enables to clearly appreciate and analyze documents and images. The recommended clinical cases are ranked by similarity with the current case. For instance, it may be more relevant for the physician to consult first how she solved previous cases akin to this, and after that, find out the treatment or diagnostic given by other clinicians to similar problems. Besides, the most significant clinical cases are those that coincide with the personal patient's data such as gender, age or weight, which may be factors correlated with the causes of a medical problem or treatment. Depending on the type of diagnosis some of these data will be more relevant than other to retrieve previous cases. The environment may also opportunistically display medical guides relevant to the patient's diagnosis as supporting information for doctors or even locate and establish contact with a specialist who might be available. In the scenario, the display presents the hospital's medical guide relevant to the case they are discussing and links to previous cases that were estimated to be relevant. Based on the description of the diagnosis, the system presents the hospital's medical guide related with this particular case.

3.4.6 Design issues regarding autonomous agents

From the desirable features of a context-aware public display that support the activities depicted in Scenarios 3.4.1, 3.4.2 and 3.4.4, autonomous agents were identified as the software entities that enable the personalization and adaptation of medical information on the public display and the transferring of information between different devices. Other design issues came up which complemented those identified during the previous iteration of the methodology followed for creating a middleware for ubicomp systems:

- *Autonomous agents act as proxies to devices.* Agents representing devices, such as the public display, can be aware of the presence of other agents and users available in the environment. These agents enable allowed users and other agents to use the devices. For instance, presenting information on the public display. From this, the following design issue was raised.
- *Autonomous agents need mechanisms for authentication.* Autonomous agents representing devices or services need mechanisms for authenticating users and agents that want to access them. For instance, not all users are allowed to access the hospital information system or present information on the public display. The physician agent on the PDA may need to know a priori the agent acting as proxy to the public displays available on the environment. An autonomous agent such as this, that represents a public display, needs mechanisms for authenticating users that require accessing it. Thus, only authorized personnel may access these devices or services. And the same can be applied to agents. For instance, the agent acting as a proxy to public display needs to know in advance which other agents may require to publish information on them, or request the personalization of information for a user. Similarly, the Map agent needs to authenticate whether the agent requesting its services has permission for accessing it.
- *Autonomous agents need to communicate different types of messages.* From the scenarios presented in this section and section 3.3, were identified that agents need a communication language that enables them to communicate. This communication language needs to convey messages for requesting information from devices or services

(i.e. requesting medical information from the hospital information system) and responding to such requests, notifying information to users and devices (i.e. notifying that the lab results are available to the user), and requesting from another agent the execution of an action (i.e. personalizing the hospital map for the user).

- *Autonomous agents may have a reasoning algorithm as complex as the logic of its functionality.* An autonomous agent needs to be aware of information regarding the environment in order to generate its actions. For this, agents may need a reasoning algorithm which may include a simple set of rules or conditions, such as the Context-aware agent (Ca-a), from Scenarios 3.3.1 and 3.3.2, which verifies that a set of conditions are met in order to deliver a message; or a more complex reasoning algorithm needed for estimating the user's location, or for indexing and retrieving medical cases such as in the Scenario 3.4.4.

3.4.7 Using autonomous agents to support context-aware public displays

The design of the architecture for the context-aware hospital information system was extended to integrate public displays that are context-aware, to personalize information to the user, to allow the transfer of information to and from PDA's, and to offer opportunistic access to clinical information. To accommodate these features, three agents were identified:

- **Public Display agent (PD-a).** This component acts as a proxy for the public display. The Public Display agent (PD-a) represents the public display available at the hospital. It enables users to access the Public Display and have control over the applications displayed.
- **User's Proxy agent (UP-a).** This agent acts as proxy of the user. It visually represents the user by presenting his photograph in the display and it references information stored in the user's PDA. By selecting the icon with his own photo a user opens a folder with references to documents stored on his PDA or WebDAV folder . Finally, if a user drags and drops a window onto his photo icon it will add a link to the document contained in the window (electronic patient record, X-ray image, medical guide, etc.) into the folder in his PDA.

- **Medical Guide proxy-agent (MGp-a).** It is responsible for displaying the hospital medical guide. The agent retrieves information from the record of the patient being consulted by the user of the public display, such as the current diagnosis and type of analysis performed and displays a medical guide, if one is found to be relevant.

By revisiting scenario 3.4.1, the next section illustrates how the system's autonomous agents address the desirable features of context-aware interactive public displays discussed above.

3.4.7.1 Scenario: Accessing medical information through public displays

As Dr. Garcia examines the patient in bed 234, the X-ray results he requested are included into the electronic record of his patient in bed 225. As depicted in Figure 4 the HIS agent notifies this to the doctor by sending him a message. Dr. Garcia approaches the nearest public display when finished with his current patient, and before visiting patient 225. The doctor's location, which is constantly being tracked by the location-estimation agent on his PDA, is notified to all users and agents in the environment, such as the context-aware agent and the Public Display agent (PD-a). The PD-a acknowledges the user's presence by displaying his photograph, indicating with this that the user has been logged into the system, and the applications on the display are personalized for him. In addition to his photograph, Dr. Garcia will be shown a personalized public map indicating the location of hospital staff and services available (printer, public display, etc) and highlighting the beds of patients assigned to the current user of the display. The public map also shows messages addressed to the user that depend on his location. This includes messages related to his patients, such as additions to their electronic records (e.g. analysis results or a medical note).

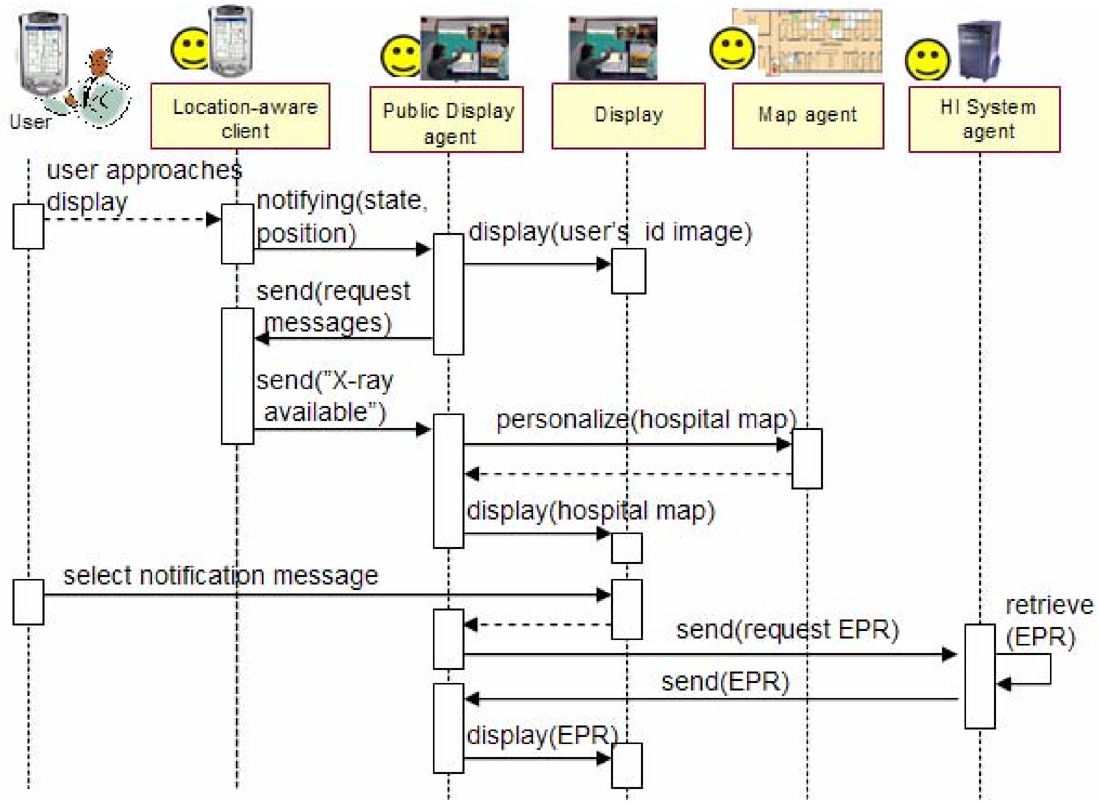


Figure 4. Autonomous agents' interactions to personalize information on the public display

As Dr. Garcia wants to examine the recently taken X-rays and thus selects bed 225. The public display agent requests the electronic patient's records to the HIS agent and these are presented to the user in two windows, one with the main page of this patient's medical records, and a second one with its latest addition, in this case, the X-ray image of interest to the doctor. A Knowledge Management agent (KM-a) will also be notified that the user is consulting the records of a particular patient. The KM-a will open the relevant medical guide and manifest himself in the public display with an icon next to the user's photo.

While Dr. Garcia is analyzing the patient's medical condition, he notices on the map, that a resident physician is nearby and calls her up to discuss with her this clinical case. The resident physician considers this case to be of particular interest and decides to store a link to this information in her personal space. She does so by dragging the window with the

X-ray image to her user-proxy agent represented by her photograph. As illustrated in Figure 5 the user-proxy agent sends the link to the image to the Location-aware client in order to be saved on the user's PDA. Thus, the resident physician can later on access this information.

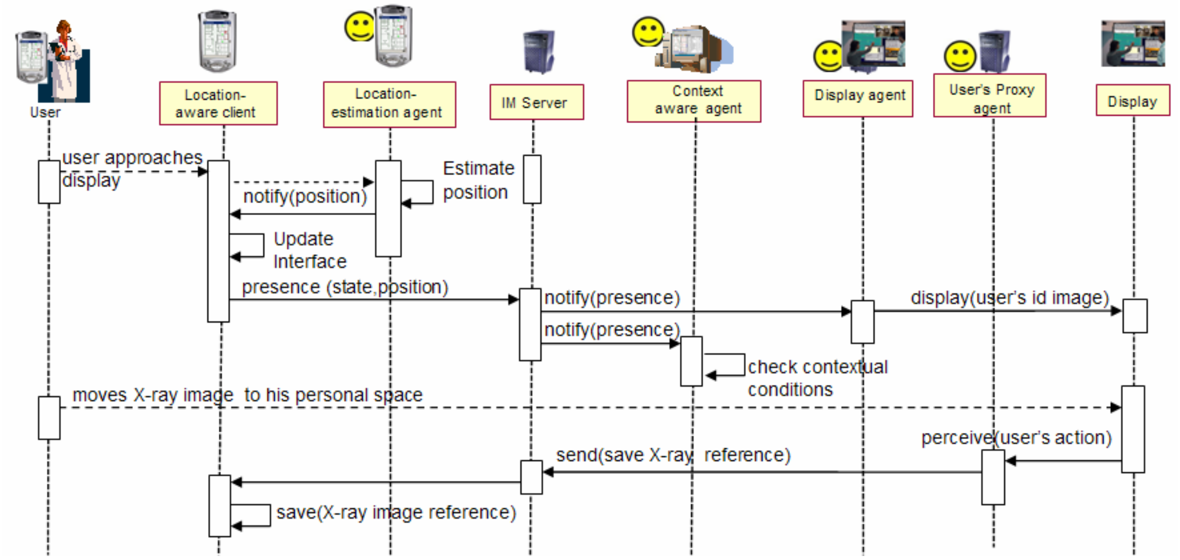


Figure 5. Agents interactions for transferring information from the public display to the user's PDA

During the design of this system, new uses for public displays were explored. Specifically, we identified that accessing medical knowledge on the public display, such as medical guides and previously documented cases, can help physicians make clinical decisions. For this, the system was extended by adding an autonomous agent that enables the context-aware retrieval of medical information:

- **Knowledge Management agent (KM-a).** This agent was designed to use a Case-based Reasoning (CBR) paradigm aimed at solving new problems by retrieving and reusing similar experiences of past cases [Aamodt and Plaza, 1994]. The model of the case memory consists in organizing specific medical cases with the same diagnosis under a generalized episode (GE) [Aamodt and Plaza, 1994]. Each GE contains indices that are features that discriminate between GE's cases, for instance, medical treatment, or the name of the physician who treated that patient. Thus, the GE is an indexing structure for matching and retrieving similar cases. The KM-a first identifies the features that best describe the patient's medical condition given by the diagnosis, which are translated to a search query. Then the agent retrieves a set of plausible candidates by comparing the query to the descriptions of the generalized episodes in the index structure. The agent will use specific patient data (gender, age, etc.) as a criterion to rank the retrieved cases. References to these previous cases are presented to the user when requested. Finally, the KM-a updates the case memory by (a) adding a reference to the patient's case as a leaf of the GE tree, if one of the recommended treatments was applied, (b) as a leaf indicating that it is a case by the physician in particular, and (c) creating a new GE if it is a new clinical case.

The following scenario describes the interactions of the agents identified during the design of the context-aware public displays system and the KM-a to support the context-aware retrieval of medical information.

3.4.7.2 Scenario: Context-aware presentation of medical knowledge

As represented in the sequence diagram of Figure 6, when the Public Display agent (PD-a) retrieves the information most recently added to the patient's records, it notifies to the Knowledge Management agent (KM-a) and the Medical Guide proxy agent (MGp-a) that this document has been presented to the user. The KM-a formulates the search query based on the diagnosis and personal information of the patient, and the MGp-a formulates its query based on information that describes the patient's medical condition, such as, symptoms and diagnosis. Based on these queries, the agents retrieve medical information from the Case Base (CB) repository and from the Hospital's Medical Library (HML) respectively. The retrieved information will be sent to the Public Display agent (PD-a) that will adapt it on the display. For instance, while the medical guide is automatically displayed, the links to previous clinical cases will be represented with the KM-a icon. In our scenario, while Dr Garcia analyzes the X-ray image and the medical guide retrieved by *the MGp-a, she selects the KM-a agent's icon for additional recommendations, then the links to previous relevant cases are displayed.*

Finally, the Doctor selects the link to the first recommended case, which is ranked as the most relevant since it is one of her patients and the medical condition is the most similar to the new patient's case. When the physician prescribed the treatment to her patient, the KM-a perceives that it is highly similar to the case consulted earlier. Then the agent updates its case-base by adding this case to the same generalized episode.

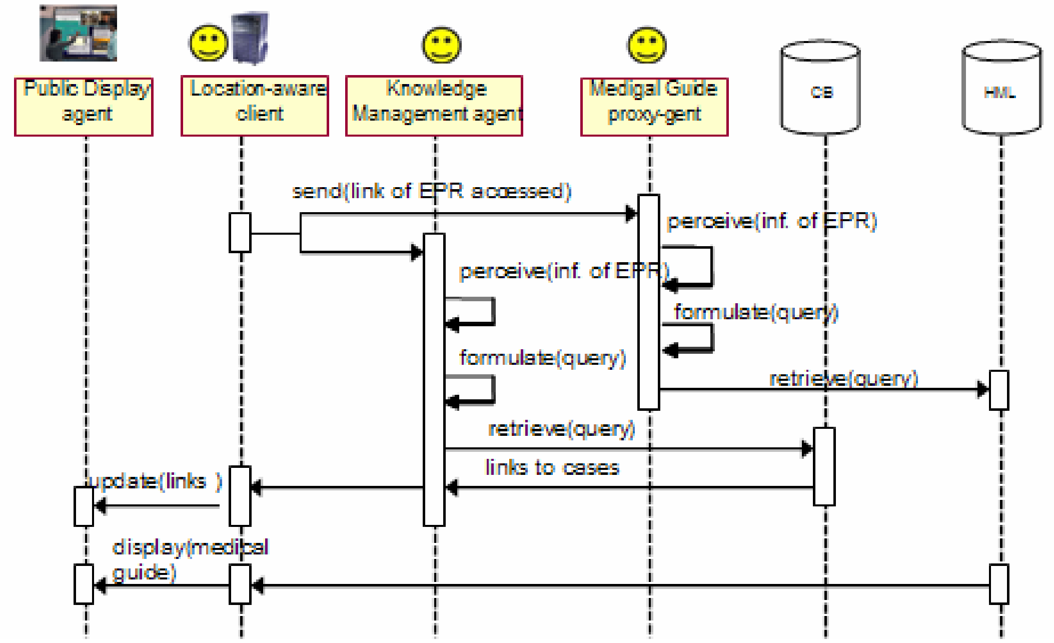


Figure 6. The Knowledge Management Agent and the Medical Guide proxy-agent retrieving medical information

3.4.8 Conclusions

This section has presented how autonomous agents can be used for addressing other desirable features of a ubicomp system, which are different from those features that were identified during the design of the context-aware hospital information system that is characterized by supporting the users' mobility and the context-aware access to medical information. The above scenarios were proposed with the aim of exploring the integration of other ubiquitous computing devices and services to the design of the context-aware hospital information system by means of the use of autonomous agents. Thus, it presented how autonomous agents can be used as the main components required for extending the functionality of a ubicomp system. To do this, autonomous agents acted as proxies to devices and services incorporated into the system without requiring modifications to the design of other system components.

In order to enable developers to easily extend the functionality of a ubicomp system by means of autonomous agents, they need well defined interfaces for communicating with other agents and users, as well as for perceiving information directly from the devices or services they represent. These issues are reflected in the definition of the requirements for the agent middleware for ubicomp systems which are presented in Chapter 4.

3.5 Discussion

As presented in this chapter, scenarios were used for defining the functionality of ubicomp technology in hospitals. In a first stage, the scenarios were used as a way to frame our understanding of hospital work. For instance, the researchers that carried out the workplace study at the hospital, participated in a design session by communicating their findings to the designers, if there was some misunderstanding regarding the medical practices, or some new issue arose, researchers went back to investigate it. Thus, scenarios enabled us to visualize how the medical practices could be augmented with ubicomp technology. Some scenarios were grounded in workplace studies, and then validated by the people involved in them. Other scenarios such as the one presented in Section 3.4.4 (Scenario: Context-aware presentation of medical knowledge) were not validated, but they were created based on observations and facts found during the study, and by investigating other research work regarding the use of medical guides and evidence-based medicine. This with the aim of proposing real scenarios that supported and reflected the activities of the hospital’s personnel.

Following the iterative methodology proposed for implementing a middleware for ubicomp systems, the following phase consisted of analyzing the proposed scenarios to find out how autonomous agents can be the proactive components that cope with the desirable features of these systems. By rewriting the scenarios, it was possible to visualize how the user's activities could be supported by autonomous agents that seamlessly execute actions for the users (such as personalizing information in a public display) or on behalf of a user (i.e. delivering a message to another physician when certain contextual conditions are met). In these scenarios, autonomous agents were proposed as the main components of a ubicomp system since they have the following attributes that help to implement the desirable functionality of the ubicomp system: Autonomous agents are *proactive* entities that control their behavior [Bradshaw, 1997]; they are *reactive*, which means agents perceive their environment and respond in a timely fashion to changes that occur in it; agents have a *collaborative behavior* since they must be able to communicate with the users or devices that they represent and with other agents in order to collectively tackle problems that no single agent can handle individually; they also have *temporal continuity*, this means that agents remain in execution while they provide relevant services to users, or until their services are no longer needed in the ubicomp environment; they also possess *mobility* which refers to the fact that an agent is able to migrate in a self-directed way from one host platform to another [Bradshaw, 1997]. This last characteristic was identified in agents that require updating their reasoning behavior when certain conditions in their environment change.

Thus, the selected scenarios were used to define the functionality that can be supported by autonomous agents, identify how autonomous agents augment users' activities, and use autonomous agents as a design abstraction for designing ubicomp systems. The design issues raised during the analyses of these scenarios were used as the basis for identifying the functional requirements of an agent middleware for ubiquitous computing systems and designing this middleware as described in Chapter 4.

Chapter 4

The SALSA middleware

Once the use of autonomous agents for designing ubicomp systems depicted in the scenarios previously presented was explored, the next step was to make the design decisions for a middleware that facilitated the development of these autonomous agents. Based on the analysis made through the scenarios, the functional requirements for an agent middleware for ubiquitous computing systems were identified. This section presents these requirements and the design and implementation of the middleware.

4.1 SALSA’s functional requirements

Chapter 3 presented how autonomous agents can be used for designing a ubiquitous computing system in which autonomous agents play the role of decision makers in the environment. Each of the agents has specific goals, and to achieve them, they have to monitor the context of the environment, and autonomously decide how to act. This lead to the first requirement for the agent middleware for ubiquitous computing systems proposed in this thesis:

To implement autonomous agents as decision makers, the middleware should provide mechanisms for implementing the agents’ components for perceiving, reasoning, and acting. An agent should include a perception module to gather information from environment sensors and devices (e.g. to estimate the users location), directly from users (through an interface), and other agents; a reasoning module governs the agent’s actions, including deciding what to perceive next; and finally, the agent executes the appropriate action, which may include: sending a notification message to a user or requesting a service from a device.

Autonomous agents in a ubiquitous computing environment can perceive different types of context information. For instance: the Location-aware client may need to be aware of the location and state of users and devices to predict the user’s needs; this information is provided by agents representing other users and devices; the Knowledge Management agent needs to abstract information of the case being consulted by the physician in order to present relevant medical information to him. Thus, the requirement is:

The middleware should provide higher-level mechanisms to enable agents to perceive context information from other agents, directly from the devices or services, or from the users. Considering that ubiquitous computing environments are highly dynamic, mainly due the fact that the user's context may change unpredictably, the agent's perception component should be able to perceive context information at unpredictable times. The middleware should provide an agent communication language flexible enough to enable programmers to specify the type of context information that will be perceived by the agents of a ubicomp system. Thus, the agent communication language should enable agents to identify the type of information perceived and the target of the information.

Once an agent received information, the reasoning component of an autonomous agent has to analyze it to decide how to act. As illustrated by the scenarios, the agent's reasoning algorithm can be as simple as a set of rules for checking when the contextual conditions are met for delivering a message to a user, or more complex such as the Case Base Reasoning of the Knowledge Management agent, which lead to the following requirement:

The middleware should be flexible enough to enable developers to endow agents with any reasoning algorithm. The middleware should provide abstractions that permits not only the implementation of any kind of reasoning algorithm, but also enables developers to easily modify or update the agent's reasoning requiring little or no modifications to the other agent's components.

Agents may need to update its reasoning algorithm at execution time. This particular requirement came up particularly from the features of the Location-estimation agent that resides on the user´s PDA. This agent should be able to estimate the user's location independently in of the floor or department of the building in which is located the user. For this, it is proposed that the middleware provides facilities for enabling agents to update their reasoning algorithm by acquiring the code from a trusted server.

The actions of autonomous agents may require them communicate with other agents by interchanging different types of messages for conveying the intention of the interaction and the information content. Agents may also need to communicate with users in order to notify and present information to them. For this, it is require that:

The middleware should provide a communication platform that enables agents to convey information to other agents, users, devices and services by using the same protocol and communication channel. The communication platform should enable users to be aware of the presence of other users and agents that offer relevant services for their activities, but they should be unconscious of other agents with whom they do not need to interact explicitly or that hide a complex functionality. Finally, it should enable agents to negotiate services with other agents, or request them to execute an action. Thus, based on the design issues regarding the communication of agents presented in Chapter 3, the messages conveyed among agents can be of the following types: a request of information, response to a request, request to execute an action or service, notification that the action was executed, notification of information (that was not previously requested), notification of the presence of an agent (representing a user or a service), and notification of information perceived from a device or sensor. The communication language should be flexible enough to allow programmers specify the content of each of these messages and the programming language of the middleware should facilitate the creation of these messages.

Autonomous agents need mechanisms for registering and authenticating agents. Autonomous agents need to know what agents have permission of requesting their services. The middleware should provide an infrastructure of services that enable the naming, registration, authentication, and location of agents representing users, devices and services.

To enable developers to create the software entities of a ubiquitous computing environment with which users need to seamlessly interact, the Simple Agent Library for Smart Ambients (SALSA) was created. The following Section presents the design of SALSA to address the above requirements.

4.2 Design of SALSA

The design of SALSA includes defining: the agent's life cycle which implements the agent's behavior; an expressive language that enables agents to communicate with other agents; and the architecture which consists of a set of services and a library of abstract classes to implement autonomous agents

4.2.1 Agent's life cycle

Since it was identified that a SALSA agent should have components for perceiving, reasoning and acting which may involve communicating with other agents, the life cycle of agents in a ubicomp environment was defined. As showed in Figure 7 the agent's life cycle includes the following states:

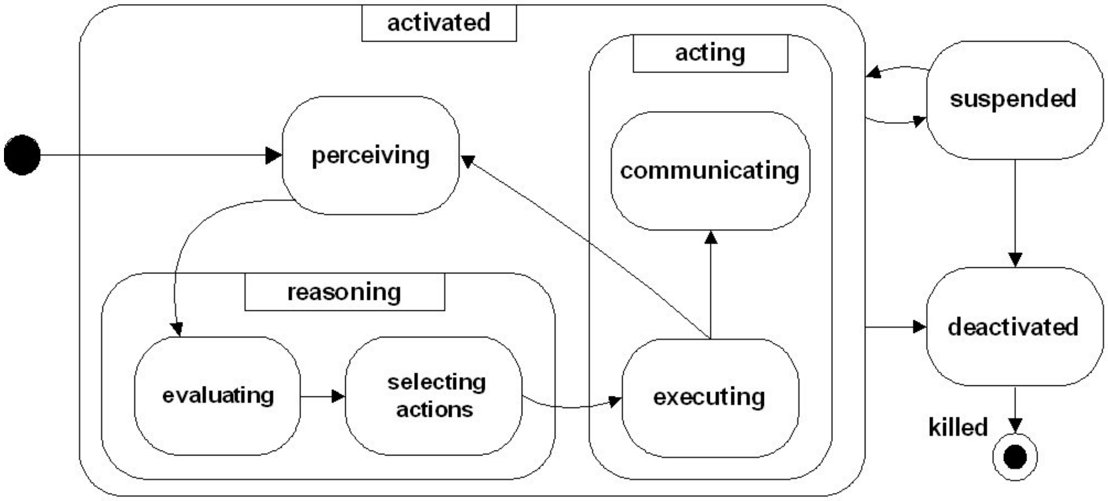


Figure 7. The life-cycle of an agent

- **Activated.** This is the initial state of an agent when it is created. This is a super-state that contains different sub-states that an activated agent can present.
- **Perceiving.** It is the initial sub-state. An agent acquires contextual information from its environment in different ways. For example, an agent may receive information from a user or another agent representing a service; or agents may perceive information

directly from devices or sensors. An agent may get into this state if its action plan requires it. That is, it moves from the acting state to the perceiving state.

- **Reasoning.** The agent evaluates the perceived information, which may require applying a simple or complex reasoning algorithm to interpret or transform this information into derived contextual information.
- **Executing.** Based on the results of the reasoning component, the agent executes an action that may involve: communicating with other agents; moving agent's code to another platform, continuing perceiving information, or terminating its execution.
- **Communicating.** The agent interacts with one or more agents in order to provide the information that is necessary to reach its goals. An agent can enter into this state if it is dictated by its action plan or because the agent needs to transmit specific knowledge.
- **Suspended.** An agent in this state is alive but it is not performing any activity. It is waiting to be reactivated. For example, if an agent is waiting for information that it has requested from another agent, then it changes from the communicating state to the suspended state, and when contacted by another agent, its state returns to communicating.
- **Deactivated.** If the agent has met its goal, then it is deactivated and killed.

4.2.2 Architecture of the SALSA middleware

As illustrated in Figure 8, the SALSA middleware consists of the following layers:

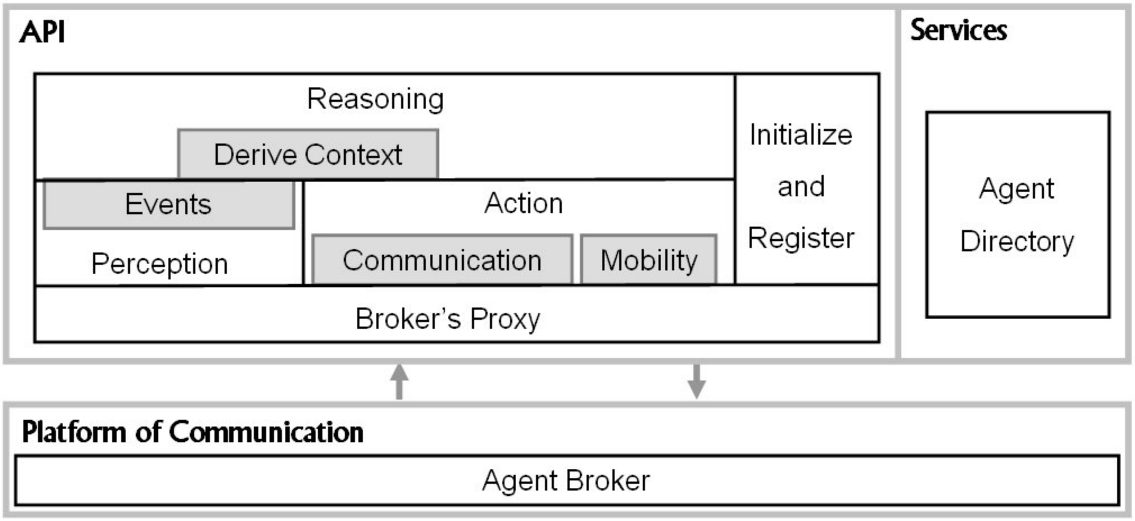


Figure 8. SALSA's Architecture

4.2.2.1 Communication Platform

The communication channel among agents and users is a *Broker* component which is responsible for coordinating the communication, and enables the configuration of the environment in a manner that is transparent for users since they do not know the location of the agents even though they can be aware of the presence of some of these agents. Thus, an *Agent Broker* is the component that should handle the communication among the ubiquitous devices, services and users, which are represented by agents. SALSA should provide a protocol of communication which consists of an expressive language that enables the exchange of different types of objects between agents (such as perceived information, requests for services), between agents and users (such as events generated by the user's actions), and between agents and services (such as the state of a service). This information will be sent or perceived by the agent through a proxy to the Broker, which is an agent's component. The *Broker's Proxy* and the set of messages that can be communicated among agents are created by developers by using the SALSA API as explained in the next section.

4.2.2.2 API (Application Programming Interface)

This is a class framework designed for facilitating the implementation of the agents' components and the use of the services provided by the SALSA middleware. Thus, it is the SALSA API that enables developers to implement the agent's components for perceiving, reasoning and acting, and to control the agent's life cycle. The *Perception* component gathers context information from the environment's sensors and devices, from the users through a graphical user interface, and from other agents through the *Broker's Proxy*. The perceived information generates events which are captured by the *Reasoning* component, which governs the agent's actions. The programmer, based on the logic of the agent, implements this component by using a reasoning algorithm, such as a simple condition-action rule, a neural network or case based reasoning. The *Action* component contains the action plan to follow based on the agent's reasoning. It also includes sub-components that allow the agent *Communication*, *Mobility* in order to update its reasoning component, and to *Derive Context* information based on information perceived by the agent.

4.2.2.3 Services

The SALSA middleware provides an *Agent Directory* service which is accessible through the *Initialize and Register* module of the SALSA API. It provides a set of classes that allow programmers to register the agent's attributes in one or more *Agent Directories*, and enable agents to look for services provided by other agents.

4.3 Implementation of SALSA

Figure 9 presents a physical view of the middleware represented as a package containing the following components:

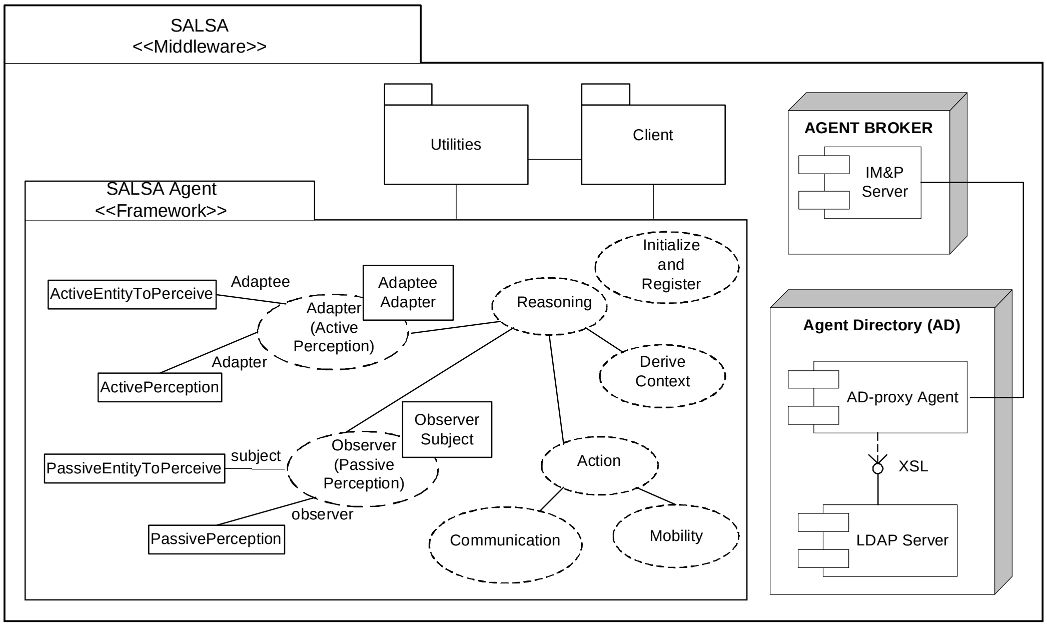


Figure 9. Architecture of SALSA

- Resources represented by physical components deployed in nodes.
- A class framework represented by a stereotyped package that includes the collection of SALSA classes. These classes are grouped in collaborations that shape the way in which they were organized.
- And the Client and Utilities packages that provide the classes to facilitate the development of the agent’s proxy to the Broker and to parse the messages communicated via the Broker.

An explanation of the main SALSA subsystems follows:

4.3.1 Agent Broker

Given the nature of ubiquitous computing systems in which agents need to asynchronously notify information to users and communicate with other agents by using the same channel of communication, an Agent Broker was proposed. The implementation of the Agent Broker is the Jabber Instant Messaging and Presence (IM&P) server (www.jabber.org). This server reports the state of people and agents, and handles the interaction among people, agents, and devices through XML messages.

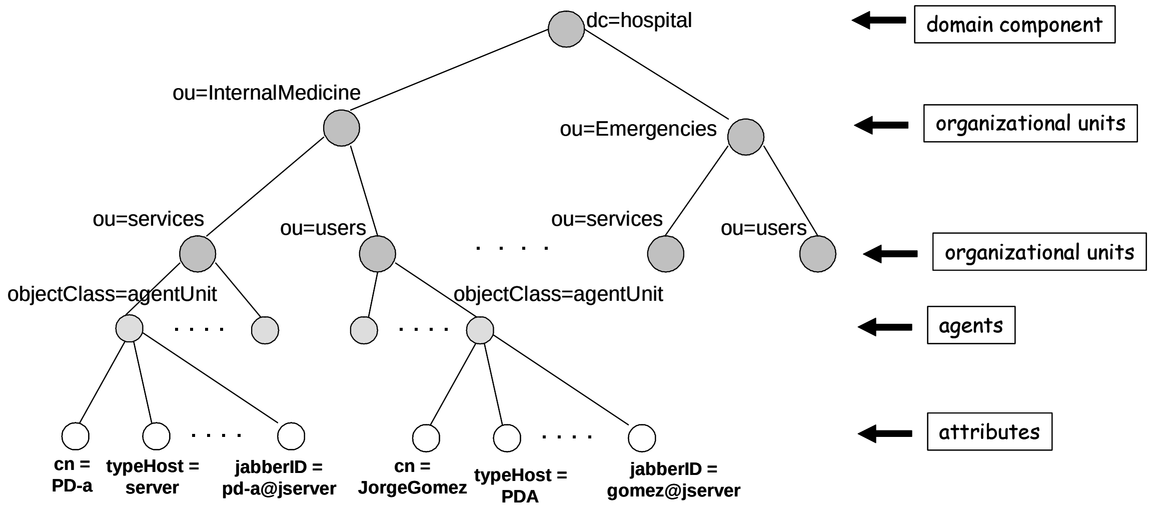


Figure 10. LDAP structure for the SALSA Agent Directory

An agent interacts with other agents and users through its Proxy to the Agent Broker, which is an instant messaging client that sends XML messages. The IM&P server stores the status of people and agents and notifies their changes to other agents subscribed to them. Developers can define new states according to the device or service the agent represents. For instance, an agent representing a printer may notify if it is busy or has a paper jam. This allows developers to provide a consistent interface for users to interact with other people, devices, and services [Favela *et al.*, 2002].

4.3.2 Agent Directory

SALSA provides a set of services that allow programmers to register and look for new agents added to the ubicomp environment in an Agent Directory. The implementation of the Agent Directory consists of a server that implements the Lightweight Directory Access Protocol (LDAP); and a SALSA agent (AD-proxy agent) acting as proxy to the Agent Directory.

The LDAP information model is based on entries. An entry is a collection of attributes that has a globally-unique Distinguished Name (DN). The DN is used to refer to the entry unambiguously. Each of the entry's attributes has a type and one or more values. The types are typically mnemonic strings, like "cn" for common name.

As illustrated in Figure 10 entries are arranged in a hierarchical tree-like structure. For the SALSA Agent Directory, it reflects the organizational boundaries of a ubicomp environment. Entries representing an organization (or the domain of the ubicomp system) appear at the top of the tree. Below them are entries representing the areas of an organization. In the next level are entries representing organizational units, which are the users and services/devices of the ubicomp environment. After this level, are the agents that represent users or services/devices. And finally the leaves of the tree represent the attributes of these agents. Figure 10 shows an example of an LDAP directory tree for the context-aware hospital information system illustrated in the scenarios presented in Chapter 3.

The SALSA API provides the facilities to register agents in the LDAP directory and for making specialized searches of agents. Thus, agents can communicate with the AD-proxy agent by using the communication protocol provided by SALSA. As illustrated in Figure 11 when an environment agent (Agent A) requests information from the AD-proxy agent, it converts the received SALSA message into an LDAP message. This is done through the XSL filter (eXtensible Stylesheet Language) of the AD-proxy agent. Then, the LDAP server retrieves the information requested and sends it to the directory agent which transforms this information into a SALSA message by wrapping the response to be delivered to Agent A.

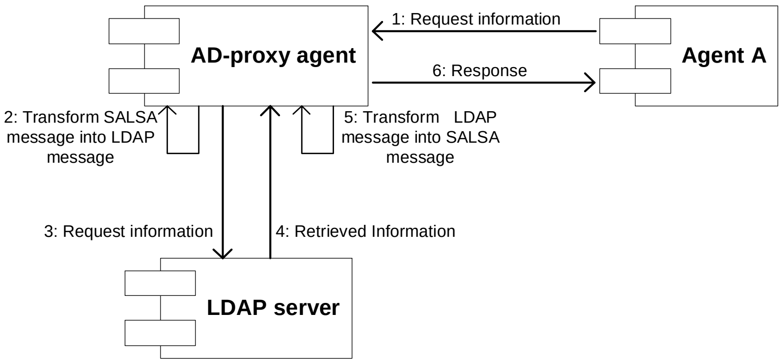


Figure 11. Agent Directory Components interacting to provide information to Agent A

4.3.3 SALSA class framework

The SALSA class framework provides a set of classes to facilitate the implementation of the internal architecture of an agent and control its life cycle. The collaborations of classes included in the package of the SALSA class framework, shown in Figure 9, represent the mechanisms to implement the agent's components, such as the classes for implementing passive and active perception, reasoning and action. The complete set of classes provided by SALSA are depicted in Figure 12 and explained in the following sections.

4.3.3.1 Agent perception

Two types of perception were identified for SALSA agents: active and passive. In active perception, an agent decides when to request or gather information from another agent or entity in the environment such as a sensor. In passive perception, the agent receives data without requesting it.

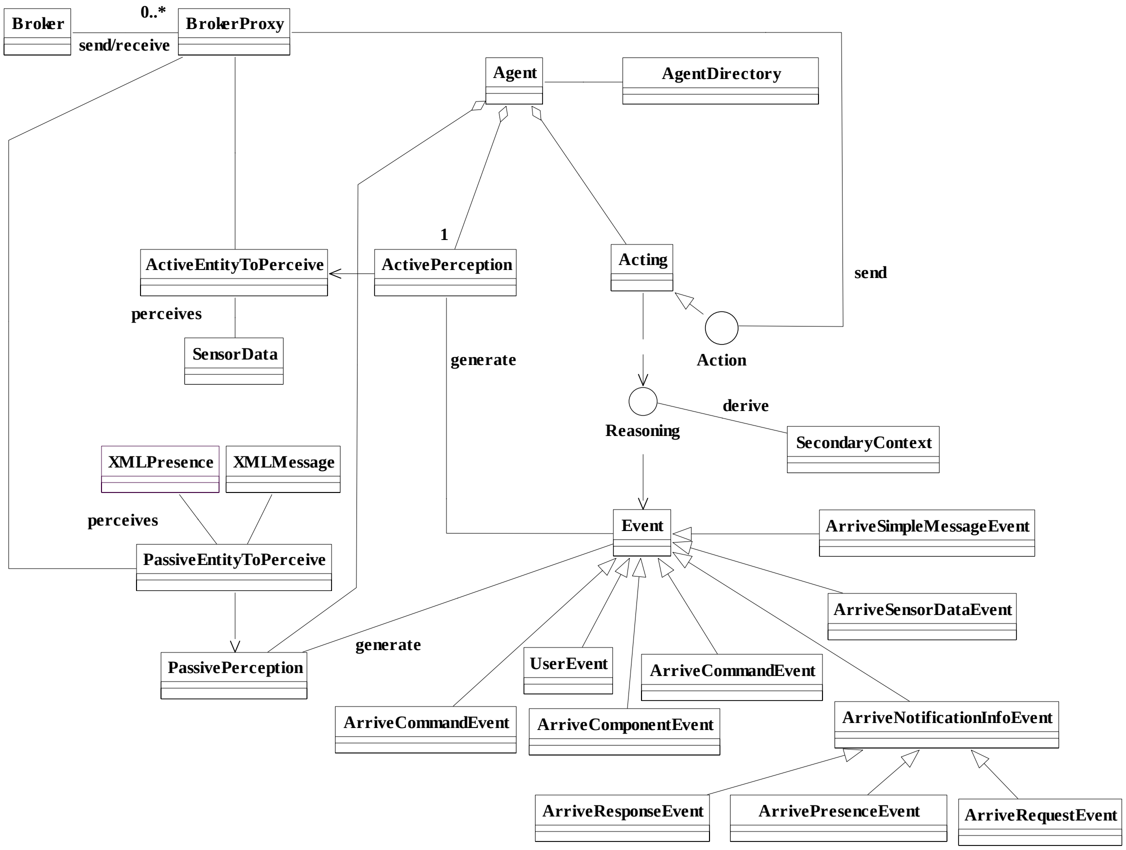


Figure 12. Class library of SALSA

As illustrated in Figure 9, the *Passive Perception* was implemented based on the Observer design pattern [Breemen, 2003]. This type of agent perception starts when a user, device or other agent sends data to an agent through the Agent Broker. In this case an agent has the role of observing the environment and acting according to the information received. Figure 13 shows the main classes for implementing this type of perception. The `PassiveEntityToPerceive` class represents the subject to be observed by the agent; and the `PassivePerception` class captures the information sent by the subject. For the active perception, an agent decides on its own when to sense an environment entity, and requests this information from another agent, or directly from a sensor's entity. This type of perception implements the Adapter design pattern. Figure 14 shows that the classes for implementing active perception are `ActiveEntityToPerceive`, which has the role of an Adaptee according to the Adapter design pattern. This class represents the environment entity that obtains data from a sensor or device. An agent decides when to perceive information by invoking the method `passivePerception.perceive()` from the `Action` object. Then `ActivePerception`, which has the role of Adapter, invokes the `ActiveEntityToPerceive` object, that is the interface to the sensor, in order to get data from a sensor or device (`activeEntityToPerceive.getInput()`).

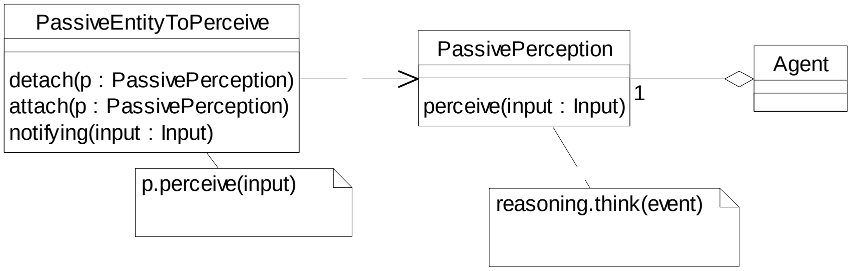


Figure 13. Passive Perception of a SALSA Agent

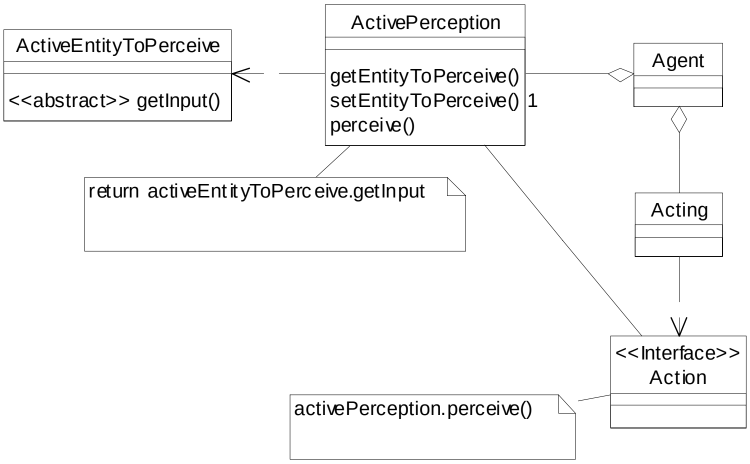


Figure 14. Active Perception of SALSA agents

When any of the perception components receive information, a SALSA event is generated indicating the type of information to the reasoning component as described in Table IV. In addition to this, a SALSA event also contains the perceived data, which can be an XML message received through the Agent Broker, or an object containing the data that was read directly from a sensor's interface.

Table IV shows in column 1 the events produced when information is perceived by an agent. The second column explains the type of information that was received and how the event may be produced. Column 3 describes the information contained in the object event, which can be an XML message or an object data. Column 4 presents the type of perception that can generate each type of event. For instance, the `UserEvent` is created when the user is interacting with the agent through its GUI (Graphical User Interface); this event can only be perceived directly through a passive perception as indicated in column 4 of Table IV. The only active perception supported by SALSA, is when the agent perceives data directly from a sensor or device. The passive perception of a SALSA agent, in which data is received through its IM client, is due to another agent that sends information by using the communication methods of the SALSA API as presented in column 5.

4.3.3.2 Agent reasoning

The information perceived by an agent is subtracted from the event by the reasoning component in order to be analyzed. The programmer, based on the logic of the agent, implements this component by using an appropriate reasoning algorithm, such as production rules, a neural network or case base reasoning. The Reasoning class contains the abstract method `think()` that should be implemented by the developer. Its implementation depends on the complexity of the agent's behavior. The reasoning component can use the facilities of SALSA to derive context information from the primary context information perceived by an agent. For this, SALSA provides the class `DeriveContext` which uses an XSL file as a filter that contains a set of rules to deduce secondary context from the data perceived by the agent. The set of rules of the XSL filter should be defined by the developer. Using these facilities of SALSA, developers need only indicate, to the `DeriveContext` class, the primary contextual variables and the name of the XSL file.

Thus, when an event is passed to the reasoning component, its derive context component is in charge of: detecting the type of event, extracting the data contained in the event, and verifying if the data is an expected contextual variable in order to be analyzed by the filter to check if some of the conditions have been met. Then, the derive context component returns an XML message to the agent's reasoning in order to indicate the inferred situation.

4.3.3.3 Agent action

To implement the action component, the framework provides the `Action` class with an abstract method that a developer should overwrite to specify how the agent must react. From, the action component, the agent can invoke the methods of communication provided by SALSA in order to collaborate with other agents. These methods are presented in Table V and SALSA communication protocol is explained in the next section. The acting component also enables the agent mobility. It was implemented based on the pattern Rc2s (request a component to a server) , which enables an agent to update its reasoning component by getting a copy of the reasoning algorithm from other agent residing on a server [Koukoumpetsos and Antonopoulos, 2002].

4.3.3.4 Agent communication

The messages sent among agents through the Agent Broker, are encoded using XML (eXtensible Markup Language). A number of researchers have suggested that ACL (Agent Communication Language) messages and its content ought to be encoded in XML because it offers several advantages over traditional ACLs, e.g. KQML and FIPA ACL, which are based on Lisp-like. Some of these advantages are [Labrou, 2001]:

- XML describes data in a human-readable format
- It is a database-neutral and device neutral format. Data marked up in XML can be targeted to different devices using the eXtensible Style Language (XSL).
- One can use off-the-shelf tools for parsing XML, instead of writing customized parsers to parse the ACL messages.

Table IV. Description of the SALSA events generated when information is perceived

1. Event	**2. Description**	**3. Wrapped information**	**4. Type of perception**	**5. Method used for communicating the perceived information**
`UserEvent`	Programmers may use this event to wrap an event produced when the user interacts with the user interface of the ubicomp system. For instance, to request information.	A GUI's event	Passive	
`ArriveRequestEvent`	Arrives a request of information from an agent	XML message:	Passive (through IM)	`sendResquest()`
`ArriveResponseEvent`	Arrives information that was previously requested by this agent	XML message:	Passive (through IM)	`sendResponse()`
`ArriveCommandEvent`	An agent is requesting to execute a functionality provided by this agent, such as a service	XML message:	Passive (through IM)	`sendCommandRequest()`
`ArriveNotificationInfoEvent`	Arrives information that was not previously requested	XML message:	Passive (through IM)	`sendNotificationInfo()`
`ArrivePresenceEvent`	Arrives a presence message indicating a change of state of others agents and users	XML message:	Passive (through IM)	`sendPresence()`
`ArriveSensorDataEvent`	Arrives data perceived from a sensor	An XML message: or `SensorData` object	Passive (through IM) or Active	`sendDataSensor()`
`ArriveSimpleMessageEvent`	Arrives a non-SALSA message. It could be defined by the programmer,	XML message:	Passive (through IM)	`sendMessage()`

Table V. Methods used for communication by SALSA agents

Method for requesting an action or service	**XML message sent by the SALSA method**
`sendCommandRequest()`	<message to=’**agentA**@server_jabber’ from=’agentB@server_jabber’> <x xmlns=’x:command’> <params><type>**TypeOfCommand** </type> **// TAGS DEFINED BY THE DEVELOPER** </params> </x> </message>
`sendResquest()`	<message to=’**agentA**@server_jabber’ from=’agentB@server_jabber’> <x xmlns=’x:request’> <params><type>**TypeOfRequest** </type> **// TAGS DEFINED BY THE DEVELOPER** </params> </x> </message>
`sendResponse()`	<message to=’**agentA**@server_jabber’ from=’agentB@server_jabber’> <x xmlns=’x:response’> <params><type>**TypeOfResponse** </type> **// TAGS DEFINED BY THE DEVELOPER** </params> </x></message>
`sendNotificationInfo()`	<message to=’**agentA**@server_jabber’ from=’agentB@server_jabber’> <x xmlns=’x:notificationInfo’> <params><type>**TypeOfNotification** </type> **// TAGS DEFINED BY THE DEVELOPER** </params> </x> </message>
`sendDataSensor()`	<message to=’**agentA**@server_jabber’ from=’agentB@server_jabber’> <x xmlns=’x:dataFromSensor’> <params><type>**TypeOfData** </type> **//SENSED DATA** </params> </x> </message>

- XML makes the ACL more WWW-friendly, which facilitates the development of software agents.
- Using XML will facilitate the practical integration with a variety of Web technologies.

XML incorporates links, which allow interfacing the ACL message to the knowledge repository that is the WWW.

The aim of the SALSA development framework is to use a friendly agent language taking advantage of XML to encode any kind of message. For defining the types of messages that can be communicated among SALSA agents, the Extensible Messaging and Presence Protocol (XMPP) of the IM server (www.jabber.org) was extended. SALSA provides developers with an API that facilitates the composing, sending, and receiving of messages between agents. However, the code for every content message type of the communicative act is left to the programmer, because it depends on the intent of the message generated by each agent in the ubiquitous environment.

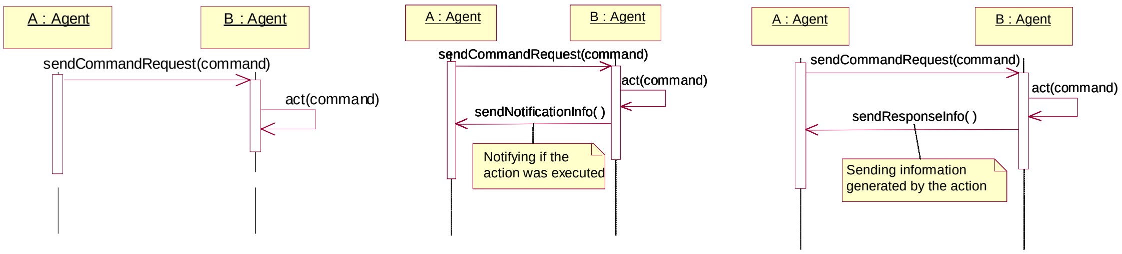
The API of SALSA for implementing the communication among agents consists of several methods that form the message that wants to be communicated. For instance, the `sendCommandRequest(mapa@serverJabber,”personalize”,marcerod)` method is used by an agent to request another agent to execute a specific action or service. When it is invoked, it will form an XML message (as the illustrated in Table V) with tags that specify to whom the message is addressed (mapa@serverJabber), the type of service requested (personalize), and the parameters needed to perform the action (userID). An example of this XML message is illustrated in Figure 15.

```
<message to=’ mapa@serverJabber @server_jabber’
            from=’agentB@server_jabber’>
           <x xmlns=’x:command’>
                <params><type>personalize </type>
                   // TAG INDICATING THE USER FOR WHOM PERSONALIZED
                       THE MAP
                       <user > marcerod </user>
              </params>
           </x>
    </message>
```

Figure 15. XML message requesting to personalize information to the Map agent

The communication protocol of SALSA enables agents to collaborate to reach a common goal, such as adapting and personalizing information for a user. This collaboration may involve one or more of the following actions:

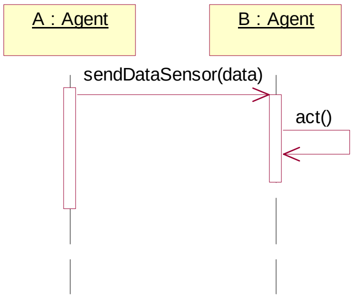
- *Negotiating for a service*. In this case, an agent (A) requires a service from another agent (B). As illustrated in Figure 16, agent A requests agent B to execute a service or a specific action. Agent B could respond by notifying agent A that the action was successfully executed or notifying that it can provide such service (Figure16 b)); or by providing information returned by the requested service (Figure16 c)).


A : Agent
B : Agent
sendCommandRequest(command)
act(command)
A : Agent
B : Agent
sendCommandRequest(command)
act(command)
sendNotificationInfo( )
Notifying if the action was executed
A : Agent
B : Agent
sendCommandRequest(command)
act(command)
sendResponseInfo( )
Sending information generated by the action

a) b) c)

Figure 16. SALSA's communication methods for requesting the execution of a service

- *Requesting information.* An agent (A) requires information from another agent (B). As depicted in Figure 17, agent B can respond by sending the requested information to agent A (Figure 17 a)); or by sending a notification that it can provide this information (Figure 17 b)).



a) **b)**

Figure 17. SALSA's communication methods for requesting information

- *Notifying information perceived from a sensor or device.* Agents that act as proxies to devices or sensors, get information from them and may require to communicate with other agents of the environment to be processed. In Figure18, agent A sends information perceived from a sensor to agent B.

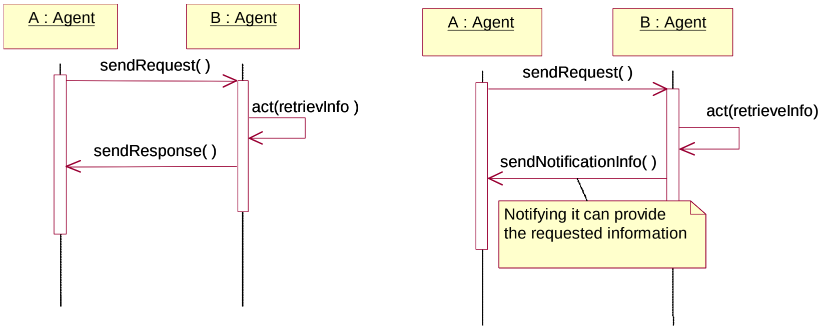


Figure 18. SALSA's communication methods for sending information perceived from a sensor

- *Notifying the presence and status of agents.* An agent notifies a change of presence or status to other agents subscribed to its presence. This enables agents to be aware of the available services of the ubicomp environment. Agents also received the presence of the users in the ubicomp environment.

4.3.3.5 Agent initialization and registration

SALSA provides the class `AgentDirectory` which includes methods for registering an agent on the Agent Directory, and for making different types of searches on it. By using the method `register()`, a developer indicates how the agent will register its attributes on the Agent Directory. This allows developers to update the agent's attributes without having to manually configuring the records on the Agent Directory. Developers can define the SALSA attributes for an agent with the methods contained in the class `AgentAttributes` which are presented in Table VI.

The current searches supported by the proxy-agent to the Agent Directory (AD) are: looking for agents available in the ubicomp environment, searching for agents available in a specific area of the ubicomp environment, and looking for a specific service in the environment or in an area. These methods are described in Table VII, and can be invoked by any of the agent's components. Thus, while an agent is executing, it can look for information regarding other agents with whom it needs to interact or it can look for information about itself. For instance, an agent representing a service may require publishing an image that represents it on the map of the ubicomp area showed on a public display; then the agent requests this information from the proxy-agent of the AD.

Table VI. Attributes for a SALSA agent

Attribute	Description	SALSA methods
`typeAgent`	Type of agent: represents a user or service	`setTypeAgent(String type)`
`Name`	Name assigned to the agent during the design phase. For instance, LE-a, KM-a, or Ca-a.	`setName(String name)`
`Jid`	Identification of the agent on the Instant Messaging server (Jabber). For instance: LEagent@jabberServer	`setJabberID(String jid)`
`description`	Brief description of the agent	`setDescription(String description){`
`typeHost`	Type of host on which the agent resides: PC, Table PC, PDA, mobile phone, etc.	`setTypeHost(String typeHost)`
`urlImage`	Link to the image used for representing an agent. Agents that represent services or devices may require being represented by an image on a GUI of the ubicomp system.	`setUrlImage(String urlImage)`
`Area`	Department of the organization	`setArea(String area)`
`Floor`	Floor of the building in which a department of the organization is located	`setFloor(String floor)`

4.4 Discussion

This Chapter has presented the requirements of the SALSA middleware which were defined when addressing the design issues raised in the scenarios that describe how autonomous agents can be used as the main components of ubiquitous computing systems. These requirements were addressed during the design phase of the middleware. The implementation of the SALSA API was introduced in this Chapter, and more details regarding the API can be found in Appendix A.

The SALSA API was initially implemented in Java, which enabled agents to be executed on any computing platform. However, it was observed that this version of SALSA does not enable agents to access the native libraries of Widows CE, therefore a sub-set of the SALSA API was implemented in C#, called micro-SALSA (mSALSA), which enables developers to create the components of the agents and use the SALSA communication protocol. Developers have the option of programming in any of these languages and take advantage of the programming facilities offered by each of them.

The following Chapter illustrates how SALSA facilitates the development of an ubicomp system by revisiting one of the scenarios presented in Chapter 3.

Table VII. Methods of the `AgentDirectory` class for requesting information from the Agent Directory proxy-agent (AD-pa)

Methods	**Description**
`lookForAllAgents()`	By invoking this method, an agent requests from the AD-pa, information about all agents available in the ubicomp environment
`lookForAllAgents(String area)`	An agent requests from the AD-pa, information about all agents available in a specific area of the ubicomp environment
`lookForAservice()`	An agent requests from the AD-pa, information about all agents offering a specific service available in the ubicomp environment
`lookForAservice(String area)`	An agent requests from the AD-pa information about all agents offering a specific service available in a specific area of the ubicomp environment

Chapter 5

Creation and evolution of ubicomp systems with SALSA

To illustrate the flexibility of SALSA for allowing the progressive development of ubiquitous computing systems, this Chapter explains how an autonomous agent can be implemented with SALSA, and then, how this agent can be easily integrated to a ubiquitous computing system. For this, the following sections describe the implementation of the location-estimation agent and its integration into the context-aware hospital information system (CHIS) which was introduced in scenario 3.3.1. Finally, it is presented how the CHIS functionality was extended to support context-aware public displays by revisiting the scenario 3.4.1.

5.1 Estimating mobile user's location

Estimating the location of a user has been a subject of considerable attention in recent years. Of particular interest are location estimation techniques that make use of an existing wireless LAN (Local Area Network) infrastructure, since they have better scalability and less installation and maintenance costs than ad-hoc solutions. These methods measure the RF (Radio Frequency) signal strength between a mobile device and several access points of the wireless LAN to estimate location [Bahl and Padmanabhan, 2000].

To estimate the location of the mobile users of the CHIS system, we used an autonomous agent that wraps in its reasoning component a neural network trained to map RF signals from a WLAN to 2D coordinates. Once trained, the neural network was used to classify incoming patterns into labeled classes, X,Y coordinates. More details of the neural network implementation are explained in [Rodriguez *et al.*, 2004].

5.1.1 Implementation of the location-estimation agent

Figure 19 represents the main components of the LE-a and how it is attached to the location-aware client. The figure represents as a circle the interface from which an agent perceives information and as a receptacle the perception component. This notation helps to clearly visualize how an agent may perceive information directly from a device/sensor or from another agent. For instance, the LE-a has one entity to perceive the SNR, while the location-aware client (which is also an agent) has an entity to perceive the user's position sent by the LE-a and a second entity to perceive information from other agents through an IM client, through which this agent can also communicate with other agents as part of its action plan. According with the perceived information the LE-a decides what action to execute. Thus, the agent decides to estimate the user's location based on the SNR received from three access points, or to update its reasoning component when the SNR decays considerably from floor to floor. It was considered that one location-estimation agent had to be trained for each hospital floor in order to calculate the current user's location. When the strength of the signal from the access points in one floor goes below certain threshold, the LE-a asks the Agent Directory to request information of the server in which resides the LE-a that holds the trained neural network for that floor. Thus, by using the mobility attribute of agents, the LE-a on the PDA can update its reasoning component.

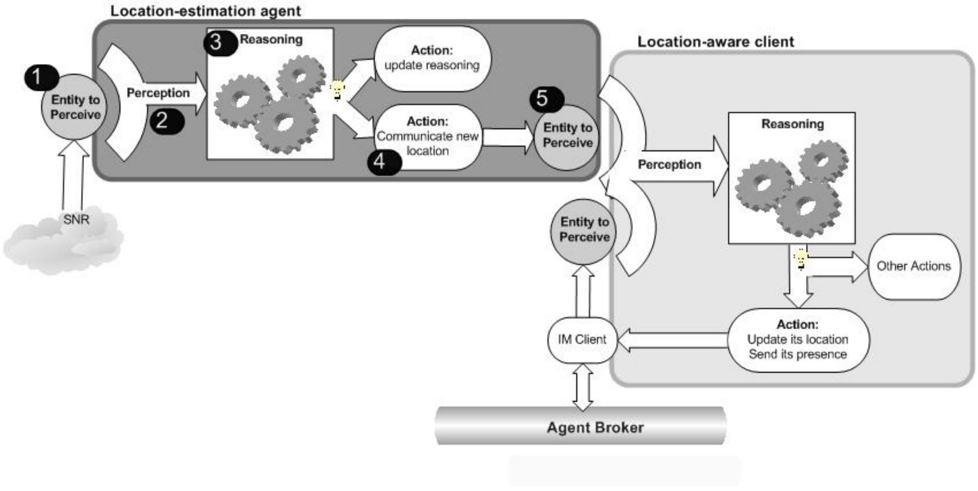


Figure 19. Components of the Location-estimation agent and the Location-aware client

Figure 20**Figure**a) shows a class diagram based on the AUML (Agent Unified Modeling Language) notation that illustrates the design of the location-estimation agent (LE-a). At the top of the diagram is specified the information that define the state of the agent: the perceived information (SNR- Signal to Noise Ratio) and the output information (coordinates X,Y). After that, the diagram defines the actions of the agent; i.e. notifying the estimated user's position to the location-aware client, for which the LE-a uses the SALSA communication protocol (`sendDataSensor()`) as indicated at the diagram bottom. Figure 20 b) shows how the agent's components for perceiving, reasoning and acting, interact to estimate the user's location. These components are instances of SALSA classes as explained in the following sections.

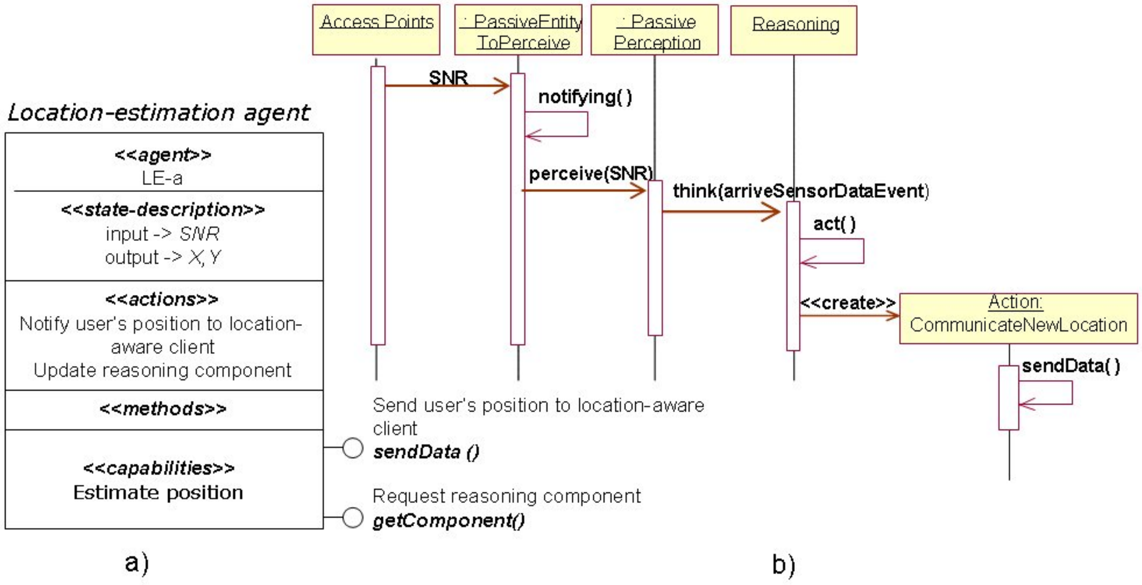

Location-estimation agent
<<agent>>
LE-a
<<state-description>>
input -> SNR
output -> X, Y
<<actions>>
Notify user's position to location-aware client
Update reasoning component
<<methods>>
<<capabilities>>
Estimate position
Send user's position to location-aware client
sendData ()
Request reasoning component
getComponent()
a)
Access Points
: PassiveEntity ToPerceive
: Passive Perception
Reasoning
SNR
notifying( )
perceive(SNR)
think(arriveSensorDataEvent)
act( )
<<create>>
Action: CommunicateNewLocation
sendData( )
b)

Figure 20. AUML class diagram (a), and sequence diagram of the Location-estimation agent (b)

5.1.1.1 Perception Component

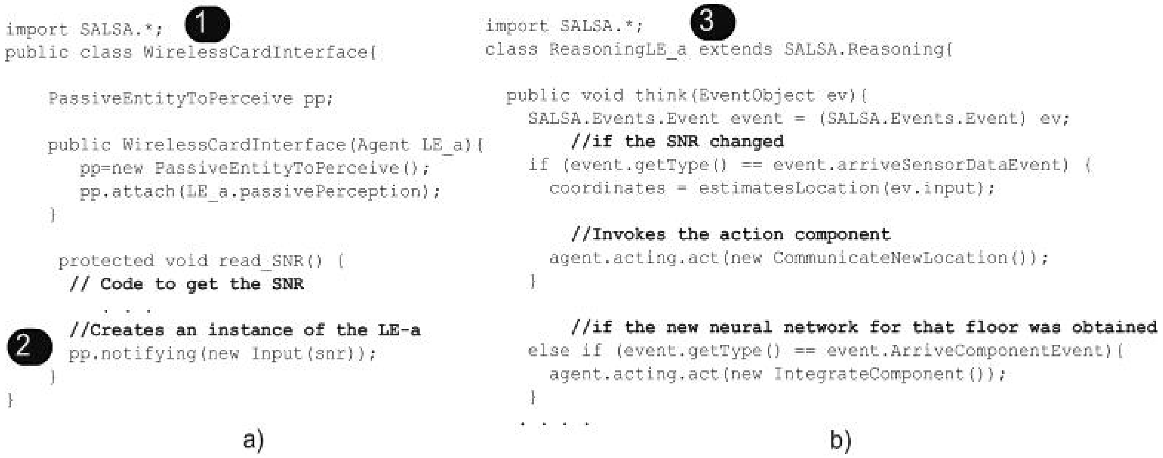
The agent's perception module receives the SNR (Signal to Noise Ratio) through the `PassiveEntityToPerceive` object, which represents the memory of the wireless network (WLAN) card. The developer implemented this interface to read data from the

WLAN card, wrap the data in an `Input` object, and then notify it to the `PassivePerception` component of the LE-a. When the SNR value is changed, the `PassivePerception` object generates an `arriveSensorDataEvent` which is passed to the reasoning component.

Figure 21a) shows the pseudo code for implementing the perception component of the LE-a using the SALSA classes. The code was numbered to illustrate what element of the LE-a is being implemented according to Figure 20. Thus, element ❶ of the LE-a is the entity for perceiving information. This is the `WirelessCardInterface` that contains the code for reading the SNR from the wireless LAN card. This class has embedded a `PassiveEntityToPerceive` object (`pp`) which has a reference to the LE-a. Thus, when a new SNR is read, it is passed to the passive perception component (❷) by invoking the method `pp.notifying().` Thus, developers need to implement just the interface that reads the data from a device/sensor and to use the SALSA classes to connect the interface with the perception component, which is automatically activated when an instance of the `Agent` class is created.

5.1.1.2 Reasoning component

As illustrated in Figure 21 b), the `ReasoningLE` class (❸) is specialized from the `Reasoning` abstract class of SALSA. Its `think` method was overwritten to process the perceived input and then, to indicate to the agent what action should be executed. If the received SALSA event was of type `arriveSensorDataEvent`, it indicates that a new estimation of the user's location has to be calculated. Then, the `estimatesLocation()` method is invoked, which implements the trained Neural Network. Providing an abstract class to implement the reasoning component enables programmers to easily update or replace its logic with another algorithm that may be more efficient.



```
import SALSA.*;
public class WirelessCardInterface{

    PassiveEntityToPerceive pp;

    public WirelessCardInterface(Agent LE_a){
       pp=new PassiveEntityToPerceive();
       pp.attach(LE_a.passivePerception);
    }

     protected void read_SNR() {
      // Code to get the SNR
         . . .
      //Creates an instance of the LE-a
      pp.notifying(new Input(snr));
    }
}
```

```
import SALSA.*;
class ReasoningLE_a extends SALSA.Reasoning{

  public void think(EventObject ev){
    SALSA.Events.Event event = (SALSA.Events.Event) ev;
        //if the SNR changed
    if (event.getType() == event.arriveSensorDataEvent) {
      coordinates = estimatesLocation(ev.input);

        //Invokes the action component
      agent.acting.act(new CommunicateNewLocation());
    }

        //if the new neural network for that floor was obtained
    else if (event.getType() == event.ArriveComponentEvent){
      agent.acting.act(new IntegrateComponent());
    }
  . . . .
```

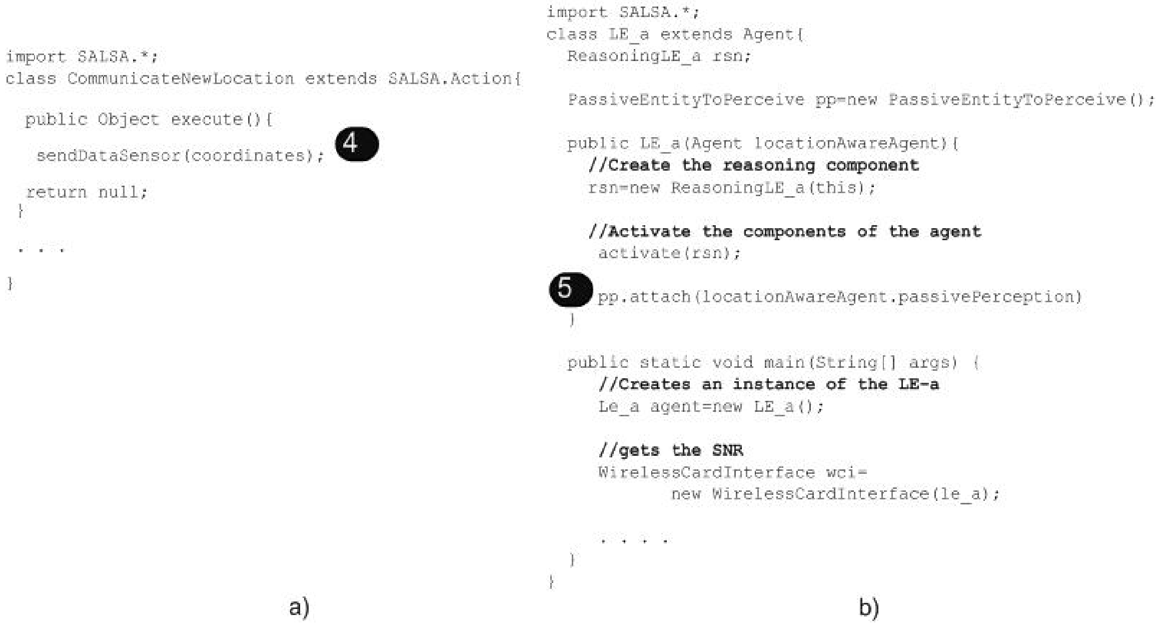
Figure 21. a) Code for implementing the entity for perceiving information from the WLAN card. b) Code of the reasoning component

5.1.1.3 Action component

When a new user's location is estimated, the reasoning component decides to communicate it to the location-aware client which is also executing in the handheld computer. Figure 22 a) presents the code for implementing the LE-a communication with the location-aware client. To do this, the `execute()` method of the abstract class `Action` was overwritten to invoke the SALSA method `sendDataSensor()`(❹).

5.1.1.4 Creation of the agent

Finally, once the Perception, Reasoning, and Action components of the agent were implemented, the main body of the agent had to be created. This was done by extending the SALSA `Agent` class (see Figure 22 b)). Thus, when an instance of this agent is created, its components are activated and the life cycle of the agent begins.



```
import SALSA.*;
class CommunicateNewLocation extends SALSA.Action{

  public Object execute(){

   sendDataSensor(coordinates);

  return null;
 }

 . . .

}
```

a)

```
import SALSA.*;
class LE_a extends Agent{
  ReasoningLE_a rsn;

  PassiveEntityToPerceive pp=new PassiveEntityToPerceive();

  public LE_a(Agent locationAwareAgent){
    //Create the reasoning component
    rsn=new ReasoningLE_a(this);

    //Activate the components of the agent
     activate(rsn);

     pp.attach(locationAwareAgent.passivePerception)
  }

  public static void main(String[] args) {
     //Creates an instance of the LE-a
     Le_a agent=new LE_a();

     //gets the SNR
     WirelessCardInterface wci=
            new WirelessCardInterface(le_a);

     . . . .
  }
}
```

b)

Figure 22. Code of the action sending data to the location-aware agent a). Code of the LE-a creating its components b)

5.2 Context-aware Hospital Information System (CHIS)

With SALSA we built the context-aware hospital information system (CHIS) with the aim to support the activities of the hospital staff [Favela *et al.*, 2004; Muñoz *et al.*, 2003]. CHIS is a handheld-based system that enables users to locate relevant documents, such as patient's records and laboratory results; locate patients and colleagues; and locate and track the availability of devices such as medical equipment, and other computational resources such as public displays.

5.2.1 Architecture of CHIS

Figure 23 presents the design of the system architecture illustrating the main nodes in which the system's components are executing. These components are SALSA agents communicating through the Agent Broker (an IM server), as specified by the SALSA communication infrastructure.

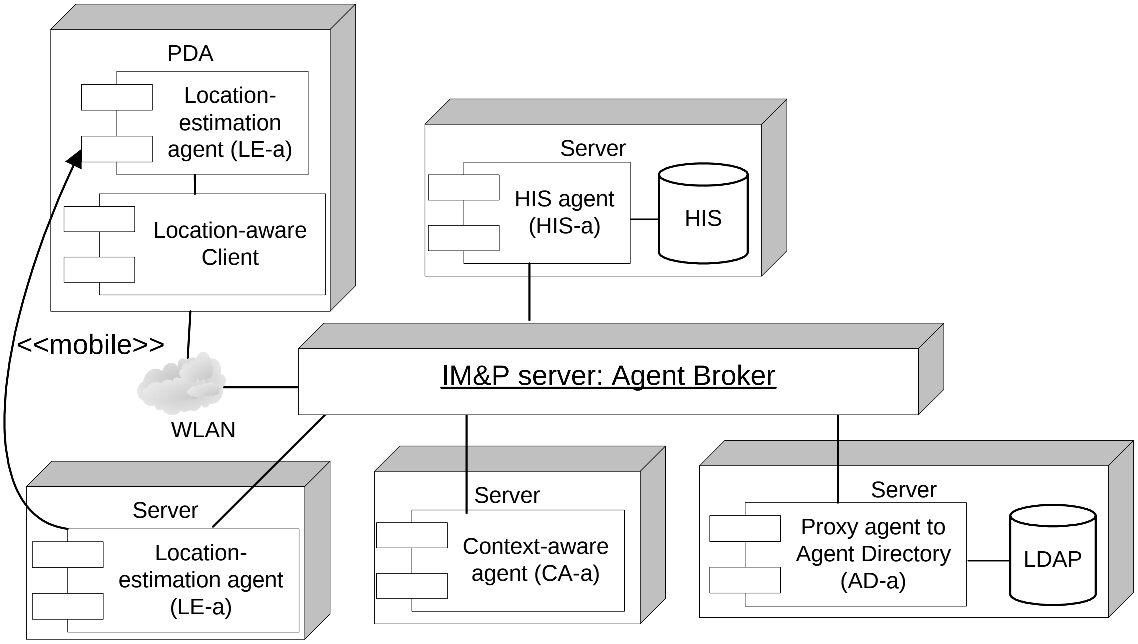

PDA
Location-estimation agent (LE-a)
Location-aware Client
<<mobile>>
WLAN
Server
HIS agent (HIS-a)
HIS
IM&P server: Agent Broker
Server
Location-estimation agent (LE-a)
Server
Context-aware agent (CA-a)
Server
Proxy agent to Agent Directory (AD-a)
LDAP

Figure 23. Architecture of the context-aware hospital information system

In the handheld computer resides the Location-aware client which notifies the user's location to other users and agents, provides mobile users with information relevant to their location, and communicates with other members of the hospital staff. Its interface is based on the IM paradigm, through which users are notified of the availability of other users and their location. In the handheld also resides the location-estimation agent (LE-a) with the purpose of obtaining the user's position (X,Y coordinates), and informs it to the location-aware client. By using the mobility attribute of agents, the LE-a on the PDA can update its reasoning component by getting the reasoning component from the server in which resides the LE-a that holds a trained neural network for a specific building's floor. The Agent Directory provides information of the agents available in the environment. For instance, the LE-a can know in which server resides the agent containing the trained neural network by communicating with the agent acting as proxy to the Agent Directory (AD-a).

The hospital information system agent (HIS-a) acts as proxy of the HIS that manages and stores the patient's clinical records and other data relevant to the hospital. This agent provides access to information contained in the HIS, and monitors its changes. Finally, the Context-aware agent (Ca-a) is the system's component that sends the messages that depend on contextual variables for their delivery, such as the recipients, location and role.

5.2.2 Implementation of CHIS

This section revisits the scenario 3.3.1 to explain the functionality of the context-aware hospital information system as illustrated in Figure 24.

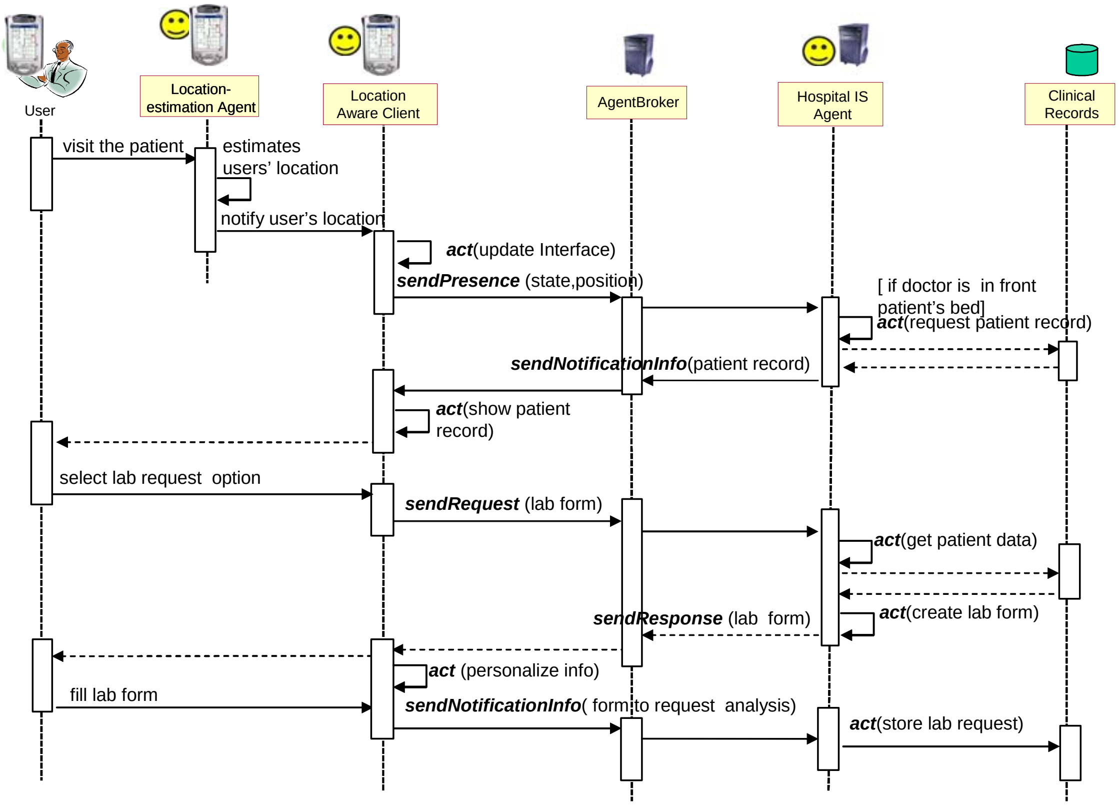


Figure 24. Agents of CHIS interacting for enabling the physician to access the patient's clinical record

When the Location-aware client perceives that Dr. Diaz is in front of one of his patients, it acts by updating its user interface to show his location (`act(update interface)`) and notifying to all users and agents subscribed to the presence of the user (`sendPresence()`). Then the HIS-a verifies if its contextual conditions match the perceived context, i.e. the user's role and location. As Dr. Diaz is in front of one of his patient's bed, the HIS-a retrieves the patient's clinical record (`act(retrive patient record)`) and sends it to the Location-aware client (`sendNotificationInfo()`) in order for the physician can analyze the patient's health-condition.

After consulting both the record and the patient, Dr. Diaz decides to request a lab analysis (`sendRequest(lab form)`) from the HIS-a. This agent creates the lab form, which includes some of the patient's data, such as his name and bed number, and sends it to the context-aware client to be displayed for Dr. Diaz (`sendResponse(lab form)`). The location aware client customizes (`act(personalize info)`) the lab form for the doctor based on his identity (Dr. Diaz), role (physician) and current location (bed 222). Then, he fills the lab form and sends it to the Hospital IS to be added to the patient's clinical record (`sendNotificationInfo(form to request analysis)`).

Finally, the HIS-a will inform the chemist of the patient analysis to be performed and the medical samples to be taken. When, the chemist adds the lab analysis to the patient's clinical record, the HIS-a will notify the test results to the doctor who will make a diagnosis of the patient's health condition.

5.2.3 Integrating the location-estimation agent to the context-aware hospital information system

To integrate the LE-a to the context-aware hospital information system, it was modified the location-aware client. Initially the user explicitly specified his location on the GUI of the location-aware client. As indicated in the pseudo code of Figure 22 b), in order for the location-aware client to perceive the user's X,Y coordinates from the LE-a, a `PassiveEntityToPerceive` object was created in the location-estimation through which the passive perception of the Location-aware agent (❺) was attached to the LE-a. As illustrated in Figure 19, the location-aware client has two receptacles for perceiving information: one for receiving data from the LE-a, and another one for receiving data contained in XML messages sent by other agents through the Agent Broker. The reasoning and action components were not modified during the integration process of the LE-a to the context-aware system for a hospital environment.

Figure 25 illustrates the interactions among the components of CHIS which includes the location-estimation agent. When Dr. Diaz visits his patient, then the SNR to the access points change. This is perceived by the LE-a (`perceive(SNR)`), which estimates the user's location based on its trained neural network (`think(estimate user’s position)`). Thus, the agent obtains the user's position (X,Y coordinates) which is communicated to the location aware client (`sendDataSensor()`). The location aware client translates the X,Y coordinates to an id of the place in which is located the user (i.e. bed number, room). Thus, the reasoning component maps the user's X,Y coordinates to an area identifier (`location: think()`) and finally, its acting component communicates the user's location to the rest of the system's agents and users (`sendPresence(state,location)`). Figure 25 shows the architecture of CHIS including the location-estimations agent**.**

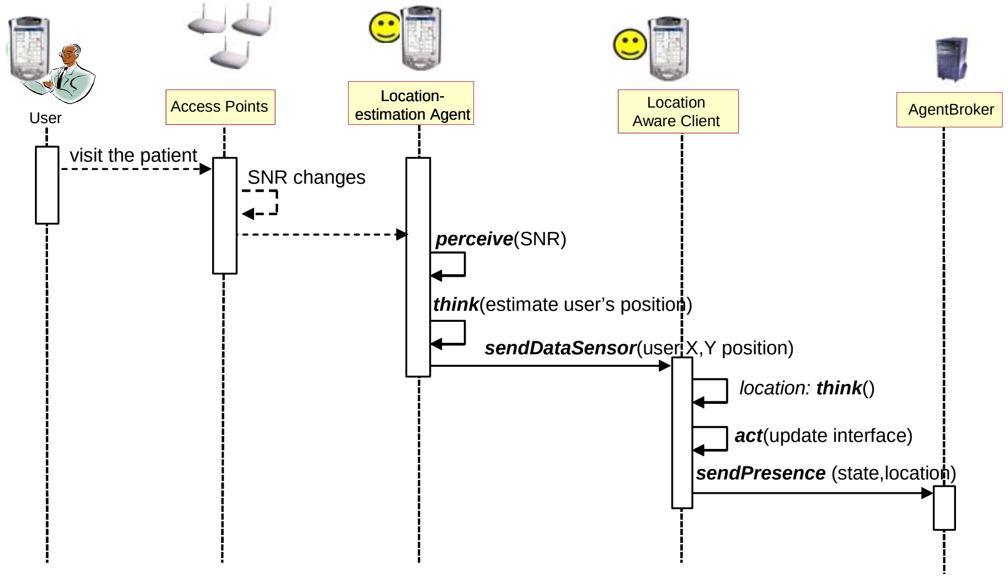


Figure 25. Location-estimation agent interacting with the agents of the context-aware hospital information system

5.3 Extending the functionality of CHIS through context-aware public displays

As explained by scenarios of Section 3.4, the system was extended to integrate context-aware public displays which can be used to visualize medical information, to support coordination and collaboration of the hospital staff [Favela *et al.*, 2004].

5.3.1 Extended architecture of CHIS

Figure 26 presents the architecture of CHIS incorporating new agents to support users' activities through context-aware public displays. In the display server node resides several agents offering several services for the medical personnel. These services were assigned implemented as SALSA agents. Thus, the responsibility of the Public Display agent (PD-a) is to act as a proxy to a public display available at the hospital. This agent enables users to access the public display and have control over the applications displayed. The responsibility of presenting a personalized map of the hospital floor was delegated to the map agent (Map-a). It displays a map of the hospital floor that indicates the location of the hospital staff, available services, and highlights the beds of patients assigned to the current physician using the display. The Map-a also shows messages addressed to the user, i.e. messages related to his patients that may indicates additions to their electronic records. Finally, the knowledge management agent (KM-a) is responsible for displaying the hospital medical guide and previous cases relevant to the case being consulted on the public display.

5.3.2 Implementation of the context-aware public displays

By revisiting scenario 3.3.1, Figure 27 depicts a sequence diagram illustrating how the autonomous agents of CHIS interact by using the SALSA communication protocol (the SALSA methods are in bold font style on the diagram). In this scenario, the HIS-a perceives a change on the hospital information system and notifies it to Dr. Garcia by sending a message (`sendNotificationInfo`) indicating that the X-ray results of patient on bed 225 are available. When the location-aware client receives this message, it will act by updating its instant messaging interface. Dr. Garcia approaches the nearest public display when finished with her current patient and before visiting patient 225.

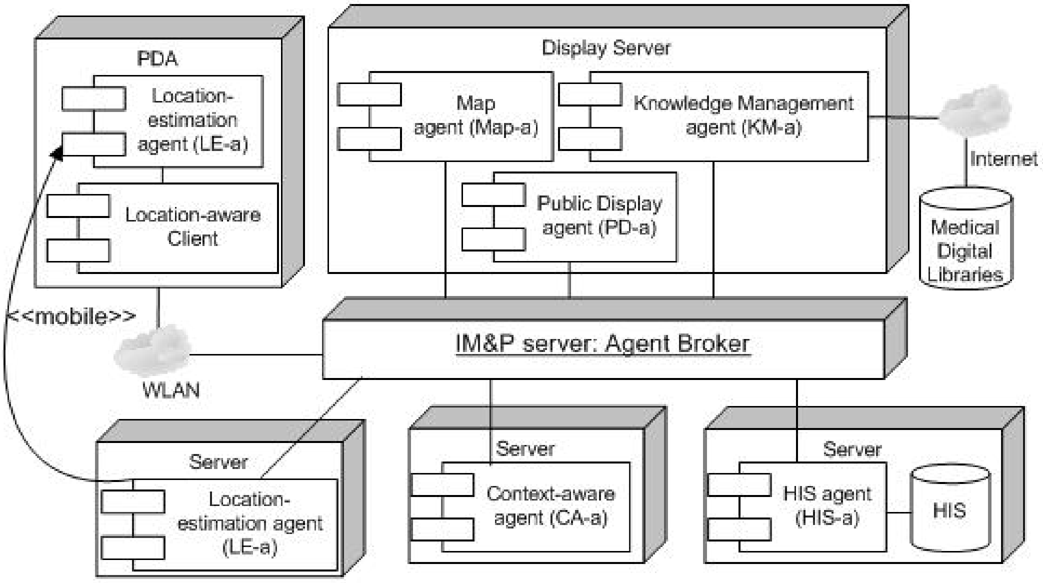

PDA
Location-estimation agent (LE-a)
Location-aware Client
<<mobile>>
WLAN
Display Server
Map agent (Map-a)
Knowledge Management agent (KM-a)
Public Display agent (PD-a)
Internet
Medical Digital Libraries
IM&P server: Agent Broker
Server
Location-estimation agent (LE-a)
Server
Context-aware agent (CA-a)
Server
HIS agent (HIS-a)
HIS

Figure 26. Architecture of the context-aware hospital information system

The doctor's location, which is constantly being tracked by the location-estimation agent on her PDA, is notified (`sendPresence`) to all users and agents in the environment by the Location-aware client. The Public Display agent acts by displaying the user's photograph, indicating with this that the user has been logged into the system. Then, the PD-a requests (`sendCommandRequest`) to the Map-a that personalizes the map application for Dr. Garcia. Finally, the PD-a also requests (`sendRequest`) the contextual messages recently received to the location-aware client, and display them on the floor map. Thus, the physician can continue accessing the records of his patients and other medical information by interacting with the public display.

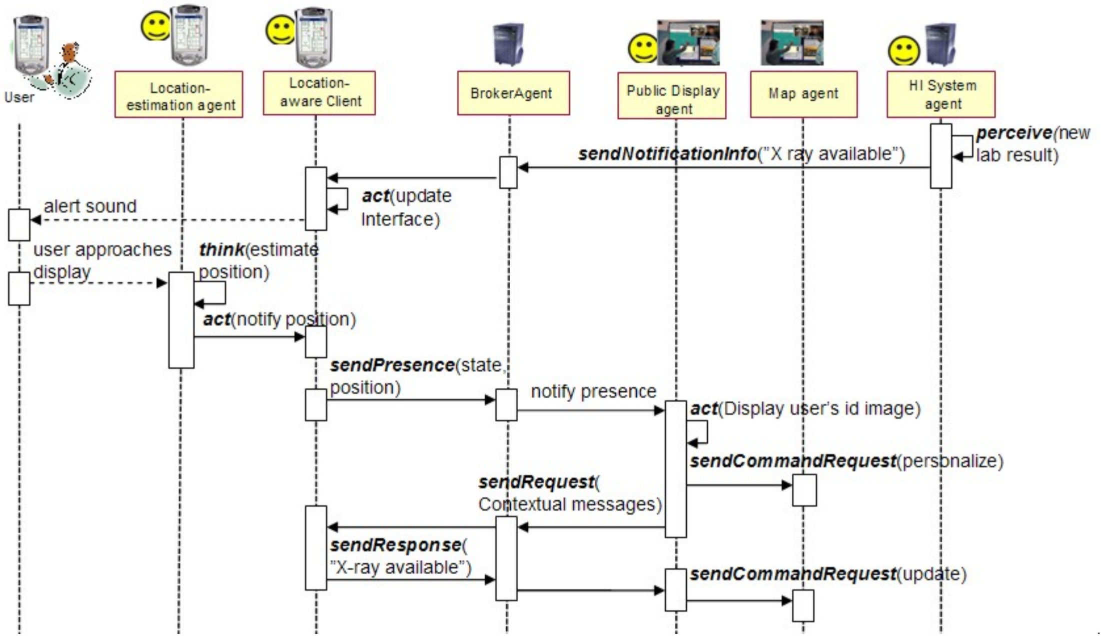

User
Location-estimation agent
Location-aware Client
BrokerAgent
Public Display agent
Map agent
HI System agent
perceive(new lab result)
sendNotificationInfo("X ray available")
act(update Interface)
alert sound
user approaches display
think(estimate position)
act(notify position)
sendPresence(state, position)
notify presence
act(Display user's id image)
sendCommandRequest(personalize)
sendRequest(Contextual messages)
sendResponse("X-ray available")
sendCommandRequest(update)

Figure 27. Implementation of CHIS using the SALSA development framework

5.4 Conclusions

The main components of the context-aware hospital information system are agents that respond autonomously in accordance with the context surrounding the activities performed at the hospital. SALSA was used to implement this system since it offers facilities to easily implement and add functionality to ubicomp systems. As the architecture of SALSA is based on the instant messaging (IM) paradigm, it allows a standardized form of interaction among users and services represented by autonomous agents. In the same form, users are aware of the presence of other users, they are also aware of the presence and state of the environment's services and devices. The scalability of a system implemented with SALSA is enabled by the IM server used as an Agent Broker, since it scales to a high volume of streaming XML connections serving hundreds of thousands of simultaneous users and agents. By using SALSA, developers may easily integrate any mechanism of reasoning; or change the current reasoning algorithm for other, without modifying the code of the rest of the agent's components. For instance, we may change the neural network of the Location-estimation agent by a nearest neighbor algorithm without altering the perception and acting component. With SALSA, it can be created different types of agents, such as personal agents and service agents that may have attributes of autonomy, mobility, reactivity and collaboration. SALSA agents have well-defined interfaces to interact with their environment, and mechanisms to encapsulate its implementation. For this reason, these agents may be considered as units of independent deployment or components, which may be re-used or integrated to any SALSA ubiquitous system. For instance, the location-estimation agent may be integrated to a different environment than a hospital by just training its neural network for this new physical environment.

Chapter 6

Evaluation

The goal of evaluating SALSA was to find out if SALSA facilitates the development of ubiquitous computing systems. The process followed for evaluating SALSA involved studying existing evaluation methodologies that could be used or adopted for evaluating SALSA. This investigation led to the definition of the criteria for evaluating SALSA and to state a set of hypothesis based on what SALSA offered to developers of ubicomp systems. Different methods were followed for evaluating one or more of the hypothesis. For instance, the ease of use of SALSA was evaluating through an in-lab experiment in which several users completed a programming exercise, while the use of autonomous agents as a design abstractions for ubicomp systems was evaluated with a design exercise. This chapter explains this criterion, the hypothesis stated, the methods followed for evaluating these hypothesis, and finally, the results and conclusions of evaluating SALSA.

6.1 Evaluation criteria and hypothesis

Not much research has been published on evaluating infrastructures (such as middleware, toolkits, or APIs) that support the development of systems. Software Engineering evaluation methods may be adapted to evaluate performance and reliability when producing an application. However, these metrics do not address the end-user experience of software development. They do not provide evidence of the infrastructure’s ease of use or its usefulness for implementing systems within a specific domain. Inspired by empirical studies for evaluating infrastructures used for implementing interactive systems [Klemmer *et al.*, 2004; Grimm, 2004] several desirable properties were identified that are relevant for evaluating a middleware:

Utility. “This criterion determines whether others can build real pervasive applications atop the infrastructure. After all, system architecture is only as useful as the programs running on top of it” [Grimm, 2004]. To evaluate whether SALSA met this criterion, the following questions were posed.

Can we build real ubicomp applications with SALSA?

Does SALSA support a variety of useful systems? If so, what are these useful systems?

The methodology followed to determine the functional requirements of SALSA was based on selecting scenarios which were used to envision how ubicomp technology along with autonomous agents support users' activities. As presented in Chapter 3, these scenarios became our vision of how ubicomp technology can support the activities of real users working in hospitals. The selected scenarios were used to propose how autonomous agents can be used as the main constructs of a ubicomp system. This leads to the first hypothesis:

Hypothesis 1:

"SALSA is useful since it is suitable for implementing useful ubicomp applications"

Completeness. "This criterion determines whether the architecture is sufficiently powerful and extensible to support interesting user-space programs" [Grimm, 2004]. Middlewares, frameworks and toolkits are developed with the goal of facilitating the development of applications that share the same characteristics, such as distribution of components. Implementing a ubicomp application involves coping with other complexities such as heterogeneity of devices, providing context-aware services, and issues related to human-computer interaction. To determine the completeness of SALSA, the following questions had to be answered:

- How does SALSA allow developers to deal with the complexities associated with the development of ubicomp systems?
- Does SALSA enable the evolution of ubicomp applications?

This last question led to state hypothesis 2:

Hypothesis 2:

"SALSA is flexible enough to enable the evolution and iterative implementation of ubicomp applications".

Ease of use. "Programming languages and toolkits should be evaluated on how readable the programs that are written with the programming language are to other programmers, how learnable they are, how convenient they are for expressing certain algorithms, and how comprehensible they are to novice users" [Klemmer *et al.*, 2004]. SALSA was developed with the aim of facilitating the implementation of ubicomp environments by means of the use of autonomous agents. However, during the development of SALSA, the question arose as to why it was necessary to provide another API for agents, and what differences and advantages SALSA offered with respect to other agent frameworks. Thus, it was determined that to evaluate the ease of use of SALSA, the following questions had to be addressed:

- How learnable is the SALSA framework?
- How comprehensible is SALSA for users who are not familiar with agents?
- How easy is it to modify the reasoning component of agents and to add new functionality to the ubicomp system by means of the use of SALSA agents?
- How convenient is SALSA for expressing the agent's pro-activity, reactivity and collaboration?

To evaluate the ease of use of the SALSA class framework, the following hypotheses were stated:

Hypothesis 3:

SALSA is considered by programmers as easy to learn even when they do have no previous knowledge of agent technology.

Hypothesis 4:

SALSA provides appropriate abstractions that facilitate the implementation of agents' reactivity.

Hypothesis 5:

The execution model of SALSA agents (perception-reasoning-action) enables developers to easily conceive a ubicomp application.

6.2 Creating real ubicomp applications with SALSA

The first hypothesis emphasizes that if SALSA enables developers to create real ubicomp applications, then it is evidence that SALSA is useful. To envision some of the scenarios presented in Chapter 4 that could be implemented with SALSA, the activities of users in their real work settings were observed. To determine whether the scenarios identified from the workplace studies were considered realistic by the hospital's staff, evaluation sessions were conducted with some of them. The following sections describe the results obtained from evaluating the scenarios.

6.2.1 Scenarios depicted the context-aware medical practices of mobile users

In the evaluation of scenarios 3.3.1 and 3.3.2, 28 hospital staff members of the internal medicine area of the IMSS General Hospital —13 physicians, 8 nurses, and 7 support staff participated, including the chemist responsible for laboratory analyses. We presented staff members with animations of the scenarios and then gave them a questionnaire to measure how real they thought the scenarios were, if they would change anything, and if they could imagine new scenarios for context-aware mobile technology. More information regarding the evaluation methodology can be found in [Muñoz *et al.*, 2003a; Muñoz *et al.*, 2003b].

The results of the scenarios' evaluation showed that the interactions of the first and second scenario were considered realistic by 89.28% and 89.75% of the participants respectively. These percentages included all the physicians. On the other hand, one (1) of the nurses had an opposing opinion, and another nurse did not give any opinion at all. Similarly, one (1) member of the support staff (who were not represented in any of the scenarios) did not answer this issue for the first scenario, and two (2) of them did not provide an answer for the second scenario.

The questionnaire also included questions to evaluate the main characteristics of the scenarios. Table VIII shows the four questions and their responses. The results validate some of our findings from the workplace study: The staff found location information useful, that communication exchanges depend on context, and that handheld computers are an appropriate mechanism for accessing medical data. Additionally, most potential users were not concerned with the likely distractions from the messages received since they are already confronted with constant disruption in their daily work. The information obtained in this evaluation enabled us to measure the usefulness of the system in the context of the scenarios of use.

Table VIII. Results of a questionnaire measuring user acceptance of scenarios and the system's context-aware features

	Strongly Disagree	Disagree	Slightly Disagree	Neither	Slightly Agree	Agree	Strongly Agree
It is useful to know who is in the hospital and where they are in relation to me	1	0	0	0	0	7	20
It is useful to send messages that depend on context for their delivery	1	0	0	0	1	7	19
It is useful to have access to the patient's medical records through a handheld computer	1	0	0	0	0	8	19
Receiving messages can distract me from my daily work*	9	9	2	3	3	0	1

*One person did not respond to this question.

After the participants validated both scenarios, they provided us with additional insights by proposing the inclusion of interactions in the scenarios or modifying them so that they reflected real medical practices. Regarding the first scenario, some participants suggested that the system could enable users to visualize X-Rays images. The hospital staff considered it important for the system to inform the physician when the chemist received the lab analysis request. Similarly, they emphasized that for scenario 3.3.2, it was necessary to provide a mechanism to verify whether the message was received by the physician on the next shift. However, this problem exists in other communication systems in which there is no guarantee that a message has been seen by the addressee. In addition, in current work practices in the hospital there is no mechanism to notify whether the requests for lab analysis or medical notes have been delivered. Recommendations like this one were considered in the design of the system. Another relevant comment was that messages like the one sent by the physician in scenario 3.3.2 had to be integrated into the electronic patient record. However, our objective was not to emphasize the message content, but to illustrate how the system enables users to condition the delivery of a message. Other comments related to this scenario indicated that the system enables hospital personnel to transfer some medical information to personnel on the next shift. This indicated to us that the main idea of this scenario was understood and well accepted. The participants also suggested opportunities for applying the technology proposed by the scenarios—for example, scheduling a surgery by tracking the availability of the operating room and the specialists involved. This confirmed that the system illustrated in the scenario was considered useful. This evaluation helped us to see that it is only when people have a picture of their work that they can provide details beyond what we can observe without their input.

6.2.2 Public Displays to support coordination and communication

For this evaluation, a 3 minute animation presenting scenario 3.4.1 was created and showed to eleven physicians and two nurses. Afterwards, the subjects were asked to fill out a questionnaire and brief interviews were conducted with them in order to validate the system design depicted by the scenario. The questionnaire focused on specific aspects of the support provided by the technology. In the interviews, we asked them whether the scenarios were realistic and if the suggested use of the technology would enhance their professional practice. More information regarding the methodology followed in this evaluation is found in [Favela *et al.*, 2004].

The results of the questionnaires indicated that 70% of the subjects discuss clinical records or lab results with other doctors and nurses at least once a day, and all of them share this information at least once a week. When asked what media they considered to be the most appropriate for consulting a patient's medical record, 8 of them (61%) favored PDAs and 7 (53%) public displays. Only two respondents marked personal computers as their preferred option and none of them selected the current paper versions. Answers do not add up to 100% since subjects were allowed to mark more than one option. Responses changed slightly when asked about laboratory results, including X-rays. In this case public displays were the preferred media for 9 people (69%) and 7 of them mentioned PDAs (53%). Two people selected personal computers and two more paper versions. Clearly, physicians are attracted to the idea of having computerized access to patient records and favor the mobility offered by PDAs for clinical records, and large displays for lab results because of the limited screen size of handhelds. Yet, it seems that the need to share information with other people favored large displays over traditional personal computers. One physician expressed concern for having only a few public displays in the hospital, he believed that they would be insufficient to accommodate all potential users, since physicians often start their rounds and consult clinical records at around the same time. As made clear by this comment, these devices should be adequately distributed throughout the hospital to facilitate access. However, while the scenarios emphasized the use of handhelds and large displays, it is expected that PCs would be part of the computing infrastructure.

As illustrated by scenario 3.4.4 support for physicians to consult medical guides through the public display was proposed. Thus the use of medical guides was also investigated by asking the subjects if they found it to be helpful (38% agreed) and if the technology described in the scenario would make these guides more useful and increase their use (92% agreed). The 54% increase is an indication of a potential benefit of our design.

Finally, when the subjects were asked to compare the use of public displays, as illustrated in the scenario, with that of the whiteboards they currently use, 10 out of the 13 respondents agreed that the functionality displayed in the scenario covered all the functions of the whiteboards, and the same number of people thought that there would be no need for whiteboards at the hospital if the public displays were introduced there.

In the interviews, all participants found the scenario to be realistic and we got a clear sense that immediate and easy access to the medical records was the most attractive feature of the technology, and that they saw the combination of handhelds and public displays as providing adequate support to achieve this. One person mentioned that a limitation of the display was the fact that the map shows a single floor at the time, while on the board they include beds from different floors that are assigned to the area. This comment was considered for changing the design of the interface of the PublicDisplay application. Furthermore, the interviewees suggested extending the system to support distributed collaboration while working on the public display. In particular, one physician suggested that it would be desirable to be able to locate colleagues in other hospitals with whom they had worked before and perhaps call to consult on a clinical case. We perceive that the support could include providing awareness of the presence of remote contacts, application sharing and audio-conferencing. The evaluation presented in this section determined not only that the selected scenarios of use of ubicomp systems were considered realistic by the hospital personnel, but that the systems represented in the scenarios were also perceived as being useful for supporting their activities. The results from this evaluation showed that SALSA enabled the building of real and useful pervasive systems and facilitated their implementation as presented in Chapter 5. Thus, SALSA can be considered useful for implementing this type of pervasive systems in which users, devices and services were represented by autonomous agents. The following sections evaluate SALSA’s flexibility and ease of use for developing autonomous agents for ubicomp systems.

6.3 Completeness of SALSA

As explained in Chapter 4, SALSA provides a set of mechanisms (an API and services) to enable developers to cope with some of the complexities (described in Section 2.2.1) for building ubicomp applications. Even though these development frameworks provide support for some of the same complexities, they address them in different ways than SALSA as summarized in Table IX. This shows that SALSA supports the creation of ubiquitous computing systems by giving support for dealing with context information, discovery of services/devices and interaction, heterogeneity, and scalability by means of the use of autonomous agents. But, the main difference between SALSA and the other middlewares is that it provides an appropriate communication language in order for the main system's entities of the ubicomp environment, represented by agents, to communicate asynchronously. Thus, services, devices, and users can seamlessly interact by using the same communication protocol.

Table IX. Architectures for creating context-aware systems

DEVELOPMENT PLATFORMS **COMPLEXITIES**	**SALSA**
Robustness and routine failures	N/A
Deal with context information	SALSA provides facilities to obtain context information that can be derived by using simple rules specified in an XSL filter. Through a set of conditions in which the variables were defined as primary context information, secondary context information is derived. For instance, when an agent detects that certain values are met for specific contextual variables, it infers another context situation. In Appendix A the classes that enable this functionality are explained.
Adaptation	N/A
Discovery of services/devices and interaction	An Agent Directory contains information about the environment configuration. An agent acting as proxy to the Agent Directory provides the information requested by other agents.
Heterogeneity	SALSA agents can be executed in mobile devices (such as PDA's and Tablet PC) and personal computers acting as servers.
Scalability	SALSA supports scalability by enabling the extension of the functionality of the environment by adding other agents that represent devices, services and users.
Implementation Techniques	Autonomous Agents

Figure 28 contrasts the type of applications that can be implemented with the development frameworks explained in Chapter 2, and SALSA. These development frameworks were classified as: architectures that provide support for enabling mobile devices to interact, (mobile computing), for dealing with context-information (context-aware computing), for creating distributed systems by means of agent technology (multi-agent systems) and development architectures that enable ubiquitous and heterogeneous devices to interact (pervasive computing). In this Figure, SALSA is presented as a middleware with which developers can develop ubicomp systems as multi-agent systems, and that provides some support for dealing with context information.

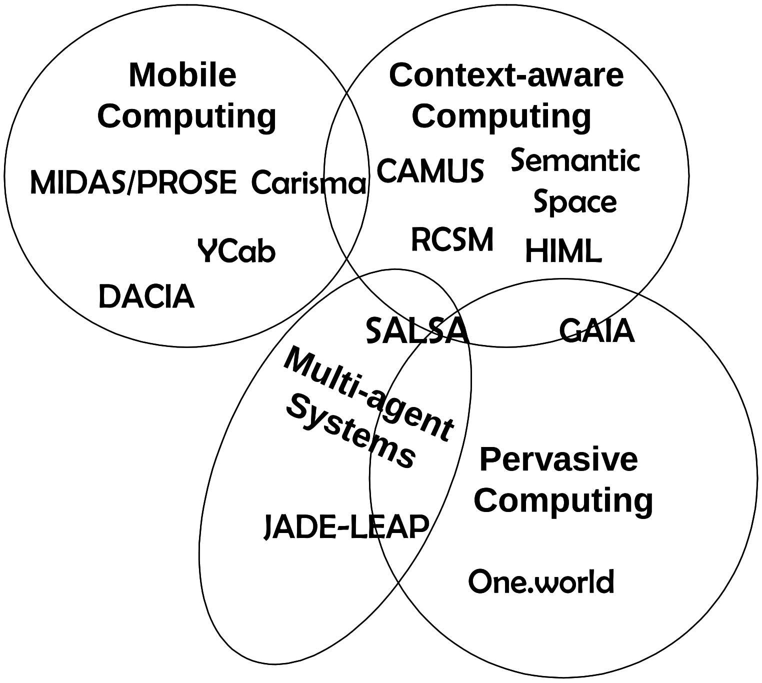


Figure 28. Scope of SALSA for implementing ubiquitous computing systems

6.3.1 Conclusions

The hypothesis for evaluating the completeness of SALSA states that it is flexible enough to facilitate the progressive development of ubicomp systems. As explained in Chapter 5, SALSA was used to implement the context-aware hospital information system

(CHIS) and the integration of new functionality wrapped in autonomous agents. In order to illustrate the process followed to make this possible, Section 5.1 provided a detailed description of the implementation of the location-estimation agent (LE-a) and the agents that enable the users interaction with public-displays and how they were integrated into the context-aware hospital information system. This provided evidence that the SALSA agent development framework facilitates the progressive development of a pervasive system.

6.4 Ease of use of the SALSA API

The first hypothesis for evaluating the ease of use of SALSA states that: "The SALSA framework is considered by developers to be easy to learn even if they have little or no previous familiarity with agent programming". In order to evaluate this, I conducted two experiments. The first experiment evaluated the API of SALSA in an in-lab experiment, and in the second one, I evaluated the use of agents as design abstractions for conceiving ubicomp systems. A group of the Fall 2004 offering of the Object Oriented Analysis and Design (OOAD) class at the CICESE Research Center participated in these experiments. Both experiments are described in the following sections.

6.4.1 Experiment 1: An in-lab evaluation of the SALSA API

The objective of this experiment was: "Evaluate the ease of use of the SALSA class framework to develop ubiquitous computing applications."

To achieve the above mentioned objective, an experiment conducted in three separate sessions was carried out. In the first session, we assessed the participants' abilities and experience in developing software systems. In the second session we explained to them the concepts of ubiquitous computing, software agents, SALSA, and the in-lab experiment in which they were going to participate. Finally, the evaluation session was conducted in a computing laboratory, in which the participants were asked to solve a programming problem using SALSA. These sessions and the results are described in the following sections.

6.3.1.1 Evaluating the participants' background

A group of nineteen (19) students was selected to participate in the experiment. Their programming and design background was evaluated in order to adapt the experiment to their level of expertise by providing information they would need to carry out the experiment using SALSA. For instance, as most of the students have little knowledge about autonomous agents, the students received information regarding the design of the system they had to implement which identified the autonomous agents and how they had to interact for providing the requested functionality.

Sixteen graduate students (of the 19) completed a questionnaire to assess their abilities and knowledge in software development and provided some personal information such as their age. Of the 16 students 5 are in the first year doctoral program and the other 11 are first-year masters students. The questionnaire consisted of four sections. The first one asked about their experience in programming software systems, programming paradigms and languages they have used. The second section questioned them about their knowledge of the JAVA programming language. Section three asked about their expertise in modeling systems using UML (Unified Modeling Language). And finally, the last section inquired about their knowledge of Software Agents.

The results showed that 93% of the participants had more than 4 years experience in developing software applications. Although 100% knew about the object oriented paradigm, 50% expressed having less experience in using the OO approach than other approaches such as imperative programming. The programming language they were more familiar with was C/C++ (87% of the participants) followed by JAVA (75 %); the rest of the participants mentioned other visual object oriented languages, such as Visual Basic and Delphi. 81 % were familiar with HTML, and half of them had used XML. Only 18% had recently learned UML during the current object oriented programming course they were taking, while the rest of the participants said they had used or learned UML before this course. With respect to their experience with software agents, 25% stated that they had no idea or only a vague idea of what an agent is. 56% were familiar with the concept of agents. Finally, 18% (3 of the 16 students) had implemented applications based on software agents.

The results obtained from this survey indicated that the participants did not have the same level of knowledge in some issues relevant for the experiment, such as the use of autonomous agents, which were then introduced during a class of the OOAD course. The results were also used to determine the additional information that they might need to consult during the in-lab experiment. For instance, since many of them were not experienced JAVA and/or XML programmers, online help was available during the experiment which provided a reference to the Java and SALSA library, and examples of how to parse XML messages with SALSA.

6.3.1.2 Introduction to SALSA for the development of ubicomp applications

A two-hour class was given to all participants to introduce them to concepts that were required in order to participate in the study. The agenda of the course is presented in Table X.

Table X. Course agenda for introducing SALSA to the participants

November 12, 2004	
Time	Activity
1.- Evaluation of participants' background	
10-15 min.	Apply questionnaire (See Appendix B)
November 17, 2004	
2.- Introduction	
5 min.	Explain the process of the experiment
15 min.	Introduce ubiquitous computing, context-aware computing and autonomous agents.
20 min.	Introduce SALSA (agent's components, facilities offered by SALSA, agents of the context-aware hospital system, and the design of SALSA)
10 min.	Questions and comments
3.- SALSA API	
30 min.	Explain the SALSA API and illustrate its use.
10 min.	Respond students questions about SALSA

We explained the concepts of ubiquitous and context-aware computing. Through a use scenario, we illustrated how ubicomp technology may enhance users' activities. Then, we explained the concept of autonomous agent, its attributes, and illustrated with the same use scenario, how ubicomp systems can be designed using autonomous agents. We explained SALSA's architecture and the execution model of SALSA agents, which specifies that when an agent perceives information, it reasons, based on the information perceived, to decide what action to execute. Then, we introduced the SALSA API by explaining the abstractions used for implementing the agent's components for perceiving, reasoning and acting, and for deriving secondary context by means of an XSL filter. Finally, we illustrated how to implement the Hello World Agent with the SALSA API.

At the end of the presentation, the participants were allowed to make questions and comments. Their main concerns were relating to the events created by the SALSA agents to perceive information; and about the use of XSL to derive contextual information. In this regard, they asked questions such as: "What happens if any of the conditions in the XSL are not satisfied?", "What is the format of the XSL?" All questions were answered. Two students made comments in the sense of the SALSA API being appropriate for implementing scenarios as the one presented".

6.4.2 In-lab Evaluation

We conducted an informal, controlled evaluation of SALSA to learn about its ease of use. During this evaluation all (19) nineteen students from the OOAD course participated. Through three programming tasks, participants were asked to implement and extend the functionality of one of the autonomous agents of a ubicomp system that monitors the state of a patient with diabetes, and based on this information, decide on a course of action. During the experiment, we recorded the participants' comments and questions, and saved their code for further review. When the participants finished their programming exercises, they were asked to answer a questionnaire to evaluate the ease of use of the SALSA API. The description of the experiment and the analysis of the results of this in-lab evaluation are explained in the following section.

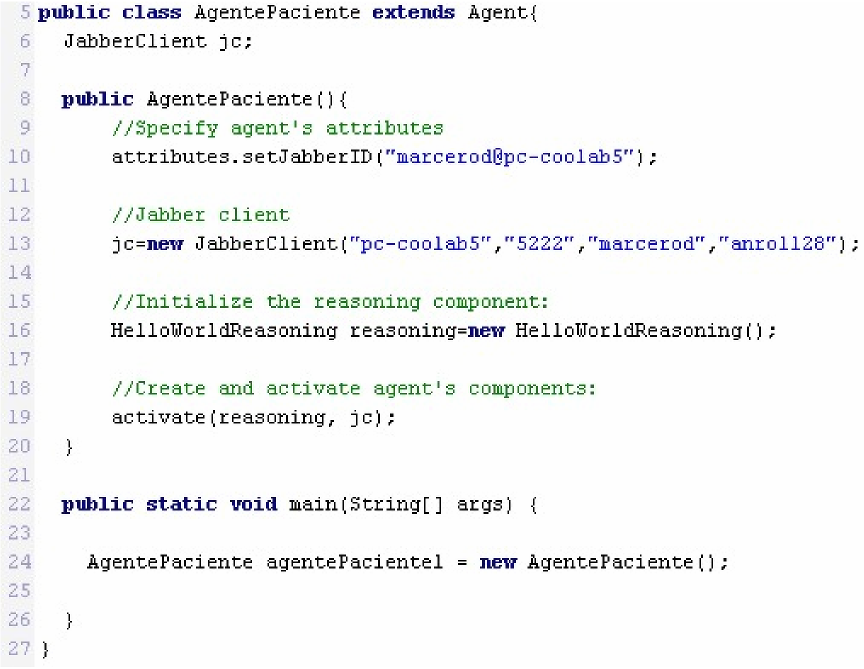
6.3.1.1 Description of the experiment tasks

We began the evaluation session by explaining the experiment. Then we handed out a description of the programming tasks as shown in Appendix B. To illustrate the desired functionality of the system, the following scenario was provided to the participants.

"George, a patient with diabetes, is alone at home watching a soccer game. Suddenly, he feels sick and the ubicomp system detects that he has hypoglycemia. That is, his glucose levels are low (76mg/dl). Thus, the system recommends him to take 2 tablespoons of sugar. A little later, the system measures the glucose level to be lower (70 mg/dl) and starts monitoring the patient's pulse and his level of perspiration, which at that moment are normal. At the same time, the system notifies George's daughter of his condition. The system continues perceiving George's vital signs. As he still presents hypoglycemia, and his pulse and level of perspiration increase, the system decides to send a warning message to George's doctor.

The main components of the system depicted in the scenario are modeled with three autonomous agents acting as proxies to the sensors, which monitor the levels of glucose, pulse, and perspiration, respectively. These agents send the information detected to a fourth agent, the patient's agent, which determines the patient's health-state and decides what action to execute. Participants were asked to develop the patient's agent. For the other agents the participants were given executables that simulated that they were actually perceiving information from the sensors. Thus, the conditions of the patient, such as hypoglycemia (low glucose) or hyperglycemia (high glucose), were simulated, which enabled the participants to verify if their agent acted as expected.

We gave the participants the code of the general structure of the patient's agent as presented in Figure 29. They had to implement the agent's reasoning and action components as requested in each task according to the execution model of SALSA.

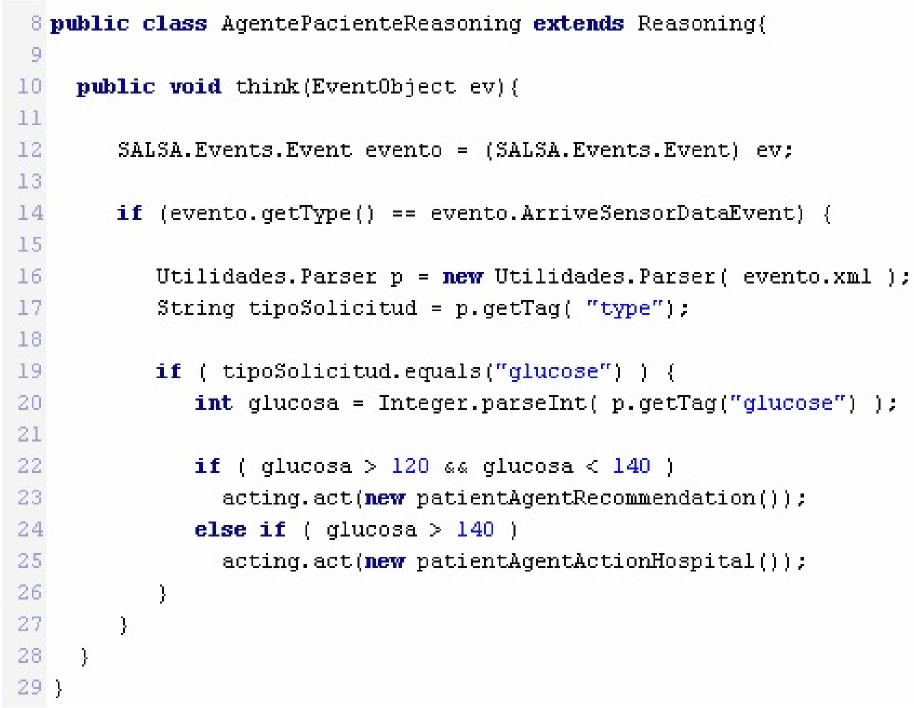


```java
public class AgentePaciente extends Agent{
  JabberClient jc;

  public AgentePaciente(){
      //Specify agent's attributes
      attributes.setJabberID("marcerod@pc-coolab5");

      //Jabber client
      jc=new JabberClient("pc-coolab5","5222","marcerod","anroll28");

      //Initialize the reasoning component:
      HelloWorldReasoning reasoning=new HelloWorldReasoning();

      //Create and activate agent's components:
      activate(reasoning, jc);
  }

  public static void main(String[] args) {

    AgentePaciente agentePaciente1 = new AgentePaciente();

  }
}
```

Figure 29. Code of the patient's agent

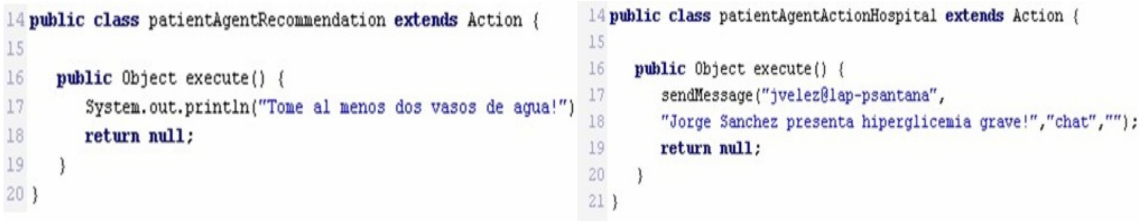
Task 1 asked the students to code the conditions for the patient’s agent to diagnose hyperglycemia. Thus, when the agent perceived that the glucose was greater than 120, it had to recommend that the patient drink 2 glasses of water. If the agent again detected hyperglycemia 15 minutes latter, it had to notify the patient’s doctor by sending a warning through instant messaging. We expected the participants to implement the reasoning component of the agents (see Figure 30) with the appropriate conditions to determine that the patient had hyperglycemia, and the actions to be execute as illustrated in Figure 31.

In task 2, participants were asked to modify the agent to detect hypoglycemia by using the facilities provided by SALSA to derive context. The agent had to act as indicated in the scenario presented before. We provided the participants with the code of the XSL filter (see Figure 32) with the conditions to detect hypoglycemia. The XSL filter read the primary context, which in this case was the information of the levels of glucose, perspiration and pulse, and then deduced a secondary context, it indicated that the patient was having hypoglycemia or hyperglycemia.



```java
public class AgentePacienteReasoning extends Reasoning{

  public void think(EventObject ev){

     SALSA.Events.Event evento = (SALSA.Events.Event) ev;

     if (evento.getType() == evento.ArriveSensorDataEvent) {

        Utilidades.Parser p = new Utilidades.Parser( evento.xml );
        String tipoSolicitud = p.getTag( "type");

        if ( tipoSolicitud.equals("glucose") ) {
           int glucosa = Integer.parseInt( p.getTag("glucose") );

           if ( glucosa > 120 && glucosa < 140 )
             acting.act(new patientAgentRecommendation());
           else if ( glucosa > 140 )
             acting.act(new patientAgentActionHospital());
        }
     }
  }
}
```

Figure 30. Agent's reasoning diagnosing hyperglycemia

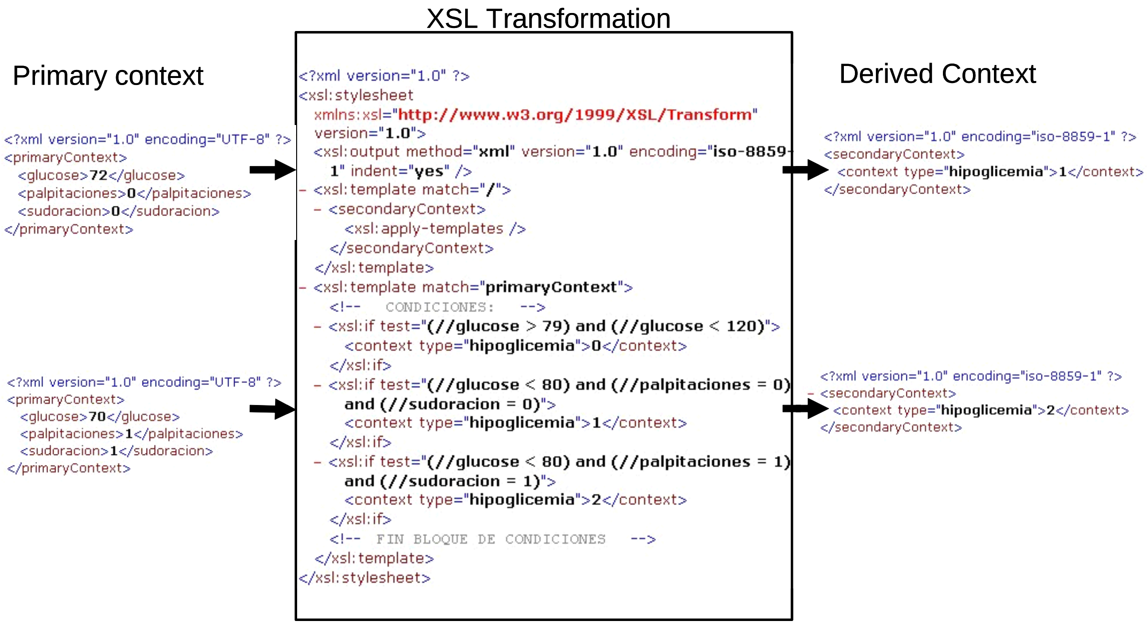


```java
public class patientAgentRecommendation extends Action {

   public Object execute() {
      System.out.println("Tome al menos dos vasos de agua!")
      return null;
   }
}
```

```java
public class patientAgentActionHospital extends Action {

   public Object execute() {
      sendMessage("jvelez@lap-psantana",
      "Jorge Sanchez presenta hiperglicemia grave!","chat","");
      return null;
   }
}
```

a) **b)**

Figure 31. Agent's actions when hyperglycemia is detected. a) The agent recommends drinking water. b) The agent notifies the doctor about the patient's condition



```
<?xml version="1.0" encoding="UTF-8" ?>
<primaryContext>
  <glucose>72</glucose>
  <palpitaciones>0</palpitaciones>
  <sudoracion>0</sudoracion>
</primaryContext>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<primaryContext>
  <glucose>70</glucose>
  <palpitaciones>1</palpitaciones>
  <sudoracion>1</sudoracion>
</primaryContext>
```

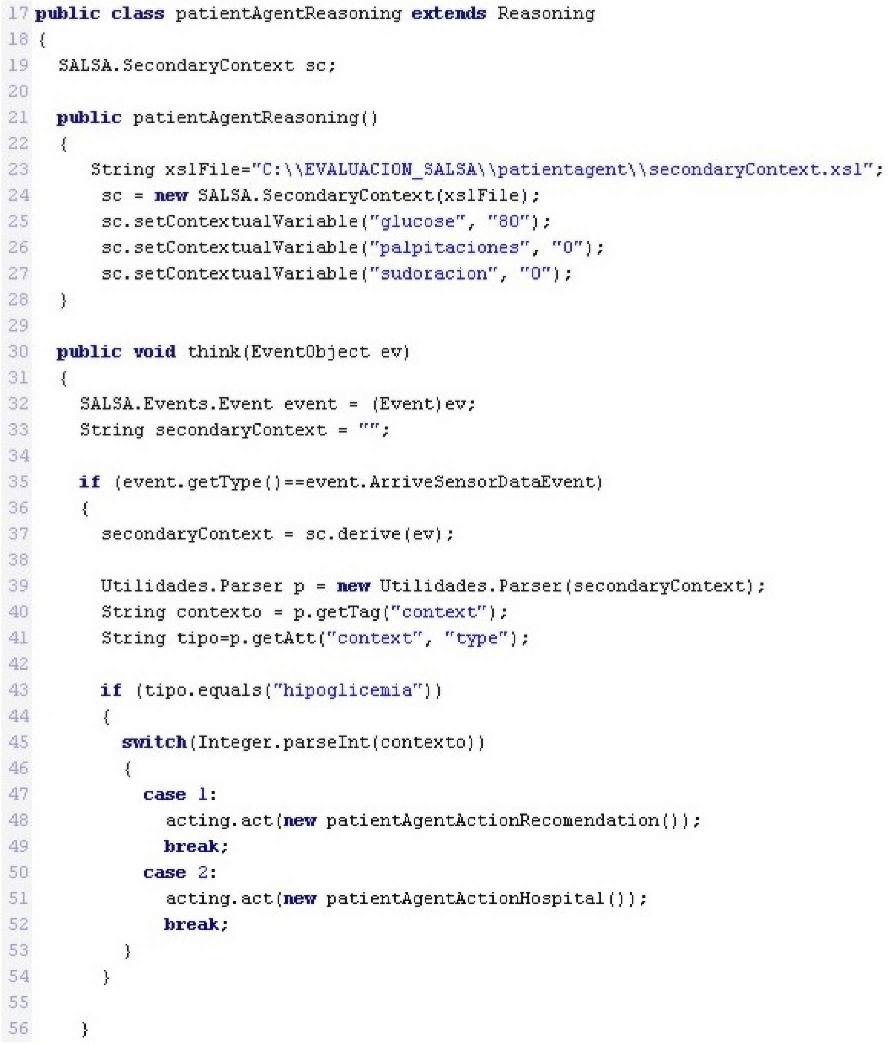
```
<?xml version="1.0" ?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="xml" version="1.0" encoding="iso-8859-
    1" indent="yes" />
- <xsl:template match="/">
  - <secondaryContext>
      <xsl:apply-templates />
    </secondaryContext>
  </xsl:template>
- <xsl:template match="primaryContext">
    <!--   CONDICIONES:   -->
  - <xsl:if test="(//glucose > 79) and (//glucose < 120)">
      <context type="hipoglicemia">0</context>
    </xsl:if>
  - <xsl:if test="(//glucose < 80) and (//palpitaciones = 0)
      and (//sudoracion = 0)">
      <context type="hipoglicemia">1</context>
    </xsl:if>
  - <xsl:if test="(//glucose < 80) and (//palpitaciones = 1)
      and (//sudoracion = 1)">
      <context type="hipoglicemia">2</context>
    </xsl:if>
    <!--  FIN BLOQUE DE CONDICIONES   -->
  </xsl:template>
</xsl:stylesheet>
```

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<secondaryContext>
  <context type="hipoglicemia">1</context>
</secondaryContext>
```

```
<?xml version="1.0" encoding="iso-8859-1" ?>
- <secondaryContext>
  <context type="hipoglicemia">2</context>
</secondaryContext>
```

Figure 32. XSL filter to detect hypoglycemia

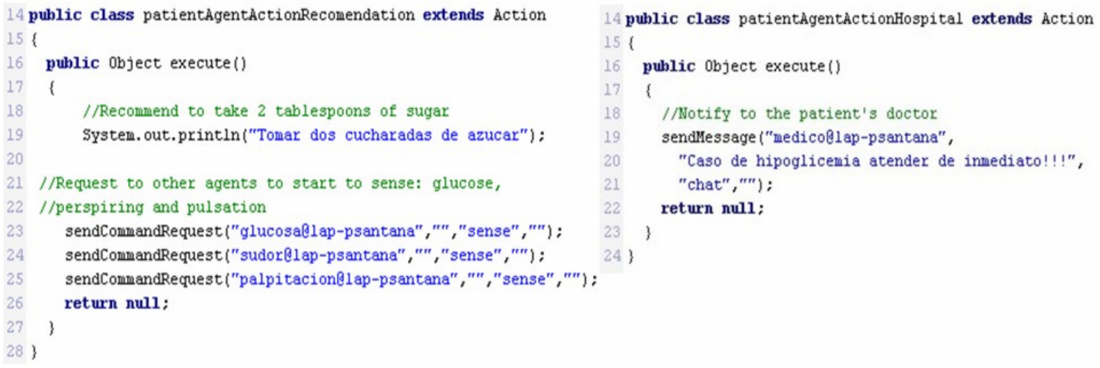
Figure 33 and Figure 34 present how we expected the participants to code the reasoning and action components. The constructor of the class that implements the reasoning component (see Figure 33) specifies that the variables used to derive contextual information are: the level of glucose, perspiration and pulsations. Thus, when an event of type ArriveSensorDataEvent is passed to the method `think(),` the context information is abstracted from the event by the `derive(ev)` method to return the context derived as an XML message in the secondaryContext variable. This message is then parsed to check the level of hypoglycemia of the patient. Thus, if the patient presents hypoglycemia for the first time, the agent acts by recommending a dose of sugar, and requests the other sensing agents to start monitoring the patient's pulse and perspiration as illustrated in Figure 34 a). If the level of glucose does not stabilize and the patient continues presenting hypoglycemia, the agent notifies this to the patient's doctor, as illustrated in the code in Figure 34 b).



```
public class patientAgentReasoning extends Reasoning
{
  SALSA.SecondaryContext sc;

  public patientAgentReasoning()
  {
     String xslFile="C:\\EVALUACION_SALSA\\patientagent\\secondaryContext.xsl";
      sc = new SALSA.SecondaryContext(xslFile);
      sc.setContextualVariable("glucose", "80");
      sc.setContextualVariable("palpitaciones", "0");
      sc.setContextualVariable("sudoracion", "0");
  }

  public void think(EventObject ev)
  {
    SALSA.Events.Event event = (Event)ev;
    String secondaryContext = "";

    if (event.getType()==event.ArriveSensorDataEvent)
    {
      secondaryContext = sc.derive(ev);

      Utilidades.Parser p = new Utilidades.Parser(secondaryContext);
      String contexto = p.getTag("context");
      String tipo=p.getAtt("context", "type");

      if (tipo.equals("hipoglicemia"))
      {
        switch(Integer.parseInt(contexto))
        {
          case 1:
            acting.act(new patientAgentActionRecomendation());
            break;
          case 2:
            acting.act(new patientAgentActionHospital());
            break;
        }
      }

    }
```

Figure 33. Reasoning component detecting hypoglycemia

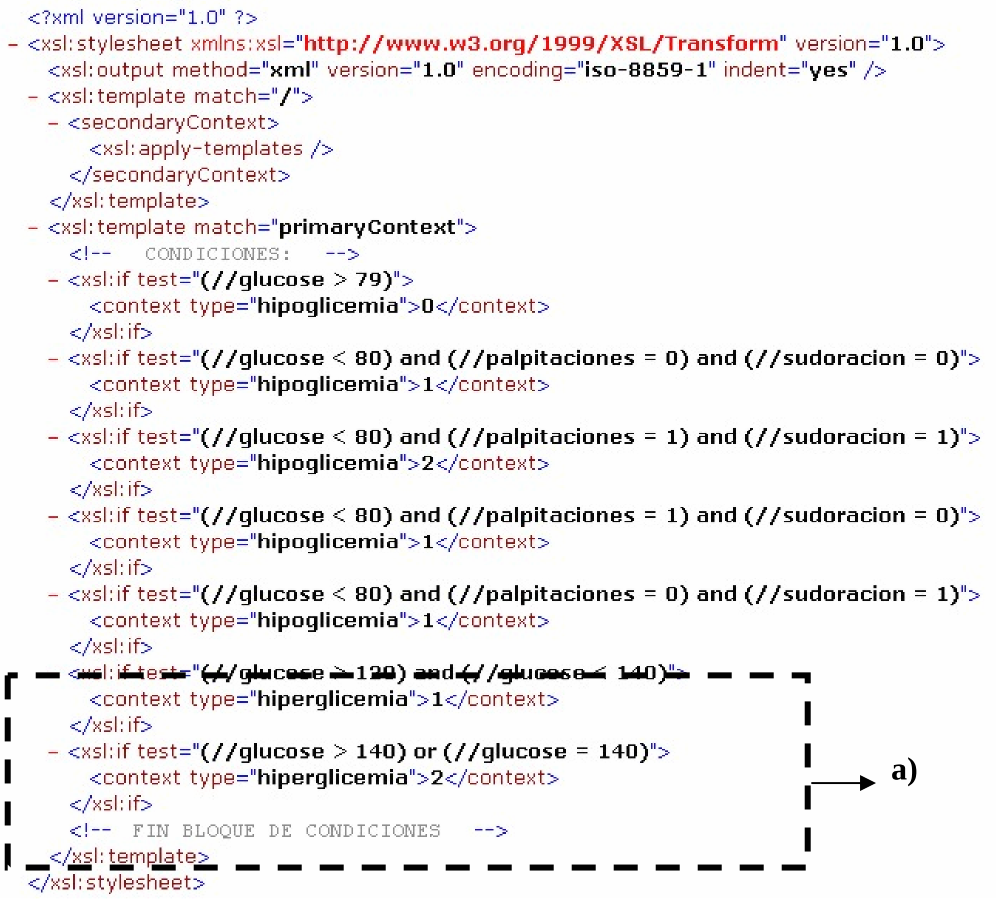
Finally, in task 3, we required the participants to modify the XSL filter to include the conditions to detect hyperglycemia as was indicated in task 1. The XSL filter had to be modified as depicted in Figure 35 a) in which the added code lines are enclosed in a dashed box.



```
public class patientAgentActionRecomendation extends Action
{
  public Object execute()
  {
      //Recommend to take 2 tablespoons of sugar
      System.out.println("Tomar dos cucharadas de azucar");

  //Request to other agents to start to sense: glucose,
  //perspiring and pulsation
     sendCommandRequest("glucosa@lap-psantana","","sense","");
     sendCommandRequest("sudor@lap-psantana","","sense","");
     sendCommandRequest("palpitacion@lap-psantana","","sense","");
     return null;
  }
}
```

```
public class patientAgentActionHospital extends Action
{
  public Object execute()
  {
    //Notify to the patient's doctor
    sendMessage("medico@lap-psantana",
      "Caso de hipoglicemia atender de inmediato!!!",
      "chat","");
    return null;
  }
}
```

Figure 34. Agent's actions when hypoglycemia is detected. a) The agent recommends the patient eat a sugar dose. b) The agent notifies the physician the patient's health-condition

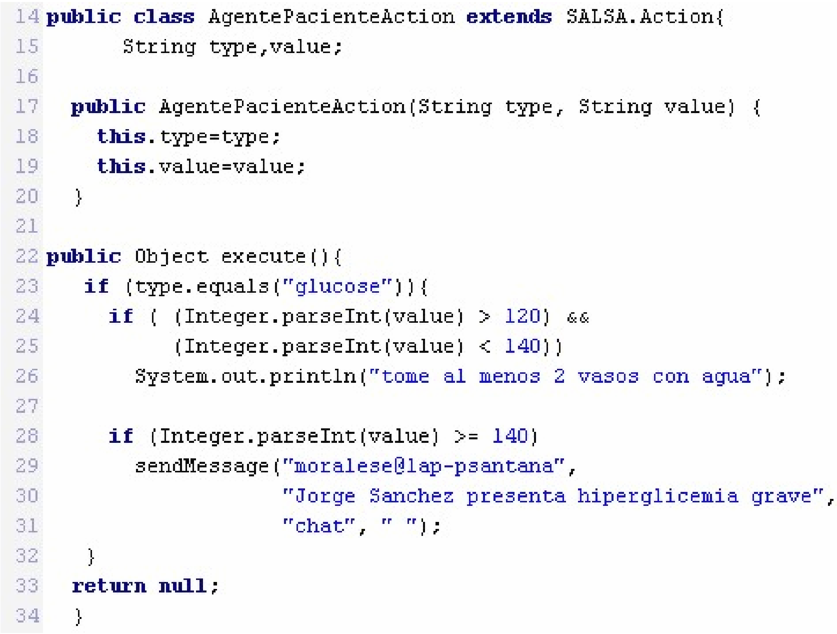


```
  <?xml version="1.0" ?>
- <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="xml" version="1.0" encoding="iso-8859-1" indent="yes" />
  - <xsl:template match="/">
    - <secondaryContext>
        <xsl:apply-templates />
      </secondaryContext>
    </xsl:template>
  - <xsl:template match="primaryContext">
      <!--   CONDICIONES:   -->
    - <xsl:if test="(//glucose > 79)">
        <context type="hipoglicemia">0</context>
      </xsl:if>
    - <xsl:if test="(//glucose < 80) and (//palpitaciones = 0) and (//sudoracion = 0)">
        <context type="hipoglicemia">1</context>
      </xsl:if>
    - <xsl:if test="(//glucose < 80) and (//palpitaciones = 1) and (//sudoracion = 1)">
        <context type="hipoglicemia">2</context>
      </xsl:if>
    - <xsl:if test="(//glucose < 80) and (//palpitaciones = 1) and (//sudoracion = 0)">
        <context type="hipoglicemia">1</context>
      </xsl:if>
    - <xsl:if test="(//glucose < 80) and (//palpitaciones = 0) and (//sudoracion = 1)">
        <context type="hipoglicemia">1</context>
      </xsl:if>
      <xsl:if test="(//glucose > 120) and (//glucose < 140)">
        <context type="hiperglicemia">1</context>
      </xsl:if>
    - <xsl:if test="(//glucose > 140) or (//glucose = 140)">
        <context type="hiperglicemia">2</context>
      </xsl:if>
      <!--  FIN BLOQUE DE CONDICIONES   -->
    </xsl:template>
  </xsl:stylesheet>
```

Figure 35. XSL filter to detect an hypoglycemia and hyperglycemia

6.3.1.2 Results from the inspection of the code

The source code of the patient's agent produced by the students was analyzed to evaluate whether they comprehended the execution model and the communication protocol of SALSA agents. Only five (5) participants did not distribute the functionality of the agent as we expected. These participants implemented part or the whole reasoning logic (a set of rules) in the action component as illustrated in Figure 36. For this, the information perceived in the reasoning component was passed to the action component to detect the patient's health condition and decide how to act, rather than having the reasoning component do this. Three (3) of the participants failed at implementing the perception of information. They did not verify whether the event received in the reasoning component was of type ArriveSensorDataEvent, which means that any perceived message, such as a presence message, is analyzed by the reasoning component to find out the patient's condition.



```
public class AgentePacienteAction extends SALSA.Action{
      String type,value;

  public AgentePacienteAction(String type, String value) {
    this.type=type;
    this.value=value;
  }

public Object execute(){
   if (type.equals("glucose")){
     if ( (Integer.parseInt(value) > 120) &&
          (Integer.parseInt(value) < 140))
       System.out.println("tome al menos 2 vasos con agua");

     if (Integer.parseInt(value) >= 140)
       sendMessage("moralese@lap-psantana",
                   "Jorge Sanchez presenta hiperglicemia grave",
                   "chat", " ");
   }
  return null;
  }
```

Figure 36. Action component implemented by one of the participants

Most of the participants were able to successfully implement each of the agent's components, but not without some difficulties. Participants were allowed to check SALSA's user manual during the study or they could ask questions to any of the two graduate students running the experiment, either relating to SALSA or any other programming issue that came up.

During the first activity participants became familiar with the API of SALSA. Even though some of the participants understood the execution model of SALSA, they did not remember the name of the abstractions and methods of the SALSA API explained during the 2-hours course. Among the main questions they made were: "What are the methods for communicating information to other agents?"; and "What are the events received in the reasoning object?" But, the main concerns of the participants during the first activity were related to XML. Their major questions were about how to abstract the information from the message by using the facilities of SALSA since the majority of the participants were familiar with HTML, but not with XML. As we expected, during the second and third activity the participants' main questions were about XSL and the facilities offered by SALSA to derive secondary context using an XSL filter, which was new for most of them.

6.3.1.3 Perception of ease of use

When the participants finished their programming tasks, they completed a survey presented in section B.1, which included topics such as their perception on the use of SALSA, how difficult it was for them to learn the API of SALSA and finally, what functionalities they considered easier to implement and which ones were the most difficult ones.

The section of the questionnaire that evaluates the perception of ease of use of the SALSA API included six assertions, each one with 7 Likert-scale answers. These questions are important predictors of Information Technology adoption according to TAM [Davies *et al.*, 1991]. In this case, SALSA is the information technology that may be adopted by users who in this case are software developers. As illustrated in Table XI, the participants perceived the API to be easy to use since most of them "slightly agree" (5) or "agree" (6) with all the questionnaire assertions. Even though some of the participants perceived that learning (Q1) and interacting (Q3) with SALSA was not so easy (5.27) and understandable (5.267), most of them "agree" that it would be easy for them to become skillful at using the SALSA API (Q5).

Two final questions were included in the questionnaire to find out the agent's functionalities that were perceived as easier to implement, and which ones were considered the most difficult ones. As illustrated in Figure 37, the Action component was considered as the agent's functionality that is easiest to implement. A participant wrote that this component was the easiest to implement *"because it does not need so many lines of code for implementing the communication of the agent"*; another participant stated: *"you just need to select the appropriate method of communication according to the type of information to convey"*. Users considered that the Action component was the simplest to implement, and as they stated, some of the difficulties they faced during the agent's implementation was that they were not familiar with the communication methods of SALSA. Implementing the Reasoning component was easy to implement for eight (8) of the participants. One subject stated that *"it was easy, because you just have to overwrite the reasoning logic"*, which means that once they were familiar with the implementation of the first programming activity, they considered that it was easy to modify the agent's behavior for the second and third activity.

The agent's functionality that was considered the most difficult to implement, by six (6) of the participants, was to derive context information by using the XSL filter. This is comprehensible, since these persons had no previous experience in XML and XSL, as they stated: *"I have never used XSL files"*, "*programming in XML is new to me*". For the same reason, some of the subjects stated that the reasoning component was also difficult to implement. One of them commented: "*It was difficult to parse the XML message*".

Table XI. Results of TAM questionnaire

Questions	Average	Stand. Dev.	Mode
Q1: Learning the SALSA API would be easy for me.	5.27	1.033	6
Q2: I would find it easy to implement intelligent systems with SALSA	5.667	0.617	6
Q3: My interaction with SALSA would be clear and understandable	5.267	1.033	6
Q4: I would find SALSA to be flexible to interact with	5.4	1.639	6
Q5: It would be easy for me to become skillful at using the SALSA API	5.8	0.676	6
Q6: I find SALSA easy to use	5.467	0.834	5

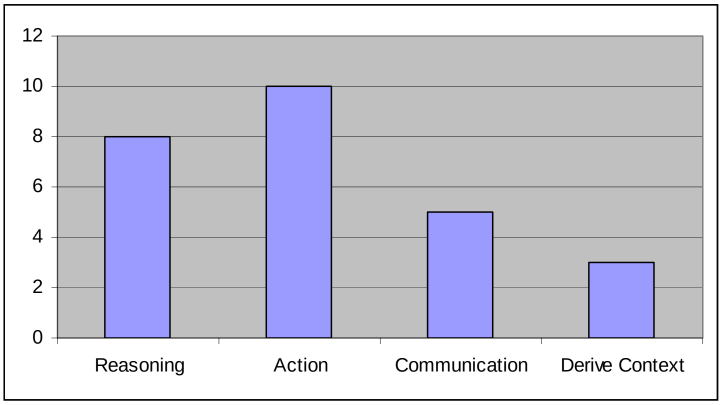


Figure 37. Answers to the question: What agent functionality you consider to be the easiest to implement?

6.4.3 Conclusions

Even though participants had concerns about how to use the SALSA API to implement some of the functionalities of the agent, most of them could successfully create the agent as we expected. It may be evidence that the execution model of SALSA and the facilities to implement it are comprehensible. For some of the participants the use of autonomous agents as an abstraction for the deployment of an ubicomp system was not innate. The problems faced by some of the participants for distributing the agent's functionality among the action and the reasoning components were due to their lack of experience in implementing SALSA agents and that they were not familiarized with the execution model of SALSA (defined in the SALSA agent's life cycle). However, the participants faced minimum problems for implementing and understanding the functionality of the perception component. This was due to the fact that most of the agent's perception is left to the SALSA infrastructure which automatically creates and activates this component when the agent is instantiated, and the programmer only has to extract the received information from the launched event. Thus, some of the participants did not remember how to capture the event generated when the information is passed to the reasoning component, which was easily solved by consulting the documentation of SALSA.

Participants had to understand that agents communicate in a different way than other type of components, such as objects or processes. The subjects perceived that the communication protocol was easy to understand. They expressed that for implementing the agent's communication they just had to select the appropriate methods according to the information that want to be conveyed. The difficulties experienced by the participants for implementing the agents' communication was due to agents using a communication language that is semantically richer than that commonly used by object technology. The SALSA communication language is based on XML, which is too verbose and adds additional complexity to the API of SALSA, a fact that was also noticed by the participants which have no experience parsing XML messages. Even though SALSA provides some facilities for extracting the content of a SALSA message, the lack of knowledge in XML by the subjects did not enable them to easily comprehend the syntaxes of the SALSA messages. In the same way, participants also experienced difficulties using XSL since they had no previous experience with it. This did not enable participants to perceive that the mechanisms in SALSA to derive secondary context were easy to use. Thus, we conclude that the SALSA API needs to include mechanisms at a higher level of abstraction to facilitate the management of XML messages and the creation of the XSL filter.

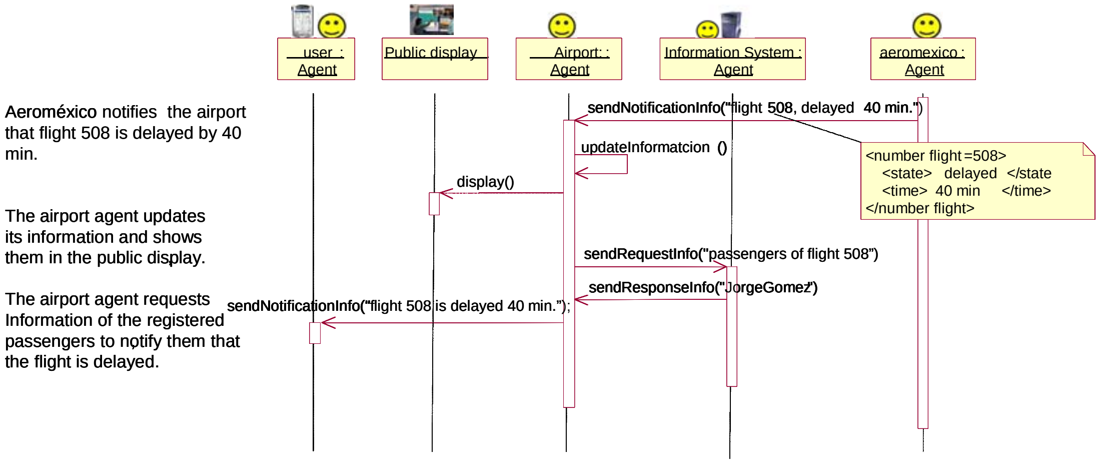
6.5 Autonomous agents as an abstraction tool for designing ubicomp systems

The objective of this stage was: “To evaluate the ease of use of SALSA agents in designing a ubicomp system”.

6.5.1 Description of the Design Experiment

During the experiment eighteen (18) graduate students (of the nineteen that participated in the Lab evaluation of SALSA) could participate in a design problem of a ubicomp system by using autonomous agents. For this, the design problem was included in the evaluation exam of the Object Oriented Analysis and Design course taken by these students. The complete exam is in section B.2. The exam lasted two hours, and the design problem, which was part of the exam, was planned to be solved in one hour. As presented in Figure 38, the exercise consisted of a description of a use scenario of a ubicomp system for an Airport setting, a sequence diagram to express the interactions among the system’s agents, and instructions requesting participants to design the proposed system and then modify it to extend its functionality. Participants had to first analyze the description of the ubicomp system and its sequence diagram to identify the main components of the system. Since the aim of SALSA is to build ubicomp systems by means of autonomous agents that represent users, services or wrap a complex system’s functionality, it was expected that the participants would identify the main components of the ubicomp system as autonomous agents. This activity enabled the participants to familiarize themselves with the design of the current ubicomp system. In the next activity, they were asked to extend the application’s functionality according to the scenario of use provided. For this, they had to identify new agents and their interactions with the other system’s agents by elaborating a sequence diagram and a component diagram. Participants were not penalized for not using the communication methods of SALSA. Finally, they had to provide a detailed description of each of the agents’ components, such as perceiving, reasoning, and acting.

The Mexico City airport has an ubiquitous computing system that allows users to consult flight information in a public display or the user's handheld computer (PDA). The following sequence diagram illustrates the interaction of the components of the system, which have been implemented as SALSA agents.


user : Agent
Public display
Airport: : Agent
Information System : Agent
aeromexico : Agent
Aeroméxico notifies the airport that flight 508 is delayed by 40 min.
The airport agent updates its information and shows them in the public display.
The airport agent requests Information of the registered passengers to notify them that the flight is delayed.
sendNotificationInfo("flight 508, delayed 40 min.")
updateInformatcion ()
display()
<number flight=508>
<state> delayed </state
<time> 40 min </time>
</number flight>
sendRequestInfo("passengers of flight 508")
sendResponseInfo("JorgeGomez")
sendNotificationInfo("flight 508 is delayed 40 min.");

You have been hired to extend the functionality of the system with SALSA agents, so that the system can also recommend services available within the airport that could be useful to the passengers while they wait for their flight. The system should have the following functionality: The user's PDA when notified that his flight has been delayed, request information of the available services and their location to the airport's Service Directory. The Service Directory provides this information and a map of their location. The system in the PDA personalizes the map according to the user's preferences and available time, highlighting services such as restaurants, book stores, internet café. A use scenario of the extended system will be as follows:

"When Mr. Jorge Gomez enters the airports waiting room, his PDA notices him that he has received a message indicating that his flight has been delayed by 40min. The system in the PDA requests to the airport's Service Directory those services that are available to determine those that could be useful to Mr. Gomez and that he could take advantage of while he waits for his flight. The Service Directory provides this information and a map with the location of the services. The PDA, knowing the time he has and his preferences, elects to highlight the location of the Vips restaurant and the 'Mexico" bookstore, since Mr. Gomez needs to buy a book for her daughter. Mr. Gomez goes to the restaurant and while he waits for his food, accesses information of the bookstore, selecting in the map a link to the bookstore's web page to consult the availability of the book he wants."

You are asked to:

a) Create a diagram showing the components of the original system. (**10 pts.**)

b) Modify the previous sequence diagram, incorporating the components that implement the new functionality required by the system. (**20 pts.**)

c) Modify the components' diagram elaborated in question *a.* to incorporate the new components of the extended system. (**10 pts.**)

d) Describe the sequence diagram you have extended, explaining the behavior of the agents. That is, what functionality is implemented by each of the components of each agent (perception, reasoning, action) (**10 pts.**)

Figure 38. Design exercise of a ubicomp system

6.5.2 Criterion to evaluate the design exercise

To evaluate the programming exercise, the following criterion was established. The total score of the activities of the design exercise was 50 points. Each exercise was reviewed to verify if it fulfilled a set of conditions. In case a condition was not achieved, the value of the exercise was decremented by a penalization value as explained in Table XII.

Table XII. Criterion applied to check the exercises of the experiment.

Exercise	Value	Criterion (Penalization)
a)	10 pts	An agent was not included as a component in the diagram (-4) A relationship between two components was not specified (-1) Conceptual problem: use of the wrong relationship to associate components (-2) Conceptual problem: a component was identified as interface or node (-3)
b)	20 pts	The system's functionality is not complete or clear due to: an interaction was not specified (-3) an agent was not specified (-2)
c).	10 pts	A component that extended the system's functionality was not included in the diagram (-4) A relationship between two components was not specified (-1) Conceptual problem: use of an incorrect relationship to associate components (-2) Conceptual problem: a component was identified as interface or node (-3) *Note:* The mistakes made in exercise a) were not penalized again in exercise c).
d)	10 pts	Functionality not correctly distributed to each of the agents' components: perception-reasoning-action (-4) One of the agent's components not explained (-1) The explanation of an agent's component is not clear or correct (-1)

6.5.3 Results

The diagrams presented in Figure 39 and Figure 40 illustrates how the system can be designed by using autonomous agents. Figure 39 presents a version of the Sequence Diagram illustrating the interaction of the Agents. It shows that the functionality of the system was extended by adding two agents that enable the user to access the available services: the Agent Directory which acts as a proxy to the Airport Directory by providing information of all the available services and the map of the airport; and the Library Agent which enable the user to access information of the Library. In this design the communication protocol of SALSA was used to enable the interactions among agents. Figure 40 presents the diagram of components illustrating the system's agents that were identified.

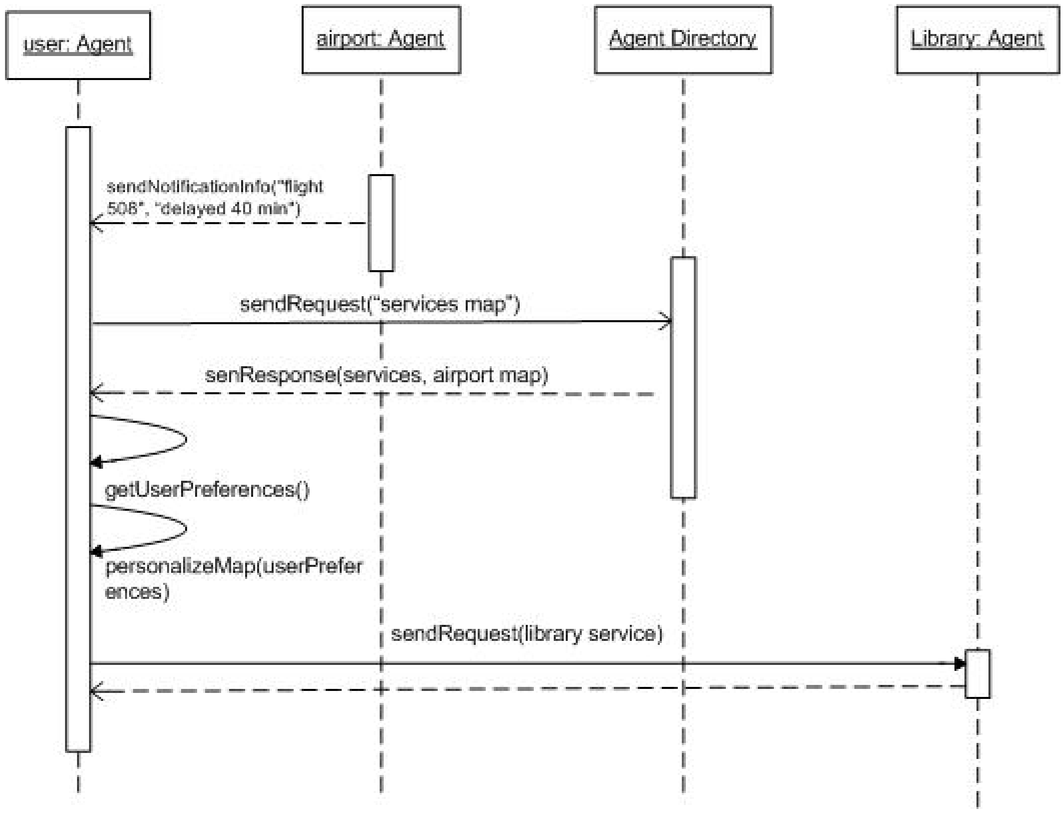

user: Agent
airport: Agent
Agent Directory
Library: Agent
sendNotificationInfo("flight 508", "delayed 40 min")
sendRequest("services map")
senResponse(services, airport map)
getUserPreferences()
personalizeMap(userPreferences)
sendRequest(library service)

Figure 39. Solution of exercise b) illustrating the agent's interactions for the requested functionality

The design exercise of the participants was evaluated based on the above diagrams and the criteria of evaluation established.

Table XIII shows the points obtained by each participant in the exercise, and the average grade obtained in each task by the group. The following sections discuss the experiment's results.

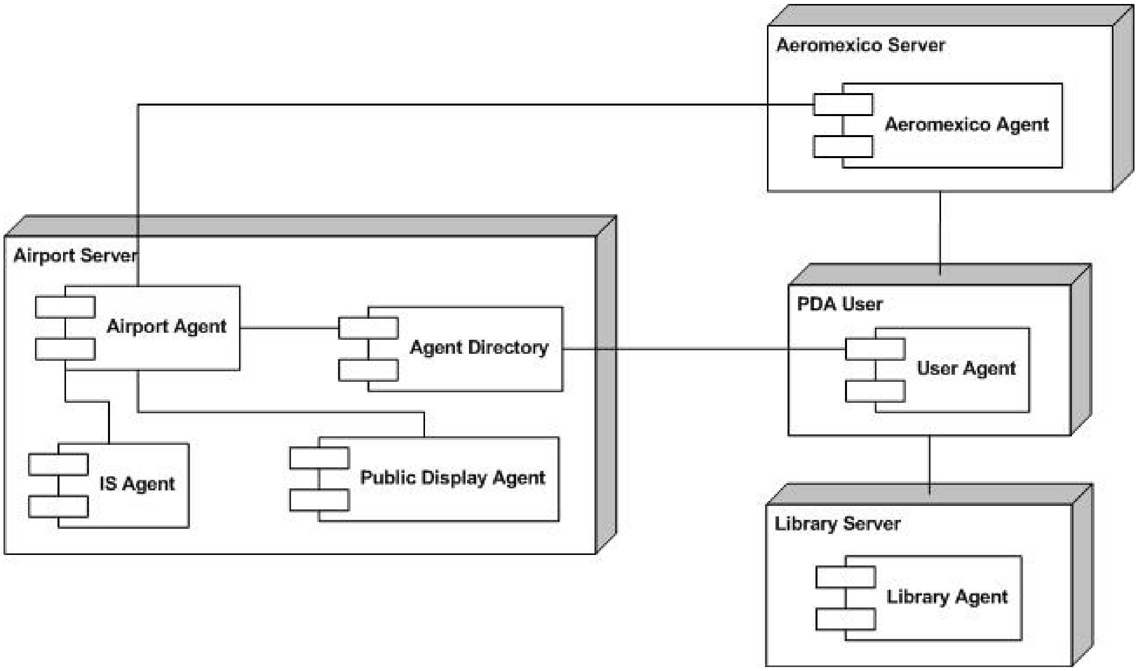


Figure 40. Diagram showing as components the system's agents

Exercise *a*: Create a diagram showing the components of the original system

The average grade of all participants was 7.67 out of 10 points. Of the eighteen (18) participants, four (4) obtained the maximum score, 10 points; three (3) persons got 9 points; and six (6) got 8 points. The points of the other five (5) participants ranged from 3 to 6. The most common mistake was related to conceptual problems of UML for modeling Component Diagrams. For instance, one of the participants used a notation for Deployment Diagrams, and the majority of the participants (13) made mistakes in establishing the relationships among the components.

Six (6) persons did not name the components as agents, and two (2) of them also described the system's functionality in terms of components or subsystems instead of agents in exercise *d*. One of these two (2) participants declared to "have a vague idea of what an agent is" in the questionnaire applied in phase one. Although most participants said they had a vague idea of what an agent is and of its application, the majority of them (16 persons) had no difficulty identifying the agents of the proposed system as components which wrap the main functionality of the ubicomp system.

Table XIII. Participants score in the design exercise

PARTICIPANT	**a)**	**b)**	**c)**	**d)**	**TOTAL**
G1/1	5	13	3	6	27
G2/7	8	13	5	7	33
G1/7	8	15	6	5	34
G1/4	8	10	6	10	34
G2/9	3	18	5	10	36
G1/3	5	15	6	10	36
G2/5	8	13	6	10	37
G1/6	4	18	9	6	37
G1/5	8	13	9	7	37
G1/2	10	18	6	5	39
G2/3	6	20	6	8	40
G1/8	9	15	6	10	40
G2/6	10	20	10	0	40
G2/4	9	17	9	6	41
G2/2	8	18	6	10	42
G2/1	10	18	6	9	43
G2/8	10	18	9	9	46
G1/9	9	20	10	9	48
AVERAGE	7.67	16.22	6.83	7.61	**38.33**

Exercise *b*: Modify the previous sequence diagram, incorporating the components that implement the new functionality required by the system

The value of exercise b) was 20 points. The group averaged 16.22 points. Three (3) participants got 20 points, and six (6) participants got 18 points. The minimum score was 10 obtained by one (1) person; and the rest of the participants obtained from 13 to 17 points. In this exercise, there were no errors due to conceptual problems with the notation of UML for Sequence Diagrams. The most frequent mistake was that the scenario was incomplete: Seven (7) participants did not include the interaction with the library service which had to be represented by a proxy-agent; two (2) participants did not specify the interaction to personalize the map for the user's preferences; and one (1) participant did not include the Service Directory, but the User Agent requested the available services to the Information System Agent. In spite of the fact that two weeks before the participants made a laboratory practice using SALSA, not all of them comprehended the scope of SALSA to extend the functionality of a system. This, may be because they did not design the system's agents that they were asked to implement during the laboratory practice, but they used some of the core features of SALSA to get an idea of how to create a single agent that interacts with other predefined agents. The second major error in which five (5) participants incurred was that the Airport Agent was used as a communication intermediary between the User Agent and the Service Directory. In spite of this, the protocol of interaction of SALSA agents was well expressed in the diagram sequence for the majority of the participants. They correctly and clearly specified the agents' interactions to send or request information to other agents. Even though they were not requested to use the real names of SALSA's methods for communication, several of them (8 persons) remembered and used them.

Exercise *c*: Modify the components' diagram created in question *a*. to incorporate the new components of the extended system

The group average for this exercise was 6.83 out of 10 points. To check this exercise, we did not take into account the conceptual problems in which participants incurred in solving exercise *a*. For instance, the participants again made mistakes establishing the relationships between components. Just two (2) participants got 10 points. Four (4) of the participants, who missed specifying a relationship among two of the components, got 9 points, even though in the sequence diagram they clearly specified it. Nine (9) persons got 6 points; two (2) obtained 5 points; and finally, one (1) person obtained 3 points. They were penalized with -4 points since they did not include a component. Nine (9) persons did not identify the Library Service as an agent, even though two (2) of them included this agent in the sequence diagram. Two (2) of these nine (9) persons did not include the Directory Service in the component diagram. Some of the mistakes made in this exercise were due to incorrect design decisions made in the sequence diagram. For instance, if the participants did not indicate the interactions with a proxy-agent to the Library in the sequence diagram, they also were not included in the component diagram. However, the other nine (9) participants created agents to add the new functionality to the system following the approach suggested by SALSA to develop ubicomp systems.

Exercise *d*: Describe the sequence diagram you have extended, explaining the behavior of the agents. That is, what functionality is implemented by each of the components of the agent (perception, reasoning, action)

The average of the group for this exercise was 7.61 of 10 points. Six (6) persons got 10 points; three (3) persons got 9 points; and the rest of the participants got from 5 to 8 points. Five (5) of the participants who obtained 6 points or fewer points failed to explain each of the components of the system’s agents (perception, reasoning and action), but they gave a general description of each of the system’s agent.

Most of the participants (14) described the agents’ components clearly. They stated how and when the agents perceive; reason based on the perceived information; and specified what events trigger an agent’s action. One of them envisioned that the user’s agent may perceive from the to-do list of his PDA that he had to buy a book for his daughter.

6.5.4 Conclusions

The results of exercise a) shows that the participants could abstract the agents as the main system's building blocks from the ubicomp application scenario. However some of them (8 persons) did not model correctly the extended system as requested in exercise c) since they did not identify all the agents' interactions in exercise b). The results of this evaluation show that using autonomous agents for designing a ubicomp system requires that developers understand the agent-oriented programming paradigm well. Mainly they need to comprehend that in this approach the idea of interaction is emphasized, as well as the idea of choice and options at the time of action rather than at the time of programming [Makoto, 2005]. This evaluation also shows that even though the students had learned about the use of SALSA agents approximately 2 weeks before, they were able to identify the agent's components and some of them used the SALSA methods for communication.

Thus, autonomous agents were used by the participants as the main building blocks or components which wrap the main functionality of the ubicomp system. Autonomous agents were used to represent the system's components that need to proactively act to enable the seamlessly user's interaction with the computing devices and services of the environment.

This section evaluated the ease of use of SALSA agents for designing ubicomp systems, but it is also necessary to evaluate SALSA by comparing it with other middlewares that enable the implementation of multi-agent systems. For this, SALSA was contrasted with the JADE-LEAP framework as explained in the following section.

6.6 Comparing SALSA with JADE-LEAP

There are several agent development frameworks that facilitate the building of multi-agent systems. Among these, the JADE-LEAP project which is a FIPA complaint platform, has been used to implement agents for ubiquitous computing environments since JADE-LEAP allows the deployment of agents on any Java-enabled device with sufficient resources and with wired or wireless connection [Bergenti and Poggi, 2001]. This section compares JADE-LEAP with SALSA in terms of their execution model and communication protocol for creating ubicomp systems with the aim of finding out which are more appropriate for handling the new interaction paradigms that emerged from the vision of ubiquitous computing (i.e. spontaneous interactions among the ubicomp components, context-aware interaction).

6.6.1 Execution model of agents

The aim of SALSA is to facilitate the building and iterative evolution of a system. In this sense, SALSA provides a suitable agent execution model and communication protocol to implement ubicomp systems. The creation of a ubicomp system involves the identification of software entities that may provide opportunistic services and information to users or hide a complex functionality from users with which they do not need to explicitly interact. To do this, the entities have to monitor the ubicomp environment, process the perceived information (i.e to infer the context of users) and decide how to act (i.e. adapting information to user's context) which might require communication with other entities. These entities are software agents that perceive context information, and reason to autonomously decide how to act. The agent execution model of SALSA enables programmers to naturally conceive the entities of a ubiquitous computing system as agents that, when alive, perceive context information, reason and then act. The agent execution model offered by JADE requires that developers analyze how to distribute the agent's tasks among one or more types of behaviors, which may be an annoying design task. For instance, developers have to decide if the logic of an agent will be implemented as a single behavior or distributed among several behaviors.

6.6.2 Communication protocol

The SALSA communication protocol provides a small set of specialized messages that indicate what the agent intends to achieve in the ubicomp environment when sending a message. An agent representing a user, service or device may require communication with other agents in order to:

- send context information which may be perceived through ubiquitous sensors;
- notify when the state of a device or the location of a mobile user has changed;
- request the mobility of an agent, which enables an agent to update its reasoning

component when some conditions of the environment have dynamically changed. Thus, an agent adapts to these changes in order to continue providing its services;

- request a service or action from another agent;

- and finally, request information that can only be retrieved by another agent acting as a proxy to an information system or repository.

Thus, with SALSA's communicative messages, agents can get context information from the environment, adapt their actions to new contexts, and collaborate with other agents by requesting information or services from them. The JADE communication protocol follows the FIPA specifications for requesting actions, information and notifications, and for negotiating services in the same way that negotiations are carried out at an auction (i.e. seller agents wait for a proposal from the highest bidder). Therefore, JADE may be more suitable for implementing applications that involve this kind of transaction, such as e-commerce applications, than for ubicomp applications. In order to illustrate the support offered by SALSA and JADE to create a pervasive system for health-care, Figure 41 presents how JADE agents interact to implement support for the scenario of Section 3.3.1, and we compare it with the SALSA implementation depicted in Figure 27 of Chapter 5. It can be seen that JADE does not provide facilities in the communication protocol and infrastructure in order for the context-aware client to notify the presence of the mobile user and his change of state and location, as SALSA does. In addition, the messages do not provide enough information about the intention of the communicative act as SALSA does. For instance, the INFORM message may be used to send the instant messages from the user's PDA to the map agent, notify of the user's presence, and communicate some type of context information. This leads the implementers to have to extend the content of the message. Thus, the transactions between agents of a ubicomp environment are better supported by SALSA than by JADE. Some of the disadvantages of using JADE are that its communication protocols rely on technology based on RPC, such as RMI. It has been considered that RPC (or its newer implementations) is not a suitable paradigm for ubiquitous computing systems [Saif and Greaves, 2001]. RPC may be inflexible and inefficient for the communication requirements of ubiquitous computing systems in several aspects, such as [Saif and Greaves, 2001]: Most of the interactions among services or devices are one way in order to indicate “do this” type commands or notifications. However, RPC enforces a tightly coupled bidirectional interaction. Thus, the calling thread is suspended even for interactions that do not require a return value to continue with its operation.

6.6.3 Autonomy of agents

An important requirement of agents is that they must be able to act autonomously when needed. FIPA establishes that an agent, representing (the AMS agent) the managing authority on an agent platform (AP), is responsible for managing the operation of an AP, such as the creation of agents, the deletion of agents, deciding whether an agent can dynamically register with the AP and overseeing the migration of agents to and from the AP (if agent mobility is supported by the AP). The life cycle of all the agents in the AP is maintained and controlled by the AMS [Bergenti and Poggi, 2001]. Thus, agent autonomy is not supported by FIPA platforms since an autonomous agent can not be controlled by any external agent or object [Bergenti and Poggi, 2001].

6.6.4 Conclusions

SALSA enables developers to cope with the requirements of ubiquitous computing scenarios such as the ones presented in Chapter 3. SALSA provides an appropriate and easy to understand agent execution model. We have argued that the SALSA communication language is more appropriate then the one provided by JADE-LEAP, to convey information among the software entities of a ubicomp system, which are conceived as autonomous agents.

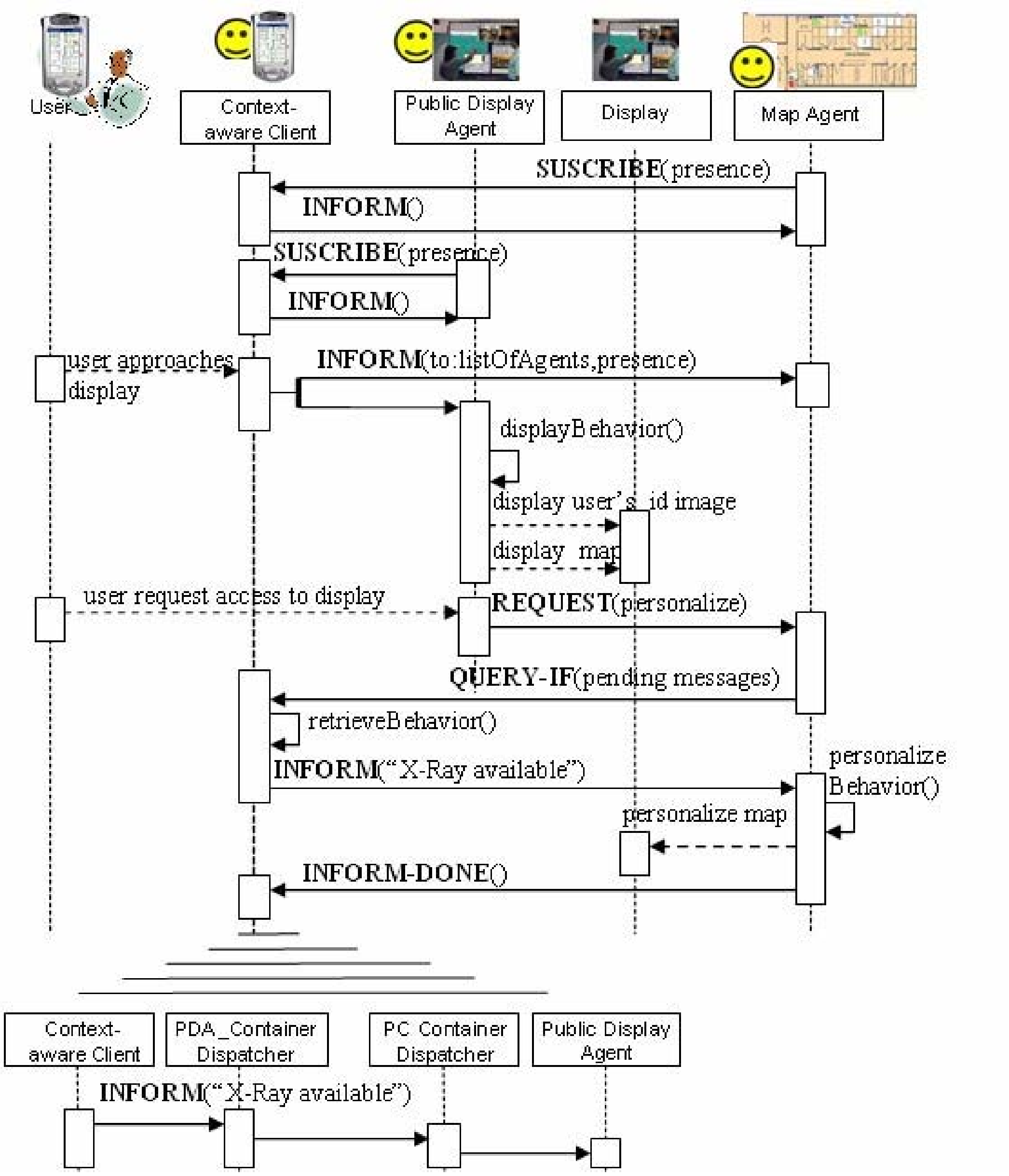


Figure 41. Design of scenario 3.3.1 by using JADE

6.7 Discussion

This Chapter has presented the results of evaluating the utility, completeness and ease of use of SALSA through different evaluation methods, that included evaluating scenarios of use with real users, in-lab programming experiments, and design exercises for evaluating the facilities provided by SALSA agents.

The criterion for determining the utility of the middleware stated that if others can build real pervasive applications atop the infrastructure, then the architecture is useful. The first experiment determined that the selected scenarios of use of ubicomp systems, which were implemented with SALSA, were perceived as being useful by real users (such as physicians and nurses) therefore SALSA can be considered useful for creating ubicomp systems in which autonomous agents enable users to seamlessly interact with other users, services and devices.

To evaluate the completeness of SALSA, an evaluation experiment was conducted to determine if the architecture was sufficiently powerful and extensible to support interesting user-space programs. For this a hypothesis arose stating that SALSA was flexible enough to enable the evolution and iterative implementation of ubicomp applications. For demonstrating this, Chapter 5 presented how SALSA facilitated the implementation and extension of the context-aware hospital information system which was proposed for supporting medical activities carried out in a hospital.

The results of the programming exercise provided evidence that the execution model of SALSA and the facilities to implement it are comprehensible (H3). For some of the participants the use of autonomous agents as an abstraction for the deployment of an ubicomp system (H4) was not innate since participants had to understand various concepts related with agents and the facilities provided by SALSA, such as the agents' communication protocol, and the use of XSL for deriving context information.

The design exercise, in which the students participated in, approximately two weeks after the programming exercise, reinforced the veracity of hypothesis 3 and 5 since the participants used autonomous agents as the main system's components. Most of them were able to identify the agents' components as defined by the execution model of SALSA, and some of them remember how to use the SALSA communication protocol.

To complement the evaluation of the SALSA facilities (H5), SALSA was compared with the JADE-LEAP middleware, a middleware used for creating agents that may also be executed in mobile devices. This was done through the design of a system by using JADE-LEAP, and designing the same system by using SALSA. As a result of this evaluation it was concluded that the SALSA communication language is more appropriate then the one provided by JADE-LEAP, to convey information among the software entities of a ubicomp system, which were conceived as autonomous agents.

Chapter 7

Conclusions

This thesis has presented how a ubiquitous computing system can be implemented by using autonomous agents as its main components. Autonomous agents have characteristics that can be used to implement some of the desirable features of ubicomp systems such as those described in Chapter 4. In those systems, autonomous agents were mainly used to enhance the user's interaction with the ubicomp environment in which autonomous agents were the software components that represented users, devices, and services. This was possible since autonomous agents are reactive to the environment (to the users' context) and proactive to decide how to act (providing services and information to users). This thesis takes advantage of these agents' characteristics in order to provide an agent middleware (SALSA) that facilitates the development of ubicomp systems.

7.1 Contributions and results

The contributions of this thesis was in providing evidence of how autonomous agents can be used as an abstraction tool for designing and implementing ubiquitous computing systems, and providing a middleware that facilitates their implementation. The results of the contributions of this thesis and how they were addressed in previous Chapters are summarized in the following sections.

7.1.1 Identifying the complexities associated with the development of ubicomp systems.

Before illustrating how autonomous agents can assist the development of ubicomp systems, this thesis focused on identifying the complexities associated to the design and implementation of pervasive computing applications. Chapter 2 explains the challenges faced when developers try to address these complexities of ubiquitous computing systems which were identified as: *robustness and routine failures, dealing with context information,* *adaptation, discovery of services/devices and interaction, heterogeneity, scalability and implementation techniques*.

In order to aid developers in dealing with these complexities, several development architectures have been proposed. Providing a development architecture that helps developers to address the ubicomp complexities represents another one of the challenges faced in ubicomp research. Chapter 2 also presents an analysis of the support provided by existing development architectures for dealing with some of these complexities.

Finally, as this thesis proposes that autonomous agents can be used as an alternative for dealing with ubicomp complexities, Chapter 3 introduces autonomous agents and their characteristics and how they can be an appropriate alternative for creating ubiquitous computing systems.

7.1.2 A set of realistic scenarios of ubiquitous computing systems

A set of scenarios of ubiquitous computing systems enabled me to analyze how autonomous agents can be used as the main system’s components. These scenarios were selected from observing users while performing their activities in their working setting. The chosen setting for studying and proposing how ubiquitous computing technology could enhance the users’ activities was a local public hospital. These scenarios are described in Chapter 4. Chapter 7 presents evidence that the hospital staff felt that the scenarios represented real medical activities, and that the proposed systems were useful for enhancing their activities.

7.1.3 Design issues regarding autonomous agents for developing ubiquitous computing systems

From the proposed systems described in the scenarios it was possible to identify the desirable system features and the design issues of the functionality of autonomous agents for creating these ubicomp systems. The following are the design issues described in Chapter 4:

- *Autonomous agents are decisions makers.* Autonomous agents review the users' context and make decisions about what activities to do, when to do them, and what type of information to communicate, and to whom.
- *Autonomous agents are reactive to the contextual elements of the environment.* Autonomous agents may need to monitor context information, such as the location of users and of services as well as the state of artifacts and users for opportunistically providing information and services to users. For this, agents need mechanisms to perceive, recognize and disseminate different types of context information.
- *Autonomous agents can represent users, act as proxies to information resources of the environment or wrap a complex system's functionality.* In the scenarios, users require access to information resources, services or devices which were identified as agents acting as proxies to them. These agents can be aware of the presence of other agents and users available in the environment. A user can interact with her personal agent that acts on her behalf for accessing information or services. Finally, agents can be wrappers of complex system's functionality that should be transparent to users.
- *Autonomous agents should be able to communicate with other agents, or directly with users and services.* Agents need a platform and communication protocol that enables them to convey information to other agents, users, information resources, devices and services. This platform and communication protocol should enable agents to seamlessly interact with users in order to enhance their interaction with the ubicomp environment.
- *Autonomous agents need mechanisms for authentication.* Agents should enable allowed users and other agents to interact with the devices and services of the ubicomp environment. Thus, autonomous agents need mechanisms for authenticating users and agents that want to access them.
- *Autonomous agents need to communicate different types of messages.* Agents need a communication language to convey messages for requesting information from devices or services and responding to such requests, notifying information to users and devices, and requesting from another agent the execution of an action.

- *Autonomous agents may have a reasoning algorithm as complex as the logic of its functionality.* An autonomous agent needs to be aware of information regarding the environment in order to decide how to act. For this, agents need a reasoning algorithm which may need a simple set of condition-action rules or exhibit a more complex behavior.

7.1.4 An agent middleware, named SALSA, for developing ubiquitous computing systems.

The above design issues were the foundation for creating an agent middleware that provides the mechanisms to facilitate the development of ubicomp systems as explained in Chapter 5.

The communication channel among agents and users is an Agent Broker (its implementation is an Instant Messaging Server) which is responsible for coordinating the communication among agents. To implement autonomous agents as decision makers, the SALSA middleware provides a library of classes for implementing and handling the execution model of agents, which consists of the components for perceiving information, reasoning, and acting as described next:

Due to the fact that the ubicomp environment is highly dynamic, an agent can perceive context information at unpredictable times from other agents, from the devices or services, or from the users. Agents can perceive information through the Agent Broker or directly from devices or sensors. The perceived information generates events which are captured by the reasoning component which governs the agent's actions. The programmer, based on the logic of the agent, implements the reasoning component by using any reasoning algorithm, such as a simple condition-action rule, a neural network or case based reasoning. SALSA provides abstractions to enable developers to easily modify or update the agent's reasoning requiring little or no modifications to the other agent's components. The action component implements the action plan to follow based on the agent's reasoning. It also includes sub-components that allow the agent's communication and mobility in order to update its reasoning component, and to derive context information based on information perceived by the agent. The actions of autonomous agents may require that they communicate with other agents and users. The communication protocol of SALSA allows agents to negotiate services with other agents, request them to execute an action and communicate with users in order to notify or present information to them. The communication platform, which is based on the instant messaging paradigm, enables users to be aware of the presence of other users and agents that offer relevant services for the users' activities.

Finally, SALSA enables the naming and registration of agents in an Agent Directory (AD). This is a service which is accessible through an agent acting as a proxy to the AD. The SALSA API provides a set of classes that allow programmers to register the agent's attributes and to look for agents available in the ubicomp environment.

Use of SALSA as a research test-bed computing systems in which autonomous agents were used as the main systems' components. These systems made it possible to identify how SALSA facilitates the development of ubicomp systems. Some of these systems were documented in Chapter 6 to provide evidence of how SALSA facilitates the implementation of autonomous agents for ubicomp systems and how these agents can be easily integrated in a ubicomp environment in order to extend its functionality. Thus, the implementation of these systems and the results of evaluating SALSA, described in Chapter 7, not only enabled the identification of the strengths of SALSA agents, but also their weaknesses which open opportunities for further research in this area as explained next.

7.2 Future research work

During the different stages of this thesis several research issues were identified. Some of them were addressed while others were not in order to define the orientation of this research and delimit its scope. This section explains the research issues that can be the foundation for further investigation.

7.2.1 Supporting mobile users' disconnections.

Mobile computing devices such as Personal Digital Assistants (PDAs) and smart cell phones are becoming major players in ubiquitous collaborative environments. As illustrated in our scenarios, these devices can act as mediators of the user's personal space and the public infrastructure, and allow the ubicomp environment to know the presence and location of users.

A natural event of mobile computing devices is that they can be disconnected for several reasons: users decide to turn off them; the device is out of the range of the network coverage; or the device's battery power is consumed. Disconnections can be a problem from different points of view:

- From the HCI (Human Computer Interaction) perspective: disconnections restrict the users' opportunity for interacting with the ubicomp environment in order to opportunistically access information and services. Thus, disconnections can limit the opportunity for reaching the "calm computing" vision in which computing devices should be naturally integrated into the users' activities.

- From the CSCW (Computer Supported Collaborative Work) perspective: the collaboration of users may be interrupted by the disconnections of their mobile devices. Some solutions propose keeping a copy of the data and collaborative application in the mobile device in order for users to continue using the application when they are next able to [Buzko *et al.*, 2001; Litiu and Parkash, 2000]. However, users lose the opportunity for continuing collaborating with their peers and participating in making important decisions.

- From the Distributed Systems approach: mobile devices can maintain data replicated in the mobile device in order to continue accessing them, but due to memory limitations it may not always be appropriate.

For dealing with disconnections, autonomous agent can be used for representing users while they are disconnected. In this case, an autonomous agent might be able to maintain a limited user presence and execute actions on his behalf, such as making decisions, accessing services on his behalf, or executing some actions. Studying the current collaborative activities of mobile users could help in identifying how autonomous agents can act on behalf of users. And then, incorporate to the SALSA middleware the support necessary for dealing with disconnections. For instance, autonomous agent may need mechanisms for knowing the users' preferences in order to know how to act, or learn from the users' behavior and activities to identify their preferences.

7.2.2 Authentication of agents

The authentication of agents that may represent users or services, was a design issue raised during the analysis stage of the ubiquitous computing scenarios. Existing research work in this issue seeks to provide a mechanism for supporting the authentication of users by using dedicated hardware, i.e. in the Gaia middleware, users have to wear active badges [Al-Muhtadi *et al.*, 2002] which requires that users accessing a service or device be physically present in the physical environment. Similarly in [Bardram *et al.*, 2003] the term proximity based-login is introduced which allows users to be authenticated by means of a Smart Card when they approach a device.

For integrating authentication mechanisms to SALSA, it is necessary to first analyze what users' information has to be taken into account for authenticating them in a ubicomp environment. For instance, in a hospital, physicians can modify the patient records for providing instructions to the nurses, but nurses only have permission to read this information. Besides that, a patient in a hospital may be examined by different nurses and physicians during the three different working shifts. Thus, the users' id and a password may not be the only items required for controlling the users' access to the ubiquitous computing devices, services and information. The users' role, her location and current time of day may be other elements that should be considered to provide users access to the ubicomp environment. We realize that the authentication problem may involve other issues regarding privacy and security for accessing entities of the environment.

Different alternatives or a mixture of them should be analyzed for integrating the service for authenticating users to SALSA:

- An autonomous agent could be the component that checks the credentials of incoming users to the environment.
- An alternative for deploying the authentication service is that this agent accesses the Agent Directory of SALSA, which could be extended for storing the profile that users should have for allowing them to interact with the services or devices registered in the Agent Directory.

- An appropriate authentication protocol for ubicomp systems should be identified.
- Finally, analyze what changes the communication protocol of SALSA may require for enabling the secure transmission of the users' credentials among agents.

7.2.3 Providing alternative communication channels

By using SALSA for implementing autonomous agents for ubicomp environments, several issues were raised for improving its communication support.

Recently SALSA was used for implementing a component that included several agents to enable information transfer between different devices [Amaya *et al.*, 2005]. During the implementation of this component, we realized that using the SALSA XML messages to transmit files, such as documents and pictures, was not appropriate since the length of the messages may have a size limit when transmitted through the Agent Broker. Besides that, in some situations it is not appropriate to send the information through the Agent Broker: For instance, SALSA agents are also being used for enabling users to use their PDAs as an interaction interface with a public display. This application provides feedback of the results of the user's actions by showing the same information presented in the Public Display on his PDA. However, the presentation of this information was delayed for several seconds since this information was notified through the Agent Broker. This situation may disturb the flow of the user's interaction with the Public Display, thus this feedback may not be as useful as expected.

In both of the above cases it was useful to use the Agent Broker to notify the presence and state of the agents, and to initiate the interaction of the user's PDA with other devices, such as a Public Display. Thus, the component for transferring information could be aware of whether the source device was available for receiving information. Then, a client-server connection was established for the transferring of files among the source and target devices. For the second case, the option of using a peer to peer connection for communicating information feedback between the user's PDA and the Public-display was analyzed, which avoided decreasing the performance of the users' interaction.

Autonomous agents may need alternative communication channels. Agents may decide which channel to use based on the type of information to be communicated, the type of interaction required, or the context of the user's interaction. Thus, context-aware communication channels may be activated in order for agents to effectively exchange information or interact with other agents and users.

7.2.4 Deriving secondary context information

During the evaluation of SALSA, presented in Chapter 7, one of the issues that arose was that the middleware needs to facilitate the creation of the XSL filter which enables the agents to predict the context situation based on a simple set of conditions. Besides that, the context information that can be derived by SALSA is limited to the conditions that can be specified in the XSL filter.

Proposing this mechanism for deriving context information, is an initial step for exploring more complex mechanisms that could be incorporated into SALSA for allowing agents to infer secondary context information. For instance, Artificial Intelligence algorithms could be used in order to learn of the users' interactions with the environment, and predicting what information or services they may need.

7.3 Conclusions

This thesis explored the use of autonomous agents to deal with some of the complexities abstracted from ubicomp application scenarios. Scenarios were used as a tool to define the functionality that can be supported by autonomous agents, identify how autonomous agents augment users' activities, and illustrate how autonomous agents can be the design abstraction of ubicomp systems. To address the design requirements for using autonomous agents to develop ubiquitous computing systems, an agent middleware that enables their development was designed and implemented.

The proposed agent middleware facilitates the implementation and evolution of ubiquitous computing systems in which autonomous agents are the proactive components that enable users to seamlessly and opportunistically interact with the ubicomp environment.

Thus, autonomous agents were used as a technique to model and design ubiquitous computing systems since they provide a natural and elegant means to manage the system's complexities and to integrate new functionality.

Bibliography

Aamodt, A. and Plaza, E. 1994. *"Foundational Issues, Methodological Variations, and System Approaches."* AI Communications, IOS Press, 7(1): 39-59 p.

Abowd, G.D. and Mynatt, E.D. 2000. *"Charting Past, Present, and Future Research in Ubiquitous Computing"*. ACM Transactions on Computer-Human Interaction, 7(1): 29–58

Al-Muhtadi, J., Ranganathan, A., Campbell, R., and Mickunas, M.D. 2002. *"A Flexible, Privacy-Preserving Authentication Framework for Ubiquitous Computing Environments"*. In Proceedings of International Workshop on Smart Appliances and Wearable Computing (IWSAWC) part of ICDCS 2002, Vienna, Austria. 771-776 p.

Amaya, I., Favela, J., and Rodríguez, M.D. 2005. *"Componentes de software para el desarrollo de ambientes de cómputo ubicuo"*. In Proceedings of Ubiquitous Computing and Ambient Intelligence (UCAMI). Granada, Spain. September 13-16. Ed. Thomson. 173-180 p.

Ametler, J., Robles, S., and Borrel, J. 2003. *"Agent Migration over FIPA ACL Messages"*. In Proceeding of Mobile Agents for Telecommunication Applications (MATA). Marakech, Morocco. October 8-10. LNCS 2881, Springer-Verlag. 210-219 p.

Bahl, P. and Padmanabhan, V.N. 2000. *"RADAR, An In-Building RF-Based Location and Tracking System"*. In Proceeding of the Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, IEEE INFOCOM. Tel-Aviv, Israel. March 26 - 30. 775-784 p.

Banavar, G. 2000. *"Challenges: An Application Model for Pervasive Computing"*. 6th. Annual Intl. Conference on Mobile Computing and Networking (MobiCom). Boston, MA USA. August 6-11. 266-274 p.

Banavar, G. and Bernstein, A. 2002. *"Software Infrastructure and Design Challenges"*. Communications of the ACM, 45(12): 92-96 p.

Bardram, J., Kjær, R.E., and Pedersen, M.Ø. 2003. *"Context-Aware User Authentication - Supporting Proximity-Based Login in Pervasive Computing"*. Fifth International Conference on Ubiquitous Computing (UbiComp). Seattle, Washington. October 12-15. LNCS 2864, Springe-Verlag. 107-123 p.

Bardram, J.E. and Bossen, C. 2003. *"Moving to get aHead: Local Mobility and Collaborative Work"*. In Proceeding of the European Conference on Computer Supported Cooperative Work (ECSCW). Klüwer Academic Publishers: Dordrecht, Boston, London. 355-374 p.

Bergenti, F. and Poggi, A. 2001. *"LEAP: A FIPA Platform for Handheld and Mobile Devices"*. Presented at Agent Theories, Architectures, and Languages. The Eight International Workshop (ATAL). Seattle, WA. August 1-3. LNAI 2333, Volume package: Intelligent Agents, Springer-Verlag. 436-446 p.

Bettstetter, C. and Renner, C. 2000. *"A comparison of Service Discovery Protocols and Implementation of the Service Location Protocol"*. In Proceedings of the 6th Open European Summer School: Innovative Internet Applications Twente (EUNICE). Netherlands. September.

Bossen, C. 2002. *"The Parameters of Common Information Spaces: the Heterogenity of Cooperative Work at a Hospital Ward"* In Proceedings of ACM Conf. on Computer Supported Cooperative Work (CSCW). New Orleans, Louisiana, USA. 16-20 November. 176-185 p.

Bradshaw, J. 1997. *"Software Agents"*. AAAI Press/MIT Press: 3-49 p.

Breemen, A.J.N.v. 2003. *"Integrating Agents in Software Applications"*. In Proceedings of the Agent Technology Workshops LNAI 2692, Springer-Verlag. 343-354 p

Brown, P.J. and Jones, G.J.F. 2001. *"Context-aware Retrieval: Exploring a New Environment for Information Retrieval and Information Filtering"*. Personal and Ubiquitous Computing, 5: 253-263 p.

Budzik, J. and Hammond, K. 2000. *" User Interactions with Every Applications as Context for Just-in-time Information Access"*. In Proceedings of Intelligent User Interfaces 2000. New Orleans, LA, USA January 9-12. ACM Press. 44-51 p.

Buzko, D., Lee, W., and Helal, A. 2001. *"Decentralized Ad-Hoc Groupware API and Framework for Mobile Collaboration"*. Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work, Boulder, Colorado. 5-14 p.

Caire, G., Lhuillier, N., and Rimassa, G. 2002. *"A communication protocol for agents on handheld devices"*. Presented at the First International Joint Conference on Autonomous Agents and Multiagents Systems (AAMAS), Bologna, Italy.

Campo, C. 2002. *"Service Discovery in Pervasive Multi-Agent Systems"*. Presented at the First International Joint Conference on Autonomous Agents and Multiagents Systems, (AAMAS) Bologna Italy.

Capra, L., Emmerich, W., and Mascolo, C. 2003. *"CARISMA: Context-Aware Reflective Middleware System for Mobile Applications"*. IEEE Transactions on Software Engineering, 29(10): 929-945 p.

Carolis, B.D. and Pizzutilo, S. 2002. *"A MultiAgent Infraestructure supporting Personalized Interaction with Smart Environments"*. In Proceedings of the First International Joint Conference on Autonomous Agents and Multiagents Systems (AAMAS), Bologna, Italy.

Carrol, J.M. 1995. *"Scenario-Based Design"*. John Wiley & Sons. London. 368 pp.

Carrol, J.M. 2000. *"Making Use: Scenario-Based Design of Human-Computer Interactions"*. MIT Press, Cambridge, MA. 376 pp.

Casebeer, L.L., Strasser, S.M., Spettell, C., Wall, M.T., Weissman, N., Ray, M.N., and Allison, J.J. 2003. *"Designing Tailored Web-Based Instruction to Improve Practicing Physicians' Preventive Practices"*. Journal of Medical Internet Research, 5(3): p.

Castro-Oliveira, I., Belo, O., and Cunha, J.P. 2000. *"Agents Working on the Integration of Heterogeneous Information Sources in Distributed Healthcare Environments"*. In Proceedings of International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI, IBERAMINA-SBIA. Atibaia, SP, Brazil. November 19-22. Springer-Verlag. 136-145 p.

Cortés, U., Annicchiarico, R., Vázquez-Salceda, J., Urdiales, C., Cañamero, L., López, M., Sánchez-Marré, M., and Caltagirone, C. 2003. *"Assistive technologies for the disabled and for the new generation of senior citizens: the e-Tools architecture"*. AI Communications, IOS Press, 16: 193-207 p.

Saha, D. and A. Mukherhee. 2003. *"Pervasive Computing: A Paradigm for the 21st Century"*. IEEE Computer, 36: 25-31 p.

Davies, F., Bagozzi, R., and Warshaw, P. 1991. *"User Acceptance of Information Technology: A Comparison of Two Theoretical Models"*. Management Science, 35(8): 982-1003 p.

Davies, N. and Gellersen, H.-W. 2002. *"Beyond Prototypes: Challenges in Deploying Ubiquitous Computing Systems"*. IEEE Pervasive Computing, 1(1): 26-35 p.

Dey, A.K. 2000. *Providing Architectural Support for Building Context-Aware Applications. PhD dissertation.* Georgia Institute of Technology.

Dey, A.K. 2001. *"Understanding and Using Context"*. Personal and Ubiquitous Computing. Springer-Verlag. London, UK. 4-7 pp.

Dourish, P. 2004. *"What we talk about when we talk about context"*. Personal and Ubiquitous Computing, 8: 19-30 p.

Edwards, K., Bellotti, V., Dey, A.K., and Newman, M. 2003. *"Stuck in the Middle: The Challenges of User-Centered Design and Evaluation for Middleware"*. In Proceedings of the Conference on Human Factors in Computing Systems (CHI). Lauderdale, FL. April 5-10. ACM Press, CHI Letters 5(1): 297-304 p.

Favela, J., Rodríguez, M., Alba, M., and Morán, A.L. 2002. *"Supporting Opportunistic Interacting with People, Resources and Agents in Ubiquitous Enviroments"*. Presented in Mobile Ad-hoc Collaboration Workshop at CHI 2002, Minneapolis, Minnesota.

Favela, J., Rodríguez, M., Preciado, A., and González, V.M. 2004. *"Integrating Context-aware Public Displays into a Mobile Hospital Information System"*. IEEE Transactions on Information Technology in Biomedicine, 8(3): 279-286 p.

Fersha, A. 2002. *"Context-aware: Bridging Physical and Virtual Worlds"*. In Proceedings of Reliable Software Technologies - Ada-Europe. Vienna, Austria. June 17-21. LNCS 2361, Springer-Verlag, 51-64 p.

Finin, T., Fritzson, R., McKay, D., and McEntire, R. 1994. *" KQML as an Agent Communication Language"*. In Proceedings of Third International Conference on Information and Knowledge Management. Gaithersburg, Maryland, USA. November 29 - December 02. ACM Press. 456-463 p.

Golden, G. 2002. *"Service Advertisement and Discovery: Enabling Universal Device Cooperation"*. IEEE Internet Computing: 18-26 p.

Greenwood, S., Nealon, J.L., and Marshall, P. 2003. *"Agent-Based User Interface Adaptivity in a Medical Decision Support System"*. Applications of Software Agent Technology in the Health Care Domain. Whitestein Series in Software Agent Technologies, Birkhäuser Springer-Verlag, Basel, 35-48 pp.

Grimm, R. 2004. *"One.world: Experiences with a Pervasive Computing Architecture"*. IEEE Pervasive Computing, 3(3): 22-30 p.

Griss, M.L. and Pour, G. 2001. *"Accelerating Development with Agent Components"*. IEEE Computer: 37-43 p.

Hansmann, U., Merk, L., Nicklous, M.S., and Stober., T. 2001. *"Pervasive Computing Handbook"*. Springer-Verlag.

Hgo, H.Q., Shehzad, A., Liaquat, S., Riaz, M., and Lee, S. 2004. *"Developing Context-Aware Ubiquitous Computing Systems with a Unified Middleware Framework"*. In Proceedings of International Conference on Embedded and Ubiquitous Computing (EUC). Aizu, Japan. August 25-27. LNCS 3207, Springer-Verlag. 672-681 p.

J. M. Carrol, 2000. 2000. *"Making Use: Scenario-Based Design of Human-Computer Interactions"*. Cambridge, Massachusetts; London England: The MIT Press.

Jennings, N.R. 2001. *"An Agetn-based Approach for Building Complex Software Systems"*. Communications of the ACM, 44(4): 35-41 p.

Kay, A. 1990. *"User Interface: A personal view"*. In B. Laurel (ed.): The Art of Human-Computer Interface Design. Addison-Wesley, Reading, Mass. 190 pp.

Kim, G., Shin, D., and Shin, D. 2004. *"Design of a Middleware and HIML (Human Interaction Markup Language) for Context Aware Services in a Ubiquitous Computing Environment"*. In Proceedings of International Conference on Embedded and Ubiquitous Computing (EUC). Aizu, Japan. August 25-27. LNCS 3207, Springer-Verlag, 682-691 p.

Kindberg, T. and Fox, A. 2002. *"System Software for Ubiquitous Computing"*. IEEE Pervasive Computing, 1(1): 70-81 p.

Kirn, S. 2003. *"Ubiquitous Healthcare: The OnkoNet Mobile Agents Architecture"*. In Proceedings of International Conference NetObjectDays (NODe). Erfurt, Germany. October 7-10. LNCS 2591, Springer-Verlag. 265-277 p.

Klemmer, S.R., Li, J., Lin, J., and Landay, J.A. 2004. *"Papier-Maché: Toolkit Support for Tangible Input"*. In Proceedings of Human Factors in Computing Systems (CHI). Vienna, Austria. ACM Press. 399-406 p.

Koukoumpetsos, K. and Antonopoulos, N. 2002. *"Mobility Patterns: An Alternative Approach to Mobility Management"*. In Proceedings of the 6th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI), Orlando , Florida, USA. 14-18 July.

Labrou, Y., Finin, T., and Peng, Y. 1999. *"Agent Communication Languages: The Current Landscape"*. IEEE Intelligent Systems, 14(2): 45-52 p.

Labrou, Y. 2001. *"Standardizing Agent Communication"*. Advanced Course on Artificial Intelligence (ACAI-01). LNCS 2086, Springer-Verlag. 74-97 p.

Laukkanen, M., Helin, H., and Laamanen, H. 2002. *"Tourists on the Move"*. In Proceedings of International Workshop Series on Cooperative Information Agents (CIA). Madrid, Spain. September 18-20. Springer-Verlag. 36-50 p.

Lesser, V. 1999. *"Cooperative Multiagent Systems: A Personal View of the State of the Art"*. IEEE Transactions on Knowledge and Data Engineering, 11(1): 133-141 p.

Litiu, R. and Parkash, A. 2000. *"Developing Adaptive Groupware Applications Using a Mobile Computing Framework"*. In Proceedings of Computer Supported Cooperative Work (CSCW). Philadelphia, Pennsylvania, USA. December 2-6. ACM Press. 107-116. p.

Maes, P. 1994. *"Agents that Reduce Work and Information Overload"*. Communications of the ACM, 37(7): 30-40 p.

Makoto A. 2005. *"Agent-Oriented Approach to Ubiquitous Computing"*. In Proceedings of the First International Conference on Embedded Software and Systems, (ICESS 2004) Revised Selected Papers. Hangzhou, China. LNCS 3605, Springer-Verlag. 30-37p.

Moreno, A., Valls, A., and Riaño, D. 2004. *"Medical Applications of Multi-Agent Systems"*. Presented at ECAI Workshop on Agents Applied in Health Care. Valencia, Spain. August 22-27.

Muñoz, M.A., Gonzalez, V.M., Rodríguez, M., and Favela, J. 2003a. *"Supporting Context-aware Collaboration in a Hospital: an ethnographic informed design"*. In Proceedings of

Workshop on Artificial Intelligence, Information Access, and Mobile Computing 9th International Workshop on Groupware (CRIWG). Grenoble, France. Springer-Verlag. 330-344 p.

Muñoz, M.A., Rodriguez, M., Favela, J., Martinez-García, A.I., and Gonzalez., V.M. 2003b. *"Context-aware mobile communication in hospitals"*. IEEE Computer, 36(9): 38-46 p.

Odell J. J. 2002. "*Objects and Agents Compared*". Journal of Object Technology. 1(1) May-June. 41-53 p.

Oviatt, S. and Cohen, P. 2000. *"Multimodal Interfaces that Process what Comes Naturally"*. Communications of the ACM, 43(3): 45-53 p.

Parunak, H.V.D. and Odell, J. 2001. *"Representing Social Structures in UML"*. In Proceedings of the 5th International Conference on Autonomous Agents. ACM Press. 100-101 p.

Pizzutilo, B.D.C.a.S. 2002. *" A MultiAgent Infraestructure supporting Personalized Interaction with Smart Environments"*. Presented at First International Joint Conference on Autonomous Agents and Multiagents Systems (AAMAS), Bologna, Italy.

Popovici, A., Frei, A., and Alonso, G. 2003. *"A Proactive Middleware Platform for Mobile Computing"*. In Proceedings of International Middleware Conference. Rio de Janeiro, Brazil. June 16-20. LNCS 2672, Springer. 455-473 p.

Ranganathan, A. and Campbell, R.H. 2003. *"An infrastructure for context-awareness based on first order logic"*. Personal and Ubiquitous Computing, 7: 353-364 p.

Reddy, M. and Dourish, P. 2002. *"A Finger on the Pulse: Temporal Rhythms and Information Seeking in Medical Work"*. Computer Supported Cooperative Work (CSCW). New Orleans, Louisiana, USA. November 16-20. ACM Press. 344-353 p.

Rist, T. 2004. *"Issues in Designing Human-Computer Interaction in AmI"*. Extended abstracts of the 2004 Conference on Human Factors in Computing Systems (CHI). Lost in Ambient Intelligence Workshop. Vienna, Austria. April 24-29. ACM Press. 1725-1726 p.

Riva, G., Vatalaro, F., Vatalaro, F., Davide, F., and Alcañiz M. (Edts.). 2005 *"Ambient Intelligence: The evolution of technology, communication and cognition towards the future of human-computer interaction"*. IOS Press. 295 pp.

Rodriguez, M.D., Favela, J., Martínez, E.A., and Muñoz, M.A. 2004. *"Location-aware Access to Hospital Information and Services"*. IEEE Transactions on Information Technology in Biomedicine, 8(4): 448- 455 p.

Román, M., Hess, C., Cerqueira, R., Ranganatha, A., Campbell, R.H., and Nahrstedt, K. 2002. *"A Middleware Infrastructure for Active Spaces"*. IEEE Pervasive Computing, 1(4): 74-83 p.

Saha, D. and Mukherhee, A. 2003. *"Pervasive Computing: A Paradigm for the 21st Century"*. IEEE Computer, 36: 25-31 p.

Saif, U. and Greaves, D.J. 2001. *"Communication Primitives for Ubiquitous Systems or RPC Considered Harmful"*. Presented at 21st International Conference on Distributed Computing Systems Workshops (ICDCSW '01). Arizona, USA. IEEE Press. 240-245 p.

Satyanarayanan, M. 2001. *"Pervasive Computing: Vision and Chal-lenges"*. IEEE Personal Communications: 10-17 p.

Schilit, B.N. and Theimer, M.M. 1994. *"Disseminating active map information to mobile hosts"*. IEEE Network, 8(5): 22-32 p.

Schoen, D.A. 1983. *"The reflective practitioner: how professionals think in action."* New York: Basic Books.

Shadbolt, N. 2003. *"Ambient Intelligence"*. IEEE Intelligent Systems, 18(4): 2-3 p.

Shoham, Y. 1997. *"An Overview of Agent-oriented Programming"*. Software Agents, Menlo Park, Calif.: AAAI Press.

Singh, S., Ikhwan-Ismail, B., Haron, F., and Yong, C.H. 2004. *"Architecture of Agent-based Healthcare Intelligent Assistnat on Grid Environment"*. In Proceedings of The International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT). Singapore. December 8-10. LNCS 3320, Springer-Verlag. 58-61 p.

Suvee, D., Vanderperren, W., Jonckers, V. *"JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development"*. Aspect-Oriented Software Development Conference (AOSD). Boston, MA USA. March 17 – 21. ACM Press. 21-29 p.

Tablado, A., Illarramendi, A., Bermúdez, J., and Goñi, A. 2003. *"Intelligent Monitoring of Elderly People"*. In Proceedings of the 4th. Annual IEEE Conference on Information Technology Applications in BiomedicineBuckingham, UK. 78-81 p.

Theureau, J., Filippi, G. 2000. *"Analysing cooperative work in a urban traffic control room for the design of a coordination support system"*, In Workplace Studies: Recovering Work Practice and Informing System Design. Edited by Luff, P. et. al. Cambridge Press. 68-91 p.

Tveit, A. 2001. *"A survey of agent-oriented software engineering."* NTNU Computer Science Graduate Student Conference. Report. Norway.

Villate, Y., Illarramendi, A., and Pitoura, E. 2002. *"Keep your data safe and available while roaming"*. In Proceedings of Mobile Networks and Applications (MONET), ACM Press. 315-328 p.

Wang, X., Song-Dong, J., Yau-Chin, C., and Ravipriya-Hettiarachchi, S. 2004. *"Semantic Space: An Infraestructure for Smart Spaces"*. IEEE Pervasive Computing, 3(3): 32-39 p.

Wegner, P. 1990. Concepts and Paradigms of Object-Oriented Programming. Keynote Talk of OOPSLA. ACM SIGPLAN OOPS Messenger. ACM Press. New York, USA. 1(1): 3-87p.

Weiser, M. 1991. *"The Computer for the Twenty-First Century"*. Scientific American, 265(3): 94-104 p.

Weiser, M. 1993. *"Some Computer Science Issues in Ubiquitous Computing."* Communications of the ACM, Special issue on computer augmented environments: back to the real world, 36(7): 75-84 p.

Weiser, M. and Brown, J. 1996. *"Designing Calm Technology"*. PowerGrid Journal, 1(1).

Wooldridge, M. and Jennings, N. 1995. *"Intelligent Agents: Theory and Practice"*. Knowledge Engineering Review, Cambridge University Press, 10(2): 115-152 p.

Wooldridge, M. 1997. *"Agent-based software engineering"*. IEE Proc. Software Engineering, 144(1): 26-37 p.

Wooldridge, M.J. and Jennings, N.R. 1999. *"Software Engineering with Agents: Pitfalls and Pratfalls"*. IEEE Internet Computing: 20-27 p.

Yau, S., Karim, F., Wang, Y., Wang, B., and Gupta, K.S. 2002. *"Reconfigurable Context-Sensitive Middleware for Pervasive Computing"*. IEEE Pervasive Computing, 1(4): 33-40 p.

Zimmermann, G., Vanderheiden, G., Gandy, M., Laskowski, S., Ma, M., Trewin, S., and Walker, M. 2004. *"Universal remote console standard - toward natural user interaction in ambient intelligence"*. Extended Abstracts on Human Factors in Computing Systems, (CHI), Vienna, Austria. 1608-1609 p.

Appendix A

API of SALSA

This Section presents the API for facilitating the implementation of agents for ubiquitous computing environments. This API includes two class libraries: SALSA which was implemented in Java (jdk 1.2) for creating agents that can be executed in any computing platform and mSalsa which was implemented in C# for creating agents for mobile devices.

The classes of the API are grouped according to the functionality that the agent implements. At the top of each class shows the name of the library in which it can be found, i.e. mSalsa, SALSA, Client; followed by a description of the class and the fields and methods it provides.

A.1 Creating and activating an Agent

mSalsa, SALSA

Class Agent

public class **Agent**

An agent is a component of a ubicomp system. It may represent users, act as a proxy to devices or services, or wrap complex system functionality. An `Agent` class must be a superclass of any agent that is a component of a ubiquitous computing system.

The following is an example that illustrates how to create an Agent that perceives information through the Agent Broker.

Example A:

```
import SALSA.*;
import Client.*;
public class HelloWorldAgent extends Agent{
    JabberClient jc; //JabberClient acting as proxy to the Agent
Broker
    public HelloWorldAgent(){
        jc=new JabberClient("pc-
coolab5","5222","marcerod","anro1224"); //Instant messaging client
        HelloWorldReasoning reasoning=new HelloWorldReasoning();
        activate(reasoning, jc);
    }

    public static void main(String[] args) {
        HelloWorldAgent agentito1 = new HelloWorldAgent();
        }
    }
```

Fields

attributes

protected SALSA.AgentAttributes **attributes**

Contains the attributes that characterizes an Agent.

---

acting

protected SALSA.Acting **acting**

The `Acting` object is the agent's component through which the actions of the agent are invoked.

---

passivePerception

public SALSA.PassivePerception **passivePerception**

The `PassivePerception` object is the agent's component through which the agent perceives information.

---

jc

protected SALSA.ProxyBroker **jc**

The `ProxyBroker` is the object that enables the agent's to communicate through the Agent Broker.

---

reasoning

protected SALSA.Reasoning **reasoning**

Contains the implementation of the agent's reasoning. This object is automatically instantiated in the `activate` method.

Constructors

Agent

public **Agent**()

Creates an Agent instance

Methods

activate

```
protected void activate(SALSA.Reasoning rsn,
                        SALSA.ProxyBroker pb)
```

Creates the agent's components and then activates the agent which perceives information and communicates by sending XML messages through the `ProxyBroker` that is passed as an argument. The `Reasoning` object that is passed as an argument should contain the reasoning algorithm for this agent.

Parameters:

`rsn` - Reasoning

`pb` - ProxyBroker

activate

```
protected void activate(java.lang.String typeAgent,
                        SALSA.Reasoning rsn,
                        SALSA.ProxyBroker pb)
```

Creates the agent's components and then activates the agent. This agent perceives information and communicates through the `ProxyBroker` that is passed as an argument. The `typeAgent` argument indicates the type of agent, such as "user" or "service". The value of this argument can be specified by passing attributes.USER, attributes.SERVICE, or any other value indicated by the user. This value will be part of the agent’s attributes which will be registered in the Agent Directory. The `Reasoning` object specifies the reasoning algorithm.

Parameters:

`type Agent` - the type of Agent: i.e. "user", "service"

`rsn` - Reasoning

`pb` - ProxyBroker

activate

```
protected void activate(SALSA.Reasoning rsn)
```

Creates the agent's components and then activates the agent which directly perceives information from a sensor, device or service, and directly communicates with other agents. The `Reasoning` object specifies the reasoning algorithm.

Parameters:

`rsn` - Reasoning

A.2 Perception component

mSALSA, SALSA

Class PassivePerception

public class **PassivePerception**

> `PassivePerception` component of the Agent. When the Agent is created, an object of this class is automatically created.

Methods

perceive

`public void **perceive**(SALSA.Input input)`

> This method should be invoked by the `PassiveEntityToPerceive` object to notify the received information to this `PassivePerception` component. When a SALSA message is perceived, this method generates a SALSA event and notifies it to the reasoning component of the agent.

Parameters:

> `input` - the `input` argument wraps the information perceived, which can be an XMLpresence, XMLmessage, or any other kind of object

SALSA

Class PassiveEntityToPerceive

public class **PassiveEntityToPerceive**

> Represents the entity from which the `PassivePerception` will perceive. In passive sensing or perception, the entity takes the initiative to send data to the agent.
> The passive perception component perceives information from a `PassiveEntityToPerceive` object attached to the entity from which the Agent needs to receive information. For instance, if the Agent will perceive information from an instant messaging client (the proxy to the Agent Broker), it needs to embed a `PassiveEntityToPerceive` object as illustrated in the following code:

```
-->THE PassiveEntityToPerceive OBJECT IS INHERETED BY THE
-->ProxyBroker CLASS
public class JabberClient extends SALSA.ProxyBroker{
    ..... private SALSA.Input in;
```

```
public JabberClient(String server, String port, String userName,
    String password) {
    .......
    connect();
->START THREAD FOR READING INCOMING DATA
    incomingData();
}

private synchronized void incomingData(){
    while(true){
        String xml = "";
        xml = read();
-->WHEN A NEW MESSAGE ARRIVES:
        parse(xml);
    }
}

public void parse(String xml){
-->PARSE THE MESSAGE....
-->VERIFY THE TYPE OF MESSAGE:
    if (tag.equalsIgnoreCase("message") == true) {
        SALSA.XMLmessage xmlMessage = new
        SALSA.XMLmessage(nl.item(i).toString());
        in = new SALSA.Input(xmlMessage);
       passiveEntity.notifying(in);
    }
    else
    if (tag.equalsIgnoreCase("presence") == true) {
        SALSA.XMLpresence xmlPresence = new
        SALSA.XMLpresence(nl.item(i).toString());
        in = new SALSA.Input(xmlPresence);
        passiveEntity.notifying(in);
         .....
    }
     .....
   }
```

NOTE: The above code is part of the implementation of the proxy to the Agent Broker, or IM client, provided by SALSA which has the perception component implemented already.

The following is an example of a class that reads data from a sensor or device and notifies the data to the agent's perception component.

Example B:

```
import SALSA.*;
public class HelloWorldSensingData {
    PassiveEntityToPerceive passiveEntity;

    public HelloWorldSensingData(Agent agent)
    {
        passiveEntity=new PassiveEntityToPerceive();
        passiveEntity.attach(agent.passivePerception);
    }

    public void readDataFromSensor()
    {
     -->INSERT CODE CREATING A THREAD THAT READS DATA FROM A SENSOR

         passiveEntity.notifying(new Input(sd));
    }
}
```

Methods

attach

public void **attach**(SALSA.PassivePerception pp)

Specifies the PassivePerception component to which the perceived information has to be notified.

Parameters:

pp - the PassivePerception component

detach

public void **detach**(SALSA.PassivePerception pp)

Removes the PassivePerception component from this entity

Parameters:

pp - the PassivePerception component

notifying

public void **notifying**(SALSA.Input input)

This method is invoked from the ProxyBroker each time a message arrives. Then the method passes the message which contains the perceived information to the PassivePerception component of the Agent.

Parameters:

input – the object that contains the message

mSalsa, SALSA

Class Input

public class **Input**

An object `Input` contains the data perceived through the `passiveEntityToPerceive` object.

Fields

data

```
private java.lang.Object data
```

It is the perceived data.

Constructors

Input

```
public Input(java.lang.Object data)
```

Creates an `Input` object with the data passed as an argument.

Methods

setData

```
public void setData(java.lang.Object data)
```

Sets the data object.

getData

```
public java.lang.Object getData()
```

Return the value of the `data` object.

mSalsa, SALSA

Class XMLmessage

public class **XMLmessage**

Contains the xml message received through the Agent Broker

Fields

msg

`java.lang.String` **`msg`**

It is the XML message.

Constructors

XMLmessage

`public` **`XMLmessage`**`(java.lang.String msg)`

Creates an instance of the `XMLmessage` class containing the `msg` variable.

Parameters:

`msg` - The xml message

mSalsa, SALSA

Class XMLpresence

public class **XMLpresence**

Contains a presence message perceived through the Agent Broker.

Fields

presence

`java.lang.String` **`presence`**

Contains the presence message

Constructors

XMLpresence

`public` **`XMLpresence`**`(java.lang.String presence)`

Creates an `XMLpresence` object containing the presence message passed as an argument

SALSA

Class SensorData

public class **SensorData**

An instance of the `SensorData` class contains the data perceived from a sensor.

Fields

data

`public java.lang.Object` **`data`**

The perceived data.

Constructors

SensorData

`public` **`SensorData`**`(java.lang.Object data)`

Creates a SensorData object containing the data perceived

Parameters:

`data` – an object that contains the data perceived

Methods

getData

`public java.lang.Object` **`getData`**`()`

Returns the perceived data.

Returns:

Object containing the data

A.3 Acting component

mSalsa, SALSA

Class Acting

public class **Acting**

Acting is the component of the Agent that enables the reasoning component to execute an action.

Methods

act

public void **act**(SALSA.Action action)

Executes the action indicated as an argument. This method should be invoked from the agent's reasoning component.

Parameters:

action - the action to be executed

---

mSalsa, SALSA

Class Action

public abstract class **Action**

This class implements the agent's actions which may involve communicating with other agents through the Agent Broker. The following is an example of the implementation of the Action component:

```
import SALSA.*;

public class HelloWorldAction extends Action {
    public Object execute(){
-> INSERT CODE FOR AGENT'S ACTION, WHICH MAY INCLUDE COMMUNICATING
WITH ANOTHER AGENT:
    sendCommandRequest("map_agent@jabber","","display","internal
medicine");
    ......
    return null;
   }
}
```

Methods

execute

public abstract java.lang.Object **execute**()

This method should be overwritten to implement the agent's actions.

Returns:

Object - an object containing information related with the execution of the action. The content of this object should be defined by developers.

---

sendRequest

```
public void sendRequest(java.lang.String to,
                        java.lang.String body,
                        java.lang.String type,
                        java.lang.String params)
```

Sends a message to request information from another Agent

Parameters:

`to` - the id of the Agent

`body` - a message to be displayed in a GUI of the agent. This argument is optional.

`type` - Type of the information to be requested. This is defined by the developer.

`params` - XML tags defined by the user. They contain information specifying other characteristics that the information requested should have.

The format of the XML message sent by this method is:

```
<message to='agentA@jabber_server'
    from='agentB@jabberserver'>
    <body>a message </body>
    <x xmlns='x:request'>
      <params><type>Type of information requested</type>
          Other tags specifying the parameters necessary for this

        request
     </params>
   </x>
</message>
```

---

sendResponse

```
public void sendResponse(java.lang.String to,
                         java.lang.String body,
                         java.lang.String type,
                         java.lang.String params)
```

Sends a message that responds to a request message sent by another Agent

Parameters:

`to` - id of the Agent.

`body` - a message to be displayed in a GUI of the agent. This argument is optional.

`type` - Type of the information send to another Agent.

`params` - XML tags defined by the user. They contain the information requested by another agent.

The format of the message sent by this method is:

```
<message to='agentA@jabber_server'
    from='agentB@jabberserver'>
    <body>a message </body>
    <x xmlns='x:response'>
        <params><type>Type of the information sent as a
response</type> Other tags specifying the parameters necessary for
this response
       </params>
    </x>
</message>
```

sendNotificationInfo

```
public void sendNotificationInfo(java.lang.String to,
                                 java.lang.String body,
                                 java.lang.String type,
                                 java.lang.String params)
```

Sends a message to another agent notifying that an event happened or notifying new information relevant for the execution of the other agent.

Parameters:

`to` - id of the Agent.

`body` - a message to be displayed in a GUI of the agent. This argument is optional.

`type` - Type of notification sent to another Agent.

`params` - XML tags defined by the user. They contain the information requested by another agent.

The format of the message sent by this method is:

```
<message to='agentA@jabber_server'
     from='agentB@jabberserver'>
     <body>a message </body>
     <x xmlns='x:notificationInfo'>
         <params><type>Type of notification</type>
           Other tags specifying the parameters necessary for this
notification
        </params>
     </x>
</message>
```

sendCommandRequest

```
public void sendCommandRequest(java.lang.String to,
                               java.lang.String body,
                               java.lang.String command,
                              java.lang.String params)
```

Request the execution of an action or service from another Agent

Parameters:

`to` - id of the Agent

`body` - a message to be displayed in a GUI of the agent. This argument is optional

`command` - the name of the requested command or service to be executed

`params` - XML tags specifying data needed for requesting the execution of an action or service

The format of the message sent by this method is:

```
<message to='agentA@jabber_server'
     from='agentB@jabberserver'>
     <body>a message </body>
     <x xmlns='x:command'>
        <params><type>Type of action or service</type>
Other tags specifying the parameters necessary for the execution of
the action
       </params>
    </x>
</message>
```

---

sendContextualMsg

```
public void sendContextualMsg(java.lang.String to,
                              java.lang.String body,
                              java.lang.String type,
                              java.lang.String params)
```

Sends a contextual message to other Agent

Parameters:

`to` - id of the Agent

`body` - a message to be displayed in a GUI of the agent. This argument is optional

`type` - Type of the message

`params` - Set of xml tags specifying other information the other agent needs to send the message

The format of the message sent by this method is:

```
<message to='agentA@jabber_server'
     from='agentB@jabberserver'>
     <body>a message </body>
     <x xmlns='x:contextual message'>
        <params>Your tags specifying the parameter </params>
     </x>
</message>
```

sendPresence

```
public void sendPresence(java.lang.String status,
                         java.lang.String nickUser,
                         java.lang.String area,
                         java.lang.String location,
                         java.lang.String typeOfAgent)
```

Sends a presence message

Parameters:

`status` - State of the agent: online, available, ...

`nickUser` - Name of the agent

`area` - Area of the ubicomp environment in which this agent is located

`location` - Specific place of the area (i.e. No. of room) in which this agent is located

`typeOfAgent` - Type of agent: user, service, device

The format of the message sent by this method is:

```
<presence>
    <params><area>areaName</area>
       <location>location</location>
       <type>typeOfAgent</type>
    </params>
</presence>
```

sendDataSensor

```
public void sendDataSensor(SALSA.PassiveEntityToPerceive passiveEntity,
                           SALSA.SensorData sd)
```

Sends the data perceived directly to the passive perception of another Agent

Parameters:

`passiveEntity` - PassiveEntityToPerceive entity from which another Agent perceives information

`sd` - SensorData object containing the data perceved from the sensor attached to this Agent.

sendDataSensor

```
public void sendDataSensor(java.lang.String to,
                           java.lang.String body,
                     java.lang.String type,
                     java.lang.String params)
```

Sends the data perceived from a sensor to another Agent. This data is sent through the Agent Broker.

Parameters:

`to` - String instant messagin id of the other Agent
`body` - String a simple message. This parameter is optional.
`type` - String the type of information. For instance: noiseLevel
`params` - String information describing the perceived data.

The format of the message sent by this method is:

```
<message to='agentA@jabber_server'
     from='agentB@jabberserver'>
     <body>a message </body>
     <x xmlns='x:dataFromSensor'>
        <params><type>Type of data</type>
Other tags providing information of the data perceived
       </params>
    </x>
</message>
```

A.4 Reasoning component

mSalsa, SALSA

Class Reasoning

public abstract class **Reasoning**

Creates the `Reasoning component of the Agent.`

The following is an example of a Reasoning component:

```
import SALSA.*;
import java.util.*;
public class HelloWorldReasoning extends Reasoning{
    public void think(EventObject ev){
->INSERT THE REASONING CODE, WHICH MAY INCLUDE VERIFY
THE RECEIVED INFORMATION IN ORDER TO DECIDE HOW TO ACT:
......
    SALSA.Events.Event event=(SALSA.Events.Event)ev;
    if (event.getType()==event.ArriveSensorDataEvent){
        ......

       ->INVOKE THE ACTION(S):
       acting.act(new HelloWorldAction(dat_int));
```

```
          }
        }
      }
```

Fields

acting

```
public SALSA.Acting acting
```

The Acting component of the Agent.

Methods

think

```
protected abstract void think(java.util.EventObject ev)
```

This method should be overwritten to specify the reasoning algorithm of the Agent. This is automatically invoked in the perception component when it perceives information.

Parameters:

ev - the EventObject contains the perceived data in the perception component

SALSA

Class SecondaryContext

```
public class SecondaryContext
```

Provides the methods to derive secondary context information from the perceived context information. As illustrated in the following example, the code for obtaining secondary context information is implemented in the Reasoning component.

```
      public class patientAgentReasoning extends Reasoning{
          SALSA.SecondaryContext sc;
          public patientAgentReasoning()extends Reasoning{
             String xslFile = "C:\\Documents and
      Settings\\Usuario\\jbproject\\patientagent\\secondaryContext.xsl";
             sc = new SALSA.SecondaryContext(xslFile);
            -->SPECIFY THE PRIMARY CONTEXTUAL VARIABLES:
             sc.setContextualVariable("glucose", "80");
             sc.setContextualVariable("pulsations", "0");
             sc.setContextualVariable("perspiring", "0");
           }

           public void think(EventObject ev){
                SALSA.Events.Event event = (Event)ev;
                String secondaryContext = "";
                if (event.getType()==event.ArriveSensorDataEvent){
      -->WHEN NEW CONTEXT INFORMATION IS PERCEIVED, THE AGENT
      DERIVES SECONDARY CONTEXT INFORMATION
                   secondaryContext = sc.derive(ev);
                   Utilidades.Parser p =
                      new Utilidades.Parser(secondaryContext);
                   String contexto = p.getTag("context");
                   String tipo=p.getAtt("context", "type");
                   if (tipo.equals("hypoglycemia")){
                       switch(Integer.parseInt(contexto)){
                     case 0: acting.act(new patientAgentAction());
                       break;
                     case 1: acting.act(new recomend());
                       break;
                     case 2: acting.act(new notifyToHospital());
                       break;
                  }
              }
           }
        }
      }
```

Constructors

SecondaryContext

`public` **`SecondaryContext`**`(java.lang.String xsl)`

Creates an object of the SecondaryContext class that will derive context based on the set of conditions stated in the file passed as an argument.

Parameters:

`xsl` - the name of the file that contains the set of conditions for deriving secondary context information. The name of the file should specify the complete path to the file. For instance:

```
String xslFile = "C:\\Documents and
Settings\\Usuario\\jbproject\\patientagent\\secondaryContext.xsl";
sc = new SALSA.SecondaryContext(xslFile);
```

Methods

setContextualVariable

`public void setContextualVariable(java.lang.String variable)`

Sets a variable that contains primary context information. Indicates to the reasoning component that this contextual information will be used for deriving secondary context. Thus, each time this variable has new information, the reasoning component will analyze it and check the contextual conditions of the xsl file.

Parameters:

`variable` **-** name of the variable which is specified in the `type` argument of the `sendDataSensor` method.

See Also:

Action class for details of the sendDataSensor method.

---

setContextualVariable

```
public void setContextualVariable(java.lang.String variable,
                                  java.lang.String valorPorDefecto)
```

Sets a variable that contains primary context information and initializes it with the indicated default value.

Parameters:

`variable` **-** the name of the variable which is specified in the `type` argument of the `sendDataSensor` method.

`valorPorDefecto` **-** the value for initializing the variable.

---

generatePrimaryContext

```
private java.lang.String generatePrimaryContext(SALSA.Events.Event event)
```

This method should be invoked in the reasoning component when an event is generated, it may indicate that a new primary context information has been perceived by the Agent.

Parameters:

`event` **-** the event that contains the perceived information.

Returns:

an xml message containing the current values of the set of primary contextual variables specified by using the method `setContextualVariable`

---

derive

```
public java.lang.String derive(java.util.EventObject ev)
```

Obtains the secondary contextual information based on the current values of the primary contextual variables.

Parameters:

`ev` **-** the event produced

Returns:

a String with the secondary contextual information in an xml format.

A.5 Initializing and registering the Agent in the Agent Directory

SALSA

Class AgentDirectory

public class **AgentDirectory**

The AgentDirectory provides the methods to communicate with the agent acting as proxy to the Agent Directory. It enables the Agent to request information from the Agent Directory and to register its attributes into it.

Fields

agent

```
private SALSA.Agent agent
```

attributes

```
private java.lang.String attributes
```

Attributes of the agent.

imAddressAD

```
protected java.lang.String imAddressAD
```

Instant Messaging id of the Agent.

Methods

register

```
public void register(java.lang.String typeAgent)
```

Registers the Agent's attributes in the Agent Directory

lookForAllAgents

```
public void lookForAllAgents()
```

Requests information of all agents available in the ubiquitous computing environment.

---

lookForAllAgents

`public void` **`lookForAllAgents`**`(java.lang.String area)`

Requests information of all agents available in a specific area of the ubiquitous computing environment.

Parameters:

`area` - an area of the ubicomp environment.

---

lookForAservice

`public void` **`lookForAservice`**`()`

Requests information of all agents offering services in the ubiquitous computing environment.

---

lookForAservice

`public void` **`lookForAservice`**`(java.lang.String area)`

Requests information of all agents offering services in a specific area of the ubiquitous computing environment.

Parameters:

`area` - an area of the ubicomp environment.

SALSA

Class AgentAttributes

public class **AgentAttributes**

`AgentAttributes` contains information that describes the agent.

Fields

typeAgent

`public java.lang.String` **`typeAgent`**

Defines the type of agent which can represent a user, device, service or wrap a complex system functionality.

---

name

`public java.lang.String name`

This is the name assigned to the agent. For instance: PublicDisplay_Agent

description

`public java.lang.String description`

Contains a brief description of the functionality of the agent.

typeHost

`public java.lang.String typeHost`

Defines the type of computing device in which the agent is executing. For instance, PC, PDA, etc.

urlImage

`public java.lang.String urlImage`

Specifies the location of the image to be used by the ubicomp system to graphically represent the agent. For instance, an image of the device represented by the agent.

area

`public java.lang.String area`

Specifies the area or department of an organization in which the ubicomp system provides information and services to the users. For instance: "internal medicine" of the hospital

floor

`public java.lang.String floor`

Specifies the floor of the building in which the agent is to be executed.

jid

`public java.lang.String jid`

id of the agent. Any agent of the ubicomp environment should have an id to login in the AgentBroker.

Constructors

AgentAttributes

```
public AgentAttributes()
```

Creates an instance of the AgentAttributes for the agent.

Methods

setTypeAgent

```
public void setTypeAgent(java.lang.String type)
```

Sets the type of the Agent.

setName

```
public void setName(java.lang.String name)
```

Sets the name of the Agent.

setDescription

```
public void setDescription(java.lang.String description)
```

Sets the description of the Agent.

setTypeHost

```
public void setTypeHost(java.lang.String typeHost)
```

Sets the type of host in which the Agent resides.

setUrlImage

```
public void setUrlImage(java.lang.String urlImage)
```

Sets the location in where the image used for representing the Agent is available.

setArea

`public void setArea(java.lang.String area)`

Sets the area of the organization in which the ubicom system is executing.

---

setFloor

`public void setFloor(java.lang.String floor)`

Sets the floor on which the Agent provides its services.

---

setJabberID

`public void setJabberID(java.lang.String jid)`

Sets the id of the Agent.

A.6 SALSA Events

mSalsa, SALSA.Events

Class Event

public class **Event**
extends java.util.EventObject

Specifies the type of event generated in tha `PassivePerception` object

Fields

StateChangeEvent

`public final int StateChangeEvent`

Indicates a change of state in this agent. For instance, if the agent represents a device, it can be used to indicate a change of state of the device.

ArriveCommandEvent

`public final int ArriveCommandEvent`

Indicates that a message arrived requesting execution a command or service.

ArriveResponseEvent

`public final int` **`ArriveResponseEvent`**

Indicates that a response to an information request arrived.

ArriveNotificationInfoEvent

`public final int` **`ArriveNotificationInfoEvent`**

Indicates that a notification message arrived.

ArriveRequestEvent

`public final int` **`ArriveRequestEvent`**

Indicates that a message requesting information has arrived.

ArriveContextualMsgEvent

`public final int` **`ArriveContextualMsgEvent`**

Indicates that a contextual message has arrived.

ArriveComponentEvent

`public final int` **`ArriveComponentEvent`**

Indicates that the requested component has arrived.

ArriveSimpleMessageEvent

`public final int` **`ArriveSimpleMessageEvent`**

It is used to indicate that any other type of message arrived.

ArrivePresenceEvent

`public final int` **`ArrivePresenceEvent`**

Indicates that a presence message has arrived.

ArriveSensorDataEvent

```
public final int ArriveSensorDataEvent
```

Indicates that information perceived from a sensor has arrived.

---

type

```
public int type
```

The type variable can be use to specify another kind of event not considered by SALSA.

---

input

```
public SALSA.Input input
```

The `input` object contains the data perceived

---

xml

```
public java.lang.String xml
```

XML message abstracted from the `input` object

---

sensorData

```
public SALSA.SensorData sensorData
```

If the type of the event generated is `ArriveSensorDataEvent`, then the `sensorData` object contains the perceived data

---

Constructors

Event

```
public Event(SALSA.Input in,
             java.lang.Object obj)
      Creates an Event that wraps the Input object with the information
      perceived
Parameters:
      in - Input object that contains the information perceived
      obj - Object in which the event was generated
```

Event

```
public Event(java.lang.String xml,
             java.lang.Object obj)
      Creates an Event that wraps the xml message perceived
Parameters:
      xml - Message perceived through a Client (IM client)
      obj - Object in which the event was generated
```

Event

```
public Event(SALSA.SensorData sd,
             java.lang.Object obj)
      Creates an Event that wraps the data perceived from a sensor (a
      SensorData object)
Parameters:
      xml - Message perceived through a Client (IM client)
      obj - Object in which the event was generated
```

Method Detail

getType

```
public int getType()
      Returns the type of event.
```

getInput

```
public SALSA.Input getInput()
      Returns the Input object containing the perceived data.
```

A.7 Proxy to the Agent Broker

Client (to be used for SALSA agents)

Class JabberClient

public class **JabberClient**
extends SALSA.ProxyBroker

> This is an Instant Messaging client (Jabber client) that can be used as a proxy to the Agent Broker. This enables the SALSA agents to communicate with other agents and users. In order for any other Jabber client to be used as Agent Broker proxy, it

should inherit the methods of the superclass ProxyBroker and overwrite the methods: send() and connect().

See Also:

ProxyBroker class.

Fields

in

```
private SALSA.Input in
```

Stores the perceived information

Constructors

JabberClient

```
public JabberClient(java.lang.String server,
                    java.lang.String port,
                    java.lang.String userName,
                    java.lang.String password)
```

Creates an instance of the JabberClient.

Parameters:

`server` - IP address or hostname of the Jabber server

`port` - number of port of the server

`userName` - username to login to the server

`password` - password to login to the server Example for creating a client:

```
JabberClient jc =
new JabberClient("158.97.22.238","5222","userName","password");
jc.connect();
```

Methods

setHost

```
public void setHost(java.lang.String host)
      To specify the host of the Jabber server
Parameters:
      host - name or ip address of the host
```

connect

```
public void connect()
```

```
      Connects to the Jabber Server specified in the Host variable.
```

disconnect

```
public void disconnect()
      Disconnects from the Jabber Server.
```

send

```
public void send(java.lang.String xml)
      Sends the xml message specified as an argument.
Parameters:
      xml - the xml message
```

mSalsa (to be used for mSalsa agents)

Class JabberClient

public class **JabberClient: XmppClientConnection**

This is an Instant Messaging client (Jabber client) implemented in C# that can be used as a proxy to the Agent Broker. This enables mSalsa agents to communicate with other agents and users.

*** Note: More information about this client is found in the documentation of mSalsa.*

SALSA

Class ProxyBroker

public abstract class **ProxyBroker**

Proxy to the Agent Broker. This should be a superclass of the Instant Messaging client through which the agent will communicate.

Methods

send

```
public abstract void send(java.lang.String xml)
```

Sends a message through the Agent Broker to another agent. This method should be overwritten by the Instant Messaging Client.

Parameters:

`xml` - Message to be sent

connect

`public abstract void connect()`

Connects to the Agent Broker. This method should be overwritten by the connect method of the Instant Messaging Client.

A.8 Parsing the SALSA XML messages

mSalsa, Utilidades (to be used for SALSA agents)

Class Parser

public class **Parser**

Provides the methods that facilitate parsing the received xml messages. The following is an example of how to parse an SALSA message.

```
Utilidades.Parser p = new Utilidades.Parser(secondaryContext);
String context = p.getTag("context");
String type=p.getAtt("context", "type");
```

Constructors

Parser

`public Parser(java.lang.String xml)`

Creates a Parser object that will be used to parse the xml message specified as an argument.

Parameters:

`xml` - message to be parsed.

Methods

getTag

`public java.lang.String getTag(java.lang.String tag)`

Obtains the value of the tag specified as an argument.

Parameters:

`tag` - name of the tag

Returns:

the value of the tag

---

getType

```
public java.lang.String getType(java.lang.String xml)
```

Obtains the type of the message which can be: "command", "request", "response", "notificationInfo", "dataFromSensor", "contextual message"

Parameters:

`xml` - the xml message.

---

getAtt

```
public java.lang.String getAtt(org.w3c.dom.Node node,
                               java.lang.String att)
```

Returns the value of an attribute of a tag.

Parameters:

`node` - the name of the tag.

`att` - the name of the attribute of the tag

Returns:

the value of the attribute

Appendix B

Forms and questionnaires

B.1 Questionnaire to evaluate Perceived Ease of Use of the API of SALSA

A) Mark with √ the response you consider more appropriate.

1. Learning the SALSA API would be easy for me.

Likely	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	Unlikely
	extremely	quite	slightly	neither	slightly	quite	extremely	

2. I would find it easy to implement intelligent systems with SALSA

Likely	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	Unlikely
	extremely	quite	slightly	neither	slightly	quite	extremely	

3. My interaction with SALSA would be clear and understandable

Likely	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	Unlikely
	extremely	quite	slightly	neither	slightly	quite	extremely	

4. I would find SALSA to be flexible to interact with

Likely	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	Unlikely
	extremely	quite	slightly	neither	slightly	quite	extremely	

5. It would be easy for me to become skillful at using the SALSA API

Likely	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	Unlikely
	extremely	quite	slightly	neither	slightly	quite	extremely	

6. I find SALSA easy to use

Likely	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	[ ]	Unlikely
	extremely	quite	slightly	neither	slightly	quite	extremely	

B) Answer the following questions:

7. What functionality of the agent do you think was easier to implement and why?

[ ] Reasoning component [ ] Communication of agents

[ ] Derive context [ ] Action component [ ] other______________

Comments:____

____

____

8. What functionality of the agent do you think more difficult to implement and why?

[ ] Reasoning component [ ] Communication of agents

[ ] Derive context [ ] Action component [ ] other_____________

Comments:____

____

____

B.2 Final Exam of the Object Oriented Analysis and Design Course

Final Exam (December 3 2004)

(2 hour exam, closed book.)

Name:____

1. Explain one advantage and one disadvantage of RMI with respect to CORBA. (**10 pts**.)

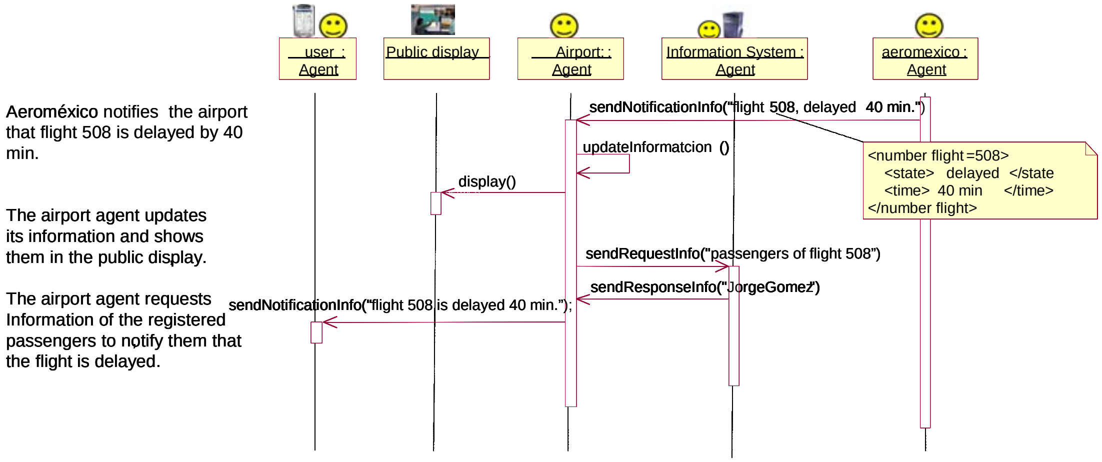
2. ¿Why is it useful to separate the interface from the implementation of a class or subsystem? (**10 pts**.)

3. Explain the concept of "design pattern" and provide an example of its application in the design of a system. (**10 pts**.)

4. ¿What is the problem of authors presenting the architecture of a system using a single diagram? (**10 pts**.)

5. Provide two benefits to a programmer offered by the use of 'Middleware' and one benefit for the final user of the system developed using the middleware. (**10 pts**.)

The Mexico City airport has an ubiquitous computing system that allows users to consult flight information in a public display or the user's handheld computer (PDA). The following sequence diagram illustrates the interaction of the components of the system, which have been implemented as SALSA agents.


user : Agent
Public display
Airport: : Agent
Information System : Agent
aeromexico : Agent
Aeroméxico notifies the airport that flight 508 is delayed by 40 min.
The airport agent updates its information and shows them in the public display.
The airport agent requests Information of the registered passengers to notify them that the flight is delayed.
sendNotificationInfo("flight 508, delayed 40 min.")
updateInformatcion ()
display()
<number flight=508>
<state> delayed </state
<time> 40 min </time>
</number flight>
sendRequestInfo("passengers of flight 508")
sendResponseInfo("JorgeGomez")
sendNotificationInfo("flight 508 is delayed 40 min.");

You have been hired to extend the functionality of the system with SALSA agents, so that the system can also recommend services available within the airport that could be useful to the passengers while they wait for their flight. The system should have the following functionality:

The user's PDA when notified that his flight has been delayed, request information of the available services and their location to the airport's Service Directory. The Service Directory provides this information and a map of their location. The system in the PDA personalizes the map according to the user's preferences and available time, highlighting services such as restaurants, book stores, internet café, etc. A scenario of use of the extended system will be as follows:

"When Mr. Jorge Gomez enters the airports waiting room, his PDA notices him that he has received a message indicating that his flight has been delayed by 40min. The system in the PDA requests to the airport's Service Directory those services that are available to determine those that could be useful to Mr. Gomez and that he could take advantage of while he waits for his flight. The Service Directory provides this information and a map with the location of the services. The PDA, knowing the time he has and his preferences, elects to highlight the location of the Vips restaurant and the 'Mexico" bookstore, since

Mr. Gomez needs to buy a book for her daughter. Mr. Gomez goes to the restaurant and while he waits for his food, accesses information of the bookstore, selecting in the map a link to the bookstore's web page to consult the availability of the book he wants."

You are asked to:

a. Create a diagram showing the components of the original system. (**10 pts.**)
b. Modify the previous sequence diagram, incorporating the components that implement the new functionality required by the system. (**20 pts.**)
c. Modify the components' diagram elaborated in question *a.* to incorporate the new components of the extended system. (**10 pts.**)
d. Describe the sequence diagram you have extended, explaining the behavior of the agents. That is, what functionality is implemented by each of the components of each agent (perception, reasoning, action) (**10 pts.**)