A METHODOLOGY AND SOFTWARE PLATFORM FOR BUILDING WEARABLE COMMUNITIES

by

GERD KORTUEM

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 2002

"A Methodology and Software Platform for Building Wearable Communities," a dissertation prepared by Gerd Kortuem in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

Dr. Zary Segall, Chair of the Examining Committee

12/3/02
Date

Committee in charge:
Dr. Zary Segall, Chair
Dr. Steve Fickas
Dr. Daniel Zappalla
Dr. John Orbell
Dr. Sharad Garg

Accepted by:

Dean of the Graduate School

An Abstract of the Dissertation of

Gerd Kortuem for the degree of Doctor of Philosophy

in the Department of Computer and Information Science to be taken December 2002

Title: A METHODOLOGY AND SOFTWARE PLATFORM FOR BUILDING WEARABLE COMMUNITIES

Approved: _____

Dr. Zary Segall

In recent years, two innovative computing and communication technologies have emerged: wearable computers and wireless personal area networks. Wearable computers advance an innovative form of personal computing based on continuously worn, intelligent assistants that augment memory, intellect, communication, and physical senses. Wireless personal area networks (WPAN) are a new class of wireless networks that provide seamless ad hoc communication over short-range radio links. The convergence of these technologies creates new opportunities for technological support of social interactions and face-to-face communities.

While past research has provided a partial understanding of the social potentials of wearable computers and wireless personal area networks, we know little about the

software engineering aspects of such systems. This dissertation aims to remedy this situation by exploring software infrastructure and architectural support for co-present communities. In particular, the goal of this dissertation is to develop a generic wearable software platform that (1) enables spontaneous interactions in face-to-face settings, (2) aids developers in the implementation of ad hoc collaborative applications and (3) supports building of co-present communities.

The contribution of this dissertation is a methodology and software platform for building *wearable communities*; that is, co-present communities that emerge when enough people use their wearable computers to form webs of personal relationships. Wearable communities are based upon embodied real-world human encounters augmented by wearable computers. The proposed methodology defines a conceptual framework for software support of wearable communities, and specifies a design language and development process. The software platform addresses the information needs of applications and provides developers with high-level programming abstractions. To address the utility and practicality of the methodology and software platform, the design and implementation of a number of wearable community applications are presented and experiences of using the methodology in software engineering education are reported.

CURRICULUM VITAE

NAME OF AUTHOR: Gerd Kortuem

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
University of Stuttgart, Germany
Johann Wolfgang Goethe-University, Frankfurt am Main, Germany

DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science, 2002, University of Oregon
Master of Science in Computer and Information Science, 1995, University of Oregon
Diploma in Computer Science, 1991, University of Stuttgart, Germany

AWARDS AND HONORS:

Fulbright Fellow, University of Oregon, 1992 - 1993

ACKNOWLEDGEMENTS

The research behind this dissertation could not have been accomplished without the personal and practical support of numerous people. My sincere gratitude goes foremost to my parents and all my friends for their love, support, and patience over the last few years.

I am appreciative and thankful for the support of my dissertation chair, Dr. Zary Segall. Without his encouragement, trust and support this dissertation would not have been written. I also would like to thank the members of my committee, Dr. Steven Fickas, Dr. Daniel Zappalla, Dr. John Orbell and Dr. Sharad Garg for their support.

Jay Schneider deserves special recognition for his creativity and imagination. He has been an untiring source of inspiration and ideas which I taped into many times.

I would also like to express my thanks and appreciation to the many students who assisted me in the development of the software for this dissertation. These are (in chronological order): Thaddeus G. Cowan Thompson, Jim Suruda, Dustin Preuitt, Jason Prideau, Christian Tan, Wonkey Shin, and Andrew Fortier. That so many of the ideas and concepts described in this dissertation have been realized is because of their enthusiasm and skills.

Kevin Matthews has shown me who far one can go with dedication and determination. I would like to thank him for the opportunities he offered me.

I would also like to express appreciation to my friend, Lance Barton, who has introduced me to a world beyond the close confines of this research. I am lucky to have met him.

Over the past seven years Cindy has been witness to the many ups and downs of my own dissertation process and no one could ask for a better friend and partner in life than she has been to me.

Thank you all.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

LIST OF PROGRAMS

Chapter I

INTRODUCTION

Wearable computing advances an innovative form of personal computing brought about by continuously worn, intelligent assistants that augment memory, intellect, communication, and physical senses (Mann 1997; Rhodes 1997; Starner 1999; Starner 2001). As wearable computing technology is taking on a larger role in our daily lives, one the most important questions is how this technology will influence people's social behavior and change the way wearable computer users interact, cooperate, organize and act collectively. Will it merely make people "smarter" by providing them with seamless, context-aware access to information or will it contribute to rich social interactions? Unfortunately, wearable computing research for the most part ignores the crucial interplay between wearable computing technology and people's social behaviors. Current systems and applications emphasize intellectual and sensory capabilities over interpersonal interactions and social competence. As we want to advance the wearable computing paradigm as an innovative form of personal computing, we need to carefully consider the social mechanisms, social potential and social constraints of this technology.

Important research questions in this respect are:

- What are the social potentials of wearable computers, if any?
- What are the requirements and characteristics of wearable systems to realize these potentials?
- How can we systematically and effectively build such systems?

- How can we evaluate the success of failure of these wearable systems?

This dissertation focuses on the second and third question, provides some insights into possible answers for the first question and ignores the last question.

The focus of this dissertation is the combination of wearable devices and wireless personal area networks. Wireless personal area networks, such as Bluetooth™, are a new class of wireless networks that enable seamless ad-hoc communication over short-range radio links (typically up to 10 m).

The convergence of wearable devices and wireless personal area networks creates opportunities for a new family of wearable computing applications aimed at fostering human connections, enabling cooperation and collaboration, and supporting natural social behaviors. Such applications hold the potential to modify established social relationships and create new ones. In particular, it provides new opportunities for community-building in the world of face-to-face encounters by bringing together strangers and by enhancing social interactions among acquaintances.

This dissertation introduces the notion of *wearable community*. By wearable community we mean a social network that emerges when enough people use wearable communication and computing technology throughout their daily lives. A wearable community is a community that is defined by the use of a particular technology, namely wearable computers, in the same way an online community is defined by the use of the Internet. Similar to online communities, wearable communities are multiparty conversation organized around affinities and shared interests, bringing together people who do not necessarily know each other personally. Unlike online communities, they are based upon embodied, co-located, real-world human encounters that are mediated and augmented by wearable computers. The notion of wearable communities builds on preceding research aimed at technology to support face-to-face communication. In particular, systems like ThinkingTags (Borovoy et al. 1996), MemeTags (Borovoy et al. 1997), i-Balls (Borovoy et al. 2001), Hummingbird (Holmquist et al. 1999) and Geney (Danesh et al. 2001) have shown how mobile and wearable devices can be used to

facilitate face-to-face communication among friends and strangers alike and support social practices of sharing, cooperation, learning and story-telling.

While we have at least partial answers to the question of the social potentials of wearable computers, we know almost nothing about the engineering aspects of systems supporting social face-to-face interactions. This dissertation aims to remedy this situation by exploring requirements and architectural support for *wearable community applications*.

Wearable community applications are a form of wearable communityware. Communityware (Ishida 1998a, Ishida 1998b) is software that fosters social exchanges and contributes to community building among a loose collection of individuals with shared interests but no common goal. Communityware stands in contrast to groupware, software that enables task-oriented cooperation and collaboration among members of a well-defined group who share a common goal or purpose.

I.1 Problem Description

While enabling technologies for the realization of wearable community applications have become widely available over the last years, namely wearable devices, short-range wireless networks, mechanisms for spontaneous networking, as well as context technologies, there is a lack of high-level development support in the form of design methods and software infrastructure. Wearable communities require loosely coupled, dynamic, decentralized systems composed of communicating wearable devices. In order to support the formation of wearable communities anywhere at anytime, such systems must be independent of external communication and computing infrastructures, relying solely on the capabilities of devices carried by individuals. Developing wearable community applications for such environments is a difficult task that touches upon a wide variety of issues. These range from human-factors issues related to collaboration paradigms based on opportunistic, proximity-based interactions

to technical issues of communication and context-awareness in dynamic ad hoc networks.

The fundamental problem in the development of wearable community applications is the semantic gap between the application domain and system layer. As of today, there is no direct support for the variety of features that wearable community applications require. Likewise, there are no programming and building abstractions for developers to leverage off when designing such applications. This results in a lack of generality, requiring each new application to be built from the ground up in a manner dictated by the underlying network technology and device platform.

This set of problems associated with the development of wearable community applications makes it clear that there is a need for the uniform support of the design, implementation and deployment of wearable community applications. Thus, the goal of this dissertation is to develop a conceptual framework and concrete tools to enable rapid development of wearable community applications.

The hypothesis of this dissertation is:

A software platform that provides support for key abstractions together with a development methodology that defines a systematic, repeatable process will enable developers to build wearable community applications within a short time frame and with reduced effort.

I.2 Contributions

This dissertation makes three main contributions:

1. The first contribution is the *WearCoM* wearable community methodology. Its purpose is to guide the design of wearable communities and wearable community applications. It consists of three components: (1) a conceptual model that defines terminology and an abstract architecture; (2) a design language that addresses the specification of important analysis and design

decisions and enables developers to specify key aspects of the application design; and (3) a development process that outlines a sequence of development steps that result in the creation of specific artifacts.

2. The second contribution is the *Proem* peer-to-peer platform. Proem is designed to tightly integrate with the WearCoM methodology and provides infrastructure and development support required for wearable community applications. The main focus of Proem is the information needs of applications and the provision of high-level programming abstractions. The three main platform components are:

   - The Peerlet application framework, a collection of libraries and APIs for the rapid development of wearable community applications.
   - The Proem runtime system, a software environment for hosting and executing applications built with the Peerlet framework.
   - The Proem protocols, a set of peer-to-peer protocols that define the way in which Proem peers communicate and cooperate over the network.

3. The final contribution of this research is the evaluation of the WearCoM methodology and Proem platform. To address the utility and practicality of the methodology and platform for the development of wearable community applications, we present five case studies and report our experiences of using the methodology in software engineering education.

I.3 Outline

This dissertation is structured as follows:

Chapter II explores social and technical aspects of wearable communities. We present a set of design principles for wearable communities that represent requirements for technological support of wearable communities and survey related systems to

support face-to-face communication. The chapter concludes by identifying the need of and the requirements for a wearable community methodology. The main outcome of this chapter is an understanding of why existing development support for wearable communities is not sufficient.

Chapter III presents background for this work. We summarize the key concepts of the following areas: wearable computing, ad hoc networking, virtual communities and communityware.

Chapter IV introduces the WearCoM methodology. We lay out the underlying conceptual framework and describe the design process and design language. Throughout the chapter we provide example to clarify key concepts. The chapter concludes with a discussion of tool support for the methodology. The outcome of this chapter is an understanding of the key requirements of a wearable community platform.

Chapter V and Chapter VI describe the *Proem* peer-to-peer platform. Chapter V starts by discussing basic assumptions and providing an architectural overview. It then goes on to discuss the set of protocols that lie at the heart of Proem: The Proem Transport Protocol which defines a common messaging layer for ad hoc networks and the Proem Presence Protocol which implements a mechanism for devices and users to announce their presence throughout a network of wearable computers. Chapter VI describes the Peerlet application framework for building presence-aware ad hoc collaborative applications.

Chapter VII presents five wearable community case studies. In each case we follow the whole development process as laid out by the methodology and present design and implementation details. Each of these case studies highlights one particular aspect of the Proem platform.

Chapter VIII discusses the main lessons learned from our experiences of using the Proem platform. We start with a brief overview of the highlights and shortcomings; next we contrast Proem with related software platforms. The chapter ends with a discussion of experiences of using Proem as educational tool in software engineering education.

Chapter IX summarizes the key results of this dissertation and presents directions for future research.

I.4 Topics Not Covered in This Dissertation

Like any other, this dissertation necessarily must limit its scope to a narrow set of questions. As a consequence, it leaves open many equally important related problems. Two subject areas which we do not investigate in this dissertation are privacy and security. Privacy is the right of individuals to control collection and use of personal information about themselves. Unlike security, which deals with safeguarding of information from unauthorized users, privacy is concerned with the amount of information known about an individual. Both privacy and security are essential for systems that broadcast personal information over wireless networks. Wearable communities depend on a careful tradeoff between revealing and hiding personal data: if users disclose too much personal data, their privacy might be violated; if they do not disclose any data, wearable communities might not emerge. Privacy and security in open decentralized systems are issues that by themselves warrant extensive exploration. As the focus of this dissertation is on the design process of wearable community applications, we will not elaborate on mechanism to ensure privacy and security. The last chapter, however, includes a brief discussion of trust as an alternative social mechanism for dealing with some of the aspects of privacy and security.

Chapter II

WEARABLE COMMUNITIES

This chapter lays the groundwork for this dissertation by presenting a conceptual framework of wearable computing support for social groups and social interactions. In order to highlight our emphasis on social aspects of wearable computers we first introduce the notion of *social wearable computing*. Social wearable computing is concerned with augmenting every-day social interactions and aims at wearable technology systems that enable collaboration and support natural social behaviors. We then explore *wearable communities* as a concrete example of the social wearable computing idea. A wearable community is a social network that emerges when enough people use their wearable computers to form webs of personal relationships. Similar to online communities, wearable communities are multiparty conversation organized around affinities and shared interests, bringing together people who do not necessarily know each other personally. Unlike online communities, they are based upon embodied real-world human encounters that are augmented by wearable computing technology. The chapter concludes with a discussion of technological challenges of *wearable community systems*, i.e. the hardware and software to facilitate and support wearable communities. The main outcome of this chapter is the realization of the need for software infrastructure and architectural support for wearable communities in the form of a *wearable community methodology* and *wearable community software platform.*

II.1 Social Wearable Computing

In recent years, wearable computing has emerged as a new discipline in the field of computer science. The applications of wearable computers are broad and encompass a wide spectrum ranging from personal to professional applications. They include augmented reality (Starner, Mann, Rhodes, Levine, Healey, Kirsch, Picard, and Pentland 1997; Rekimoto et al. 1998; Höllerer, Feiner, and Pavlik 1999; Höllerer, Feiner, Terauchi, Rashid, and Hallaway 1999; Bauer et al. 1998; Kortuem et al. 1998), maintenance (Smailagic et al. 1994; Siewiorek et al. 1998; Siewiorek et al. 1994; Kortuem, Bauer, Heiber and Segall 1999; Bauer et al. 1998; Bauer et al. 1999), personal information management (Rhodes 1997), shopping (Randell and Muller 2000), tourist information and guidance systems (Smailagic and Martin 1997; Höllerer, Feiner, Terauchi, Rashid, Hallaway 1999; Feiner et al. 1997), affective computing (Picard and Healey 1997; Healey and Picard 1998), personal imaging (Mann 1996; Mann 2001b), and video conferencing (Billinghurst et al. 1997; Billinghurst et al. 1998).

Although there is no commonly accepted definition of the term "wearable computing" we can identify two diverging strands of research: some (for example (Smailagic et al. 1994; Siewiorek et al. 1998)) see wearable computers as tools designed for particular industrial or military tasks while others (for example (Mann 1997; Starner 1999)) see wearable computing as a radical form of personal computing characterized by a near symbiotic relationship between man and computer. This symbiotic relationship is made possible by a modification of the physical and temporal boundaries of computer usage: (1) moving the computer directly onto the body and (2) making the device interactionally constant, that is, making the device's inputs and outputs always potentially active. One of the best illustrations of the latter idea, which we will refer to as *personal wearable computing,* is the Wearable Remembrance Agent (Rhodes 1997), a system that augments a user's memory by proactively searching for and presenting potentially valuable information based on a person's situational context: his location, people in the room, time of day, and subject of the current conversation. Processing is

performed on a shoulder-worn wearable computer and suggestions are presented on a head-mounted display.

The personal wearable computing idea was inspired in part by the works of two seminal researchers: In 1960, J.C.R. Licklider (1960) published a paper with the title "Man-computer symbiosis" in which he described his vision of real-time cooperation between man and computer. Two years later, Doug Engelbart (1962) wrote a report entitled "Augmenting Human Intellect: A Conceptual Framework" in which he laid down his vision of a synergistic man-machine system for improving the intellectual effectiveness of the individual human. In accordance with Licklider's and Engelbart's ideas, personal wearable computing is aimed at assisting the individual and the amplification of his intellectual and sensory capabilities. The most radical expression of this view was formulated by Steve Mann (Mann 2001a) who coined the term "humanistic intelligence". Humanistic intelligence is a computing style designed to assist human intelligence by augmenting and mediating the human senses. The proposed augmentation and mediation of human senses is enabled and fundamentally depends on body-worn computing devices that are always on and always active.

II.1.1 A Social Computing View of Wearable Computing

The personal wearable computing model has lead to a number of innovative applications: augmented memory systems (Rhodes 1997), augmented reality navigation systems (Feiner et al. 1997) and assistive wearable technology (Starner, Weaver, and Pentland 1997), to name just a few. Yet, this research has failed to realize the social potentials of wearable technology. Although the term "collective intelligence" has been around at least as long as wearable computers have become feasible, today's wearable users largely live a lonely and disconnected life. Other than email, interaction among fellow cyborgs is limited to the old fashioned way – through unmediated and un-augmented face-to-face conversation.

In this dissertation, we propose an alternative model of wearable computing, called *social wearable computing*. Social wearable computing defines a research framework that addresses the fact that wearable computing technology is taking on a larger role in human social lives. It is concerned with assisting and augmenting every-day social interactions and aims at wearable technology systems that enable collaboration and support natural social behaviors and compelling and effective social interactions. Social wearable computing is motivated and made possible by the recent emergence of *wireless personal area networks* (WPAN), a new class of wireless networks that enable seamless ad-hoc communication over short-range radio links (typically up to 10 m). Wireless personal area networks such as Bluetooth™ make it possible for wearable computers belonging to different individuals to communicate during face-to-face encounters, thus enabling computer-mediated interactions among co-located people.

In contrast to personal wearable computing, social wearable computing focuses on supporting social groups, e.g. families, groups of friends, class mates etc, rather than disconnected individuals. This includes support for group formation processes and activities within already well-established groups. Aspects of social live that fall within the reach of social wearable computing are social awareness, social identity, group belonging and social curiosity.

II.1.2 Augmenting Social Space

The key mechanism of social wearable computing is the augmentation of social interactions and social space. *Social space* is a concept introduced by Edward T. Hall as part of his theory of proxemics (Hall 1959; Hall 1962; Hall 1966). He established the idea that there are distinct levels of proximity in interpersonal communication. In western society, he distinguished four concentric spatial zones, each one a region around the body demarcated by invisible, though operative, boundaries determined by the characteristics of sense organs, the length of limbs and cultural conditioning, and the

particular relationship with the other (Table 1) (Hall 1966): *intimate space* – where touch happens – is the closest "bubble" of space surrounding a person; *personal space* is used for conversations and among friends and family members; *social space* is the space in which people feel comfortable conducting routine social interactions with acquaintances as well as strangers; and public spaces is the area of space beyond which people will perceive interactions as impersonal and relatively anonymous. The particular zone that we use depends on situational conditions such as our relationship with the others and the activity we are engaged in.

Table 1. Edward T. Hall's spatial zones

Distance	Appropriate Relationships and Activities	Sensory Qualities
Intimate distance (0 to 1.5 feet)	Intimate contacts (e.g. making love, comforting) and physical sports (e.g. wrestling)	Intense awareness of sensory inputs (e.g. small, radiant heat) from other person: touch overtakes vocalization as primary mode of communication.
Personal distance (1.5 to 4 feet)	Contacts between close friends, as well as everyday interactions with acquaintances.	Less awareness of sensory inputs than intimate distance; vision is normal and provides detailed feedback; verbal channels account for more communication than touch.
Social distance (4 to 12 feet)	Impersonal and businesslike contacts	Sensory inputs minimal; information provided by visual channels less detailed than in personal distance; normal voice level (audible at 20 feet) maintained; touch not possible.
Public distance (more than 12 feet)	Formal contacts between an individual (e.g. actor, politician) and the public	No sensory inputs; no detailed visual input; exaggerated nonverbal behaviors employed to supplement verbal communication, since subtle shades of meaning are lost at this distance.

Social space varies across individuals according to factors such as culture, age, and gender and is characterized by direct verbal communication, embodied co-presence, mutual awareness and shared artifacts.

Wireless personal area networks make it possible to augment social space. They generate a sphere-like digital space that envelops a wearable computer and its users. This digital space is filled with information broadcast by the wearable computer. The extension of the space depends on the transmission range of the wireless transceivers and can range from a few inches to several feet. The digital space is invisible to the human eye but can be sensed by wearable computers: as soon as the digital spaces of two wearable computers overlap, they become aware of each other.

The digital field created by wireless personal area networks can is the *alter ego* of human social space. Just as the concept of social space describes factors that influence inter-personal activities, the concept of digital space describes factors that influence interactions between wearable devices. Interactions in the social realm can initiate interactions in the digital realm, and vice versa. For example, during a conversation, two individuals may exchange electronic business card between their Personal Digital Assistants (PDAs). Here, social interaction leads to digital interaction. On the other hand, interactions between wearable devices can facilitate, promote or even augment social interactions between their users. For example, wearable computers connected by wireless personal area networks can inform their users about the presence of nearby persons, suggest a conversation topic for two strangers meeting for the first time or speculate about affinity relationships from repeated encounters (Terry et al. 2002). To refer to the combination of social space and corresponding digital space we introduce the term *augmented social space*. In an augmented social space, human interactions are augmented by digital interactions (Figure 1). In this context "augmented" refers to the assistive and complementary nature of the interactions and indicates a temporal, physical and semantic correlation.

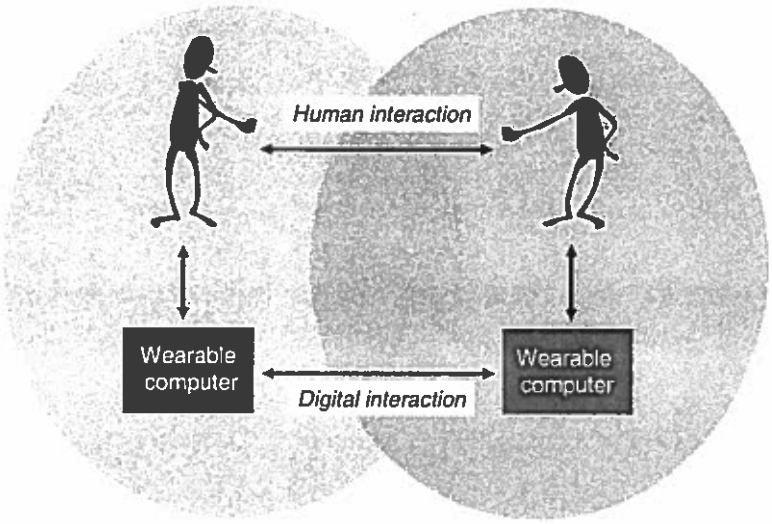


Figure 1. Augmenting Social Space

II.1.3 Wearable Computers

The personal wearable computing model is embodied by a very specific form of wearable computer. Wearable computers are typically composed of a CPU and storage unit attached to a belt or carried in a backpack, see-through head mounted display (HMD), wireless communications hardware and an input device such as touchpad or chording keyboard. Worn on the body, they provide constant access to computing and communications resources. Over the years, there have been many attempts at defining what distinguishes a wearable computer from a simple mobile or handheld device. Rhodes (Rhodes 1997, 3) lists five essential characteristics of wearable computers:

(i) "Portable while operational*: The most distinguishing feature of a wearable is that it can be used while walking or otherwise moving around. This distinguishes wearables from both desktop and laptop computers.*

(ii) Hands-free use*: Military and industrial applications for wearables especially emphasize their hands-free aspect, and concentrate on speech input and heads-up display or voice output. Other wearables might also use*

chording-keyboards, dials, and joysticks to minimize the tying up of a user's hands.

(iii) Sensors*: In addition to user-inputs, a wearable should have sensors for the physical environment. Such sensors might include wireless communications, GPS, cameras, or microphones.*

(iv) "Proactive"*: A wearable should be able to convey information to its user even when not actively being used. For example, if your computer wants to let you know you have new email and whom it's from, it should be able to communicate this information to you immediately.*

(v) Always on, always running*: By default a wearable is always on and working, sensing, and acting. This is opposed to the normal use of pen-based PDAs, which normally sit in one's pocket and are only woken up when a task needs to be done."*

According to Rhodes, the main distinguishing feature of wearable computers is that they are designed to minimize the distraction from the wearer's real-world tasks. A wearable computer is no longer the main center of user attention (as is the case with desktop and handheld computers) but it takes on an active support role. A similar definition was offered by Starner (Starner 1999) who lists the following attributes as necessary requirements for achieving human-computer symbiosis: a wearable computer persists and provides constant access, senses and models context, augments and mediates, and interacts seamlessly with the user by adapting its input and output modalities according to the context.

Social wearable computing defines new requirements for wearable computers. In order to support the social wearable computing model, we believe a wearable computer must satisfy the following operational characteristics:

1. *Constant*: A wearable computer should be always on and always running, it should not need to be turned on every time the user wants to use it (this does not exclude the possibility of low-power sleep modes).

2. *Presence-aware*: A wearable computer should be aware of the presence of nearby people (i.e. wearable computer users). The term "presence-aware" refers to the acquisition and use of knowledge about the availability and reachability of people.
3. *Communicative*: Wearable computers should be able to communicate with each other. Most importantly, wearable computers should be able to communicate with other nearby wearable computers through the use of short-range network technologies like Bluetooth or IEEE 802.15 Personal Area Networks.
4. *Proactive*: A wearable computer should be able to perform tasks autonomously and proactively without requiring explicit user intervention (interactivity may also be supported). Most importantly, a wearable computer should be able to interpret and react to changes in its knowledge about the presence of wearable computer users in the immediate environment.

This characterization of a wearable computer is consistent with previous definitions, yet adds further functional requirements: it defines the ability to communicate with nearby devices as important attribute and it replaces the generic term context-awareness with the more specific term presence-awareness.

The differences between the traditional personal model of wearable computing and our new social wearable computing model are summarized in Table 2. The models differ in what interface they focus on (human-computer vs. human-human), whether they focus on the individual or on social groups, the object of augmentation (intellectual capabilities vs. social interactions), the necessary device characteristics, and finally, the applications. In the following, we will discuss the new applications made possible by social wearable computing.

Table 2. Personal wearable computing vs. social wearable computing

	Personal Wearable Computing	Social Wearable Computing
Interface	Human – computer	Human – human
Focus	Individuals	Social groups
Augmentation	intellectual and sensor capabilities	Interpersonal interactions and social space
Device characteristics	Constant, hands-free, sensors, proactive, portable while operational (Rhodes 1997) Constancy, mediation, augmentation (Mann 2001a) Persistent and constant, senses and models context, augments and mediates, interacts seamlessly with the user (Starner 1999)	Constant, presence-aware, communicative, proactive
Applications	Augmented memory, augmented reality	Wearable communities

II.2 Wearable Communities

The social wearable computing model creates opportunities for a new family of wearable computing applications aimed at fostering human connections, enabling cooperation and collaboration, and supporting natural social behaviors. Social wearable

computing holds the potential to modify established social relationships, and create new ones. In particular, it provides new opportunities for community-building in the world of face-to-face encounters by bringing together strangers and by enhancing social interactions among acquaintances. We call the social networks that emerge when enough people use wearable computers throughout their daily lives *wearable communities.* A collection of wearable computer users becomes a wearable community when *enough people use their wearable computers to form webs of personal relationships.*

The term community has a long and rich tradition in sociology, anthropology and psychology and in recent times has become popular with the emergence of online or virtual communities (Rheingold 1993). It is thus not surprising that the term "community" has many different uses and connotations. In addition, many different types of communities have been identified over time (e.g., intentional communities, communities of interest, and communities of practice). In the most general terms, Mynatt et al. (1997, 16) define a community as "a social grouping which exhibit in varying degrees: shared spatial relations, social conventions, a sense of membership and boundaries, and an ongoing rhythm of social interaction." With a view towards online communities Howard Rheingold offered in his book, The Virtual Community (Rheingold 1993) the following definition: "Virtual communities are social aggregations that emerge from the Net when enough people carry on those public discussions long enough, with sufficient human feeling, to form webs of personal relationships in cyberspace." Finally, Jennifer Preece (Preece 2000) defines online communities as

- a collection of people
- with a shared purpose,
- guiding policies and
- a supporting computer system.

Within the community, people interact with each other and take roles; the shared purpose is a common interest or need that provides a reason for belonging to the community; policies are (implicit or explicit) protocols that guide people's interactions or folklore and rituals that bring a sense of history and accepted social norms (Preece 1999).

An important distinction exists between co-present (or face-to-face) communities (Ostrom 1990) and online communities. Most computer-mediated communities are online communities and are characterized by the fact that members are separated in time and space. On the other hand, most co-present communities are not computer-mediated and are characterized by direct interactions in face-to-face settings. A wearable community is a computer-mediated co-present community. It is defined by the use of a particular technology, namely wearable computers, in the same way an online community is defined by the use of the Internet. Similar to online communities, wearable communities are multiparty conversation organized around affinities and shared interests, bringing together people who do not necessarily know each other personally. Unlike online communities, they are based upon embodied real-world human encounters that are augmented by wearable computing technology. The notion of wearable communities acknowledges and is based on the unique value of random encounters and face-to-face interactions. We believe that fully embodied human moments are essential for community building. While online communities on the Internet have led to a separation of physical place and social space, wearable communities attempt to reunite the two.

II.2.1 Examples

In order to clarify the concept of wearable communities, we will now present some concrete examples. Over the last years, we have designed and implemented a variety of wearable community applications with the goal to explore social and technical aspects of wearable communities. Most of these applications are aimed at

University students. There are two important reasons why we choose to focus on students. First, students already form a loosely knit community and are likely to share certain interests. Second, the frequency of random encounters and casual social interactions is relatively high among students. Finally, most of our developers are students themselves and thus have a chance to use and evaluate their own technology. The three wearable community applications we will describe are PIRATÉ, WALID and mBazaar. We will discuss the implementation of these applications in detail in Chapter VII.

PIRATÉ

PIRATÉ (Kortuem et al. 2001) takes ideas from peer-to-peer file sharing applications like Napster and moves them to the wearable domain. PIRATÉ is a collaborative music guide that enables users to exchange MP3 play lists and music recommendations during brief random encounters. The goal of this application is (1) to enable user to discover new music they want to listen to based on the playing habits of the people they most often meet; (2) to provide awareness of the most favorite music titles within the community; and (3) to let users discover with whom they share a common taste in music. These goals are achieved by serendipitously exchanging and accumulating information throughout the day.

PIRATÉ implements the following scenario:

"Kim is a 20 year old architecture student and an ardent music lover. She has an extensive CD collection and has digitized all of them as MP3s. During most of her waking hours she can be seen sporting an MP3 player and headsets, especially when she is working in her studio at school.

As soon as the first generation of Bluetooth-enabled wearable MP3 players appeared on the market she rushed out to buy one despite its steep price: she knew that most of her friends at the University would do the same. Her new MP3 player has the

ability to detect other MP3 players in the vicinity and to trade play lists with them. Kim has set up her player to recognize the MP3 players of all of her music-loving friends and to automatically trade information about the songs she listens to. So while Kim and her friends are having lunch or are working in their studio, their MP3 players exchange play list and generate personalized music recommendations that take into account the listening habit of the entire social group. Furthermore, she has enabled the privacy mode of her MP3 player. As a result, her player emits an audible signal every time it is trading play lists with another device.

After each day at school, Kim downloads the information she collected from her friends' players to her laptop computer that is connected to the Internet. She then downloads freely available samples of the songs her friends listen to and copies some of them to her MP3 player. On a good day she identifies two or three new CDs she plans to buy as soon as she has enough money."

This scenario stresses two important points: first, interactions within wearable communities may be fully automated, not requiring direct interactions between users. The general willingness to share information and the opportunities presented by physical proximity suffice for successful collaboration. Second, existing personal relationships (such as between friends or colleagues) can play an important part in wearable communities.

WALID

The second wearable community application is WALID (Kortuem, Schneider, Suruda, Fickas, and Segall 1999). WALID implements a digitized version of the timeworn tradition of borrowing butter from your neighbor. You do a favor for others because you know that one day they will do it for you. With WALID two individuals use their wearable devices to semi-automatically negotiate about the exchange of real-

world tasks: dropping of someone's dry cleaning, buying a book of stamps at the post office, or returning a book to the local library.

WALID employs personal agent software to find community members close by and to negotiate the exchange of favors. The agents maintain a user's task list, becoming fully aware of the locations and activities involved. When an encounter occurs, the agents produce a negotiation. If both users approve, a deal is struck. The role of the agent in a negotiation is to evaluate the value of favors and to keep scores. Having to run across town just to drop off someone's mail, compares unfavorably with buying milk for someone if the grocery store is just a block away. WALID agents employ ideas from game theory to ensure that results of negotiations are mutually beneficial. They cooperate only if there is the opportunity to enhance the users' goals.

WALID illustrates the importance of trust in wearable communities. It is unlikely that people are willing to trade tasks unless they trust each other to a certain extent. Thus, WALID works best for close-knit groups with established trust relationships (for example groups of friends, classmates or colleagues).

mBazaar

mBazaar is a wearable community application developed to support students in buying, selling and swapping of personal items like CDs, books, bikes, furniture, and electronics. mBazaar employs personal agents that 'advertises' items a user wants to buy or sell to nearby wearable users. The use of a short-range wireless network guarantees that only people in the immediate vicinity of the advertising user can receive these 'virtual classifieds'. At the time of an encounter between two or more individuals, for example when students meet after class at a local coffee shop, the personal agents identify matches between advertised items and exchange the users' contact information. Depending on the privacy preference of each user, contact information might be a cell phone number or pictures of the users. The picture enables users to identify each other in a crowed and to verbally negotiate a possible transaction right on the spot.

The three wearable community applications are summarized in Table 3.

Many other wearable communities can be envisioned: *computing communities* in which computational resources like network bandwidth and computing cycles are shared; *helper communities* in which members pledge to assist each other when in distress; *bargain-hunter communities* in which members collectively search out sale items in local stores; *marketplace communities* in which goods are exchanged without money; *job market communities* in which free applications offer their services to passerby's; *knowledge communities* in which members collectively accumulate information to create shared understanding; and finally *political communities* in which members create new forms of spontaneous democracy and local activism.

Table 3. Wearable Communities

	PIRATÉ	WALID	mBazaar
User population	Music lovers, groups of friends, classmates, students, colleagues who frequently listen to music.	Close-knit groups with established trust relationships (for example groups of friends, classmates or colleagues)	Friends, students, colleagues
Purpose	(1) To generate personalized music recommendations based on listening behavior of community members (2) To discover people with similar music taste (3) to discover most popular music titles	(1) To strengthen the social ties of a group by identifying opportunities for mutual assistance (2) to enable users to do daily tasks more efficiently	(1) to match buyers and sellers while they are close to each other and are able to talk to each
Exchanged data	Play lists	Task descriptions	Classifieds

II.2.2 From Online Communities to Wearable Communities

The idea of wearable communities has precursors in other forms of computer-mediated communities. In the following, we will highlight the differences between online, mobile and wearable communities.

Online Communities

The invention of the Internet and the beginning proliferation of personal computers in the early 90ths created a new digital media that almost instantly was adopted by its users as a tool for social interactions. Today, people use the Internet to stay in touch with friends and family by sending greeting cards, sharing vacation photos, and chatting about daily life. One of the most powerful features of the Internet is its ability to enable total strangers to interact on a very personal and sometimes even intimate level. People who have never met in their entire life (and are unlikely to ever do so) use the Internet to discuss personal matters related to health, raising kids, romance and many other topics of shared interest. In effect, the Internet provides the connectivity for the "Global Information Community" envisioned in 1968 by J.C.R. Licklider (Licklider 1968, 2) as being "... not of common location, but of common interest." The fact that the Internet is a place that enables a new type of social network, one mediated by computer terminals and networks, was vividly described by Howard Rheingold who coined the term "Virtual Community" (Rheingold 1993). He defined virtual communities as "social aggregations that emerge from the Internet when enough people carry on public discussions long enough and with sufficient human feeling to form webs of personal relationships."

Examples of successful online or virtual communities include Slashdot, Sourceforge, and The Well.

Peer-to-Peer Communities

Recently, a new form of online communities has emerged, namely peer-to-peer file sharing communities. Applications like Napster, Gnutella (CLIP2 2002; Kan 2001), Freenet (Clarke et al. 2001) and KaZaA (Sharman Networks 2002) have created communities of users who anonymously trade MP3 files over the Internet. In contrast to traditional web-based online communities which create a centralized shared space for interaction, peer-to-peer file sharing occurs in a decentralized fashion via direct pair-wise interaction between users (Napster uses a centralized directory of available music titles, but file exchanges still occur in a direct peer-to-peer fashion). From a social point of view, peer-to-peer communities can be seen as a special form of online communities with the following unique characteristics: (1) they are very specific in their purpose (trading MP3 files is the only supported activity), (2) they are anonymous (in contrast to most web-based communities, users of peer-to-peer file sharing communities have no (semi-)persistent identities such as user names or handles; exchanges occur between computers with IP addresses but not between users), and (3) there are few guiding policies to resolve conflicts of interests (free-riders, people who download MP3s from others but do not share their own collection, are a constant problem in file sharing communities).

Reuniting Virtual Space and Physical Space

With the advent of online and peer-to-peer file sharing communities, social space and physical place have become separated to an unprecedented degree. Communities have emerged without relation to physical place, enabling people to connect across the globe at any time of the day or night without the need to leave the house or office. In some cases, this convenience can lead to a reduction in authentic psychological encounters or "human moments": when a person finds it easier to communicate anonymously with a stranger in a virtual world rather than to engage the neighbor next door, the consequences may include severe psychological problems

including isolation and depression (Kraut et al. 1998). The lack of fully embodied human moments in such cases may compensate or even reverse the otherwise positive character of online communities.

In face-to-face meetings, vocal inflection and gestures carry significant social cues which are necessarily absent from web-based tools. Thus, numerous attempts have been made to enrich the communication in online communities and to reproduce the physical world's rich set of social cues in the online world. Still, the vast majority of online communities on the Internet today are text-based.

An interesting attempt to reunite virtual space and physical space was made with the recent launch of the MEETUP web site (Meetup, LCC. 2002). MEETUP provide a service that allows members to set up or find face-to-face meetings of like-minded users in the same area. By telling the site where they are and what they are interested in (quitting smoking, reading books, having pets), users can organize and find meetings of other people in the same area and with the same interest. An important aspect of MEETUP is that the early adopters of this service are people who are already connected to a virtual community. Local meetups have already been scheduled for people from the Slashdot, LiveJournal, and Plastic communities. This indicates how interconnected virtual community members already are, and how much demand there is to meet face-to-face.

Mobile Communities

With the advent of mobile technologies like cell phones and Internet-enabled PDAs, online communities are slowly extending their reach to mobile users. Already, chatting on mobile phones has become an important part of the lifestyle of certain segments of the population. A recent study (Fox 2002) by the Social Issue Research Centre in Oxford found that gossiping on mobile phones has become a vital "social lifeline" for building relationships, solving conflicts, teaching social skills and making friends. A similar conclusion can be drawn from observations of teenagers in some

Asian and European countries, where exchanging SMS text messages via mobile phones has become a major part of adolescent rituals. Simultaneously, some online communities have sprouted mobile extensions by giving users access to community tools from their mobile devices, thereby allowing members to stay in touch with their community at anytime from any place.

Despite their widespread and successful adoption, mobile phones and Internet-enabled PDAs have serious flaws as tools for community building. Mobile phones enable communication over long distances; they do not support many-to-many communication and consequently do not facilitate random encounters (both virtual and real) and anonymous relationships. Using cell phones it is difficult to create new relationships, as they require the caller to have a fairly good idea who the callee is or a good reason (and thus a relationship) to place a call in the first place. Moreover, the use of mobile technology has the potential of decreasing real-world random encounters and face-to-face interactions: by attending to people who are located elsewhere they may reduce the amount of time and attention paid to strangers nearby and activities in public places. All this makes current mobile technology good for maintaining existing community relationships, yet ill suited for creating new ones.

With the mounting use of mobile communication and computing devices, various successful *mobile communities* have emerged over time. One example is Geocaching (Groundspeak Inc. 2002), a high-tech "treasure hunt" for users of the Global Positioning System (GPS). The game involves a GPS user hiding "treasures" -- the cache and its contents (usually Tupperware with goodies inside) -- in public places and publishing the exact coordinates on the Geocaching web site so other users can go on a "treasure hunt" to find it. Users equipped with GPS receivers seek out nearby caches and return back to the site to talk about it.

Differences between Virtual, Mobile and Wearable Communities

Wearable communities are different from online and mobile communities. Similar to online and mobile communities, they are webs of relationships that grow from computer-augmented social interactions. Yet, online and mobile communities depend on indirect and remote interactions between users. In contrast, wearable communities are based on direct face-to-face encounters. Thus, interactions in wearable communities are *situated* and involve a rich *social context*.

The three wearable community applications described above highlight the differences between traditional computer-mediated communities and wearable communities:

Social context: In a wearable scenario involving WPANs exchanges are only possible over short distances, that is, when people come face-to-face or are at least within close physical proximity. Consequently, communication partners will be aware of whom they are interacting with and be able to observe important social cues including sex, clothing and gestures. In addition, they might even be able to talk to each other. The addition of social context shapes people's willingness to engage with strangers and the particular manner in which they interact. For example, politeness and trust are two aspects of human interactions that strongly differ when people interact face-to-face vs. terminal-to-terminal across the Internet.

Usage context: The context in which wearable devices are used is different in two important respects from using a stationary computer at home or in the office. When wearable devices are involved, user attention is a scarce resource. Instead of sitting in front of a computer where a user can pay full attention to the computer and its operation, wearable computers are used in situations where the user's attention is occupied by demanding real-world task like driving, operating a machine, or simply conversing with other people.

Furthermore, with wearable computers time becomes a critical resource as well. When surfing the Internet at home or at the office people are less likely to care if an

operation takes just a few seconds or several minutes; they can always shift their attention to another task. This is no longer true if exchanges occur between two wearable devices. Since a connection can only be active while users are close to each other, a lengthy interaction among two wearable computers is less likely to succeed than a brief one.

Technical context: In comparison to wired computers networks, WPANs are limited in terms of bandwidth and reliability. Using PIRATÉ as an example, this suggests that rather than exchanging large-sized MP3 files, systems should be designed to exchange metainformation, that is, URLs pointing to an MP3 file an a server or recommendations about a song.

These observations make clear that wearable communities are significantly different from traditional computer-mediated communities. Thus it becomes necessary to rethink the whole system design and modify it in accordance with the social, usage, and technical context.

II.2.3 Factors that Bind Wearable Communities Together

When does a collection of individuals using wearable computers become a wearable community? A group of people comes together because each individual recognizes there is something valuable that they can gain by banding together. This is the theory of *collective goods* as formulated for example by Olson (Olson 1965) and it applies to all types of communities, whether augmented by technology or not. Thus looking for a group's collective goods is a way of looking for the elements that bind isolated individuals into a community. What people find in a community varies and may include knowledge, ideas, friendship, companionship, fun, excitement, money and many other qualities that people value.

Yet, a community is not a perfectly organized group where everyone who participates gains equally. One of the reasons is that peoples' individual actions typically serve their own self-interests. Although altruistic behavior is possible and

common, it is not the norm. In particular, when exchanges and transactions are involved, conflicts can arise. How individuals deal with these obstacles and how they adjust their behavior to overcome them makes the difference between a successful and a non-successful community. In the following, we list some of the factors that are relevant for wearable communities:

- *Identity:* All of our interactions, even those with strangers, are shaped by our sense of with whom we are interacting. In interactions involving social space, there are a wealth of cues of varying reliability to indicate our identity and our intentions. Our clothes, voices, bodies, and gestures signal messages about status, power, and group membership. We rely on our ability to recognize fellow group members in order to know who we can turn to and what we can expect. Our ability to identify others also allows us to hold individuals accountable for their actions. Wearable technology enables people to define and broadcast digital identities that may or may not be their true identities. In contrast to online communities, these digital identities augment rather than replace a person's true identity as both are visible at the same time. Because observers can match the digital identity to a person's appearance and behavior, it is difficult in wearable communities to switch identities at will.
- *Privacy:* Privacy and identity are two sides of the same coin. Privacy is the right of individuals to control collection and use of personal information about themselves. Unlike security, which deals with safeguarding of information from unauthorized users, privacy is concerned with the amount of information known about an individual. Privacy can be defined as an individual's ability of individuals to determine for themselves when, how and to what extent information about them is communicated to others. Not all wearable communities require members to disclose information about themselves. Lurking and voyeurism are two behaviors that, within limits, are sanctioned by social norms.

- *Situational Context*: Interactions in wearable communities take place within a rich social and physical context and are thus subject to usual norms. Because interactions occur in the real world, the location and the presence of other people matter just as much as in non-augmented communities. The differences between public and private space, crowded and empty rooms, meeting and party, or intimate conversation and public lecture remain important just as in unmediated situations (but unlike in online communities).
- *Historical context*: A wearable community is bounded in time and space. It has a beginning and an end and is located within a limited physical space. Although these boundaries might not be apparent to its members or outsiders, they nevertheless exist. With an extension in time and space, each community has a history and interactions that take place add to this history. In addition, each community member has a very personal interaction history. Having a sense of history and being able to learn from history is an important aspect of a wearable community.
- *Risks*: As wearable communities are based on interactions taking place in the real world, risks are real as well. In addition to risks of virtual communities like being verbally assaulted or loosing money, wearable communities also include risks to the person's life and wellbeing. Fear of being assaulted and injured may make people more careful than in virtual communities.
- *Trust*: Trust is an aspect of human interactions that strongly differs whether people interact face-to-face or terminal-to-terminal across the Internet (Misztal 1999). Being able to look someone in the face, and observe gestures and mimic can tell you a lot about another person. The reason why people place so much importance in a handshake when completing a transaction - such as selling a car - is that it is a sign of commitment and mutual trust. There is no equivalent of a handshake in online communities, but in wearable communities a (possibly digitally augmented) handshake is possible.

II.3 Building Wearable Communities

How can we build successful wearable communities? The success criteria for wearable communities are not different from those of real world or virtual communities: a wearable community is successful if it creates a rich social environment in which the augmentation of random encounters and social space leads to psychologically satisfying "human moments" and lasting personal relationships. What turns a loose collection of individuals into a community, is persistent and ongoing conversations. Thus a collection of wearable computer users becomes a wearable community when *enough people use their wearable computers to form webs of personal relationships.* In other words, we want wearable communities to foster social interaction and communal spirit. Yet this observation doesn't make it any clearer how we can achieve this goal. This is because wearable communities, like all communities, are the result of self-organizing behavior of a group of people. They emerge; they cannot be constructed from the outside by following a simple recipe. Referring to online communities, Jennifer Preece expressed this sentiment as follows:

"*Software developers design software, thinking that they are designing communities. Meanwhile, keen-eyed, reflective sociologists describe the emergence of communities. But communities are neither designed nor do they just emerge. How software is designed affects community development. The way people interact in a community strongly contributes to its long-term evolution. People's behavior cannot be controlled but it can be influenced*". (Preece 1999, 24)

Computer-mediated communities, in a very real sense, are self-organizing, emerging in response to technology and the needs of its users.

II.3.1 An Exploratory Design Approach based on Rapid Prototyping

An important characteristic of wearable communities is that social concerns become difficult to separate from technical practices: social and technical issues interact and co-evolve in such intimate ways that they often merge. Thus in order to design a wearable community, we need to simultaneously design technology and social interactions.

A successful community design often hinges on what Wenger (Wenger 1998) referred to as a "minimalist design". The idea behind a minimalist design is to create a provisional design and to facilitate the growing of community over time. While doing so, designers must try to identify the attributes that make communities successful and design technology to embody these attributes.

In building wearable communities, we are faced with a chicken-and-egg problem which we refer to as the *fundamental problem of wearable communities*:

Wearable communities require hardware and software to support them, but without experiences with actual wearable communities we don't really know what the success factors for wearable communities are and how to design systems to embody them.

The cyclical interdependence between technological and social factors of wearable communities is visualized in Figure 2: technology to support wearable communities exhibits certain properties which define the kind of wearable communities that can emerge. The success or failure of communities hinges on its ability to promote lasting personal relationships. The attributes that make communities successful must be embodied by the technology.

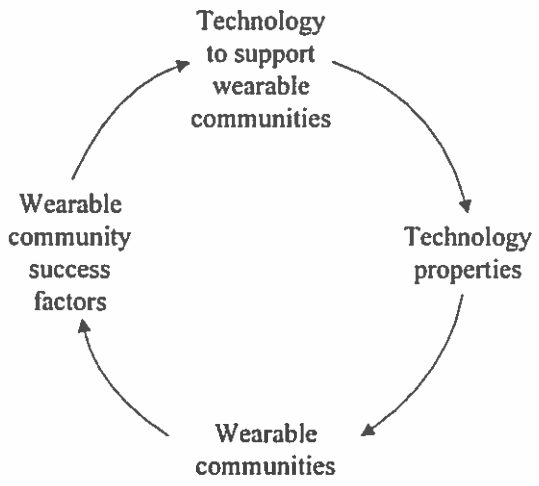


Figure 2. Wearable community design problem

How can we solve this problem?

One solution is to use an exploratory design approach based on rapid prototyping. Rapid prototyping is a circular design process based on an initial technology prototype followed by repeated evaluation and refinement. The initial prototype represents the designers' educated best-guess of what the final systems may look like. Its design may be informed by prior experiences with similar systems or theoretical insight.

A rapid prototyping approach can be divided into four distinct activities (Figure 3):

- Planning: this involves understanding the users (including goals, interests, needs, etc.), understanding the available technology, the definition of the overall purpose of a system, and the design of a prototype.
- Implementation. during implementation, a prototype is built
- Measuring: this involves observing the use of the system in a real-world setting with real users. The goal is to identify shortcoming of the prototype.
- Learning: finally, results from the measurements must be analyzed with the goal to determine in which way the prototype should be modified.

There can be many reasons that a prototype does not achieve its goals. Sometimes these will be implementation issues, but more often the problem will be an incorrect or insufficient understanding of the users. Thus it is important to go all the way back to planning and to take a fresh look at a system from a user's perspective. To succeed, prototyping must be iterative and must support rapid creation and modification of prototypes.

Rapid prototyping has successfully been used at Carnegie Mellon University for the development of successive generations of the VuMan wearable computer (Smailagic et al. 1998). Using this approach, interdisciplinary groups of students have again and again succeeded to design and build the hardware and software components of a wearable computer within just one semester.

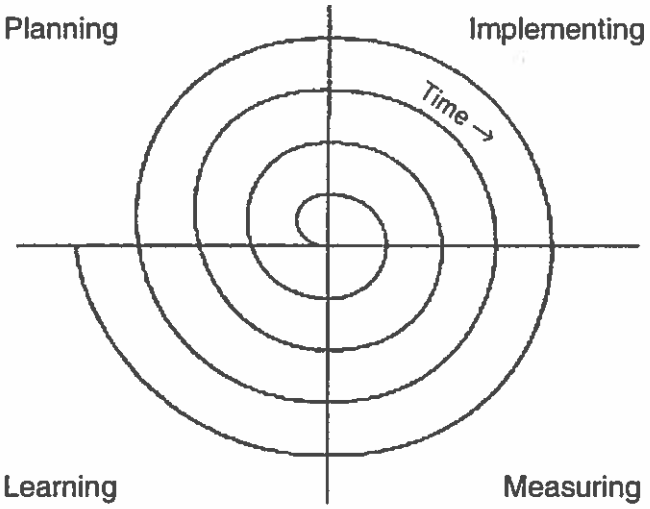


Figure 3. The iterative prototyping cycle

II.3.2 Wearable Community Systems

Building and maintaining computer-mediated communities involves the design and manipulation of technologies in ways that foster human connection. Technologies to support online communities include hardware (personal computers, networks, and hosts) and software (web servers, email list servers, message boards, chat and instant messaging software, online calendars, web browsers, newsreaders, etc.).

With regard to wearable communities the following questions arise:

- What are the characteristics and components of a *wearable community system*?
- What are the challenges in developing such systems?
- How can we systematically and rapidly build such systems?

In the remainder of this chapter, we will explore various aspects of wearable community systems and discuss the need for a software infrastructure and architectural support for wearable communities.

A wearable community system is a collection of wearable devices, distributed software infrastructure and application software that enables people to interact with fellow community members. By "system" we refer to software and hardware that sets the context for interaction. The term "wearable" indicates the wearable nature of devices and implies a usage model that is always on, communicate, proactive and presence-aware.

Within a wearable community system, communication is established by a wireless personal-area network that enables seamless connectivity among co-located devices. Thus, a wearable community system is highly dynamic, loosely connected, potentially large-scale distributed system made up of mobile host (Figure 4). In order to support the formation of wearable communities anywhere at anytime, such a wearable community system must be largely independent of external communication and computing infrastructures, relying solely on the capabilities of devices carried by individuals. Besides the obvious hardware components of the individual computers

(cpu, storage, input/output, etc.) it consists of system and application-level software that performs functions essential to wearable communities: discovery of nearby devices and users, initiating interactions across wireless links, managing user identities, handling user input and output and much more.

In contrast to online communities, for which a mature and proven technological foundation exists, wearable communities require advanced technologies that are subject to intensive research: this includes wearable devices, wireless ad hoc networks, mechanisms for spontaneous networking, as well as context technologies.

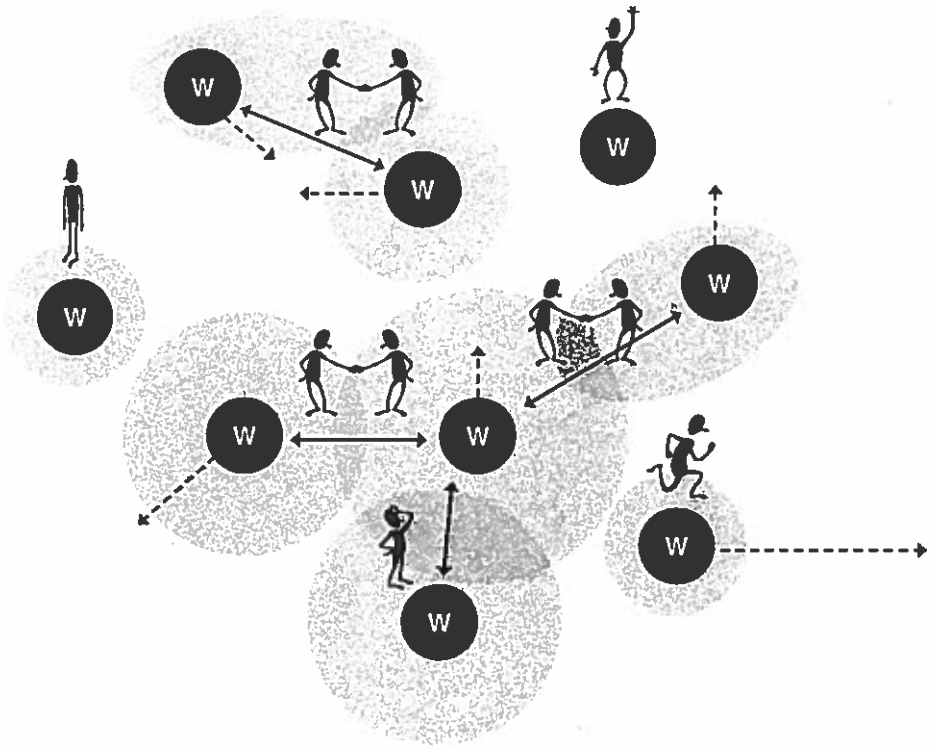


Figure 4. Wearable Community System

The key technical challenges of wearable community systems are the result of the decentralized system architecture and the independence and autonomy of individuals. In the following, we will list the primary challenges of wearable community systems. These are:

1. Creating presence-awareness in dynamic decentralized systems
2. Enabling human-centered communication
3. Enabling trust, cooperation and privacy in an open decentralized environment
4. Striking the balance between proactivity and user control

Creating Presence-Awareness in Dynamic Decentralized Systems

A Wearable community system is a highly dynamic, decentralized and self-organizing network of autonomous, wearable devices. Connectivity is determined by the distance between devices; as devices change their physical location they establish pair-wise communication links based on proximity. Thus, a wearable community system resembles a wireless peer-to-peer system in which human presence is the main resource. Important questions regarding presence-awareness include:

- What kind of personal information will users want to disclose to others?
- How can we support multiple identities for the same person to be used in different contexts?
- How can users determine who is around at a given moment and find out how they can interact with other people?
- How can we distribute presence information in a dynamic decentralized peer-to-peer network efficiently?

As individuals wander around and encounter each other, their devices must be able to detect the presence of nearby devices and exchange metainformation. Algorithms for presence-awareness must satisfy three conflicting requirements:

Expressiveness: They must allow users to create rich and possibly multiple digital identities for themselves.

Efficiency: To prevent overloading the algorithms must be efficient and/or adaptable.

Timeliness: To facilitate interactions among people in movement algorithms must react fast. For example, if two individuals, each of them carrying a wearable device, walk towards each other with a constant speed of 5 kilometers/hour and network transmission radius is 10 meters, devices will be able to communicate for 15 seconds or less.

Enabling Human-Centered Communication

Traditionally, addressing and naming schemes in distributed systems are designed to enable communication among devices. However, in a wearable community system the ultimate goal is communication among humans. Individuals may own and operate more than one device at different times. In addition, individuals might want to use alternative identities in order to protect their privacy. Thus, we must find answers to the following questions:

- How can we define device-independent identities for users?
- How can we move identities from device to device?
- How can we route a message destined for a particular user to the device he or she is currently using?
- How can we enable multi-hop communication among individuals in a dynamic crowed?
- How can we prevent unauthorized users from intercepting messages not destined for them?

Enabling Trust, Cooperation and Privacy in an Open Decentralized Environment

A wearable community system is an open environment without a well-defined population of individuals and devices. Individuals may join or leave a community at will. Therefore, encounters often occur between strangers who have never met before and might not do so again in the future. There is no *a priori* agreement among members of a community to cooperate. Instead, cooperation is the result of combined actions taken by independent users in their own interest. Sharing of resources and access must be granted between hosts who are unknown to each other. Developers cannot expect that all hosts and users are benevolent; hosts may attempt to gain access to private data, willingly corrupt data and messages, introduce wrong information or do harm in some other way. One of the main privacy concerns is protecting a user's anonymity. Monitoring network traffic or gaining access to confidential personal data can compromise a user's anonymity. Not only must a system prevent spying and monitoring, but users must also given control what information is disclosed, to whom, and when.

In such an environment, we are faced with three related challenges:

- How can we enable users (i.e. devices) to spontaneously and effectively cooperate if users (i.e. devices) are autonomous and non-trusted?
- How can we represent, codify and notions of trust and reputation?
- How can we enable individuals who have met before and have established relationships to form secure trusted groups?
- How can we protect a user's privacy and protect him/her from malicious community members if we can't trust anyone?

Striking the Balance between Proactivity and User Control

In a wearable community, digital interactions between devices are spontaneous and numerous. In such a scenario, it is no longer feasible to expect from a user to direct

an application and the many ongoing interactions in real-time. Instead, in order to perform acts of ad hoc cooperation, wearable devices must be able to reach out and negotiate with each other autonomously and proactively, i.e. they must be able to anticipate the requirements of the users and act accordingly. This requires a move from an interactive model of human-computer interaction to a new model based on supervision. Such supervisory interfaces will fundamentally rely on software agents that act in the interest and on behalf of the user.

Some of the challenges of supervisory interfaces are:

- How can we maintain human control over agent-based applications?
- How can we enable users to define and modify their agents' behavior?
- How should interfaces be designed that enable humans to supervise large numbers of concurrent real-time actions?

In sum, developing wearable community systems is a difficult task that requires highly specialized knowledge in a variety of fields ranging from human-factors issues related to collaboration paradigms based on opportunistic, proximity-based interactions to technical issues related to ad hoc networking and context-awareness.

II.3.3 Towards a Wearable Community Methodology

In recent years, the enabling technologies for the realization of wearable community systems have become available, i.e. wearable devices, wireless networks, mechanisms for spontaneous networking, as well as context technologies. As a consequence, individual solutions to the above challenges exist and it is possible to build wearable community systems today (see Chapter III). Yet, it is not possible to

build them rapidly and effectively. In particular, there is no support for the exploratory design approach based on rapid prototyping advocated above.

This dissertation aims to remedy this situation by exploring software infrastructure and architectural support for wearable communities.

As first step towards a solution, we introduce the distinction between *wearable community infrastructure* and *wearable community applications.*

- Wearable community infrastructure comprises system-level software components that are useful and necessary for the support of a large variety of wearable communities. This infrastructure is generic in a sense that it embodies fundamental aspects of the wearable community domain. Among other thing, wearable community infrastructure should include mechanism for distributed communication, discovery and presence-awareness, definition of user identities, situational and historical context and trust (see Chapter II.2.3).
- Wearable community applications are community-specific software applications that are built on top of and make use of functionality provided by wearable community infrastructure. They are installed on and run by users on their wearable computers.

As of today, there is no comprehensive wearable community infrastructure and there is little or no direct support for the variety of features that wearable community applications require. Likewise, there are no programming and building abstractions for developers to leverage off when designing wearable community applications. This results in a lack of generality, requiring each new application to be built from the ground up in a manner dictated by the underlying network technology and device platform. This situation is detrimental to an exploratory design approach and rapid prototyping.

To deal with the intervening technological and social issues of wearable communities we propose a *wearable community methodology* and associated *wearable community platform.* The methodology should incorporate the following parts:

1. A well-defined *conceptual framework* is required as reliable foundation. It should express the basic concepts and an abstract model of a wearable community system.
2. *Description techniques* are required that allow designers to capture essential design elements. They are necessary for communication between the designers and between designers and developers. Examples for description techniques are graphical notations like class diagrams and state transition graphs from modeling languages like UML.
3. Development should be organized according to a *process model.* Such a process model supports system development by clearly defining individual development tasks, roles and results as well as the relationships between them.

The description techniques and the process model should be directly supported by the associated wearable community platform. The platform realizes a wearable community infrastructure and should provide support for rapid development of wearable community applications. In particular, it must (1) enable spontaneous interactions in face-to-face settings, (2) aid developers in the implementation of ad hoc collaborative applications and (3) support formation of wearable communities.

If successful, the methodology and platform will function as catalyst for wearable community applications and thus wearable communities.

II.4 Summary

People are social animals. We love to interact with other people and to a large degree depend on social interactions for our psychological well-being. Social

interactions do not have to involve direct person-to-person conversations. Merely being at the same place at the same time can be an important indicator for similarities in taste, interests, preferences, and believes: concerts, street demonstrations, trade shows and social venues like bars are proof of that. Social behavior in public places like coffee shops and sidewalks can take the form of sending non-verbal signals and gestures and making others aware of one's preferences and personality, as well as observing and interpreting signals from others. To be 'seen' and asserting one's social role and group identity is an important aspect of any community, whether virtual or real. The notion of wearable communities is based on the idea that augmented random encounters and augmented social interactions can nurture social networks. Wearable computers connected by proximity networks increase the bandwidth for social messages. They enable people to send and observe digital signals (in addition to non-digital), allowing new forms of *digital curiosity*. Most importantly, however, interactions that take place in digital space can facilitate and promote interactions in social space, and can lead to rich human encounters. The assumption of this dissertation is that, if properly designed, wearable technology can indeed beget community.

An important characteristic of wearable communities is that social concerns become difficult to separate from technical practices. To deal with the complexity of intervening technological and social issues we propose a methodology and associated platform for building wearable community applications. In Chapter IV, we will describe the WearCoM wearable community methodology, followed in Chapters V and VI by a description of the Proem wearable community platform. Before that, in Chapter III, we will explore the technological foundations of wearable community systems and discuss precursors to wearable community systems. An evaluation will be presented in Chapters VII and VIII, where we address the utility and practicality of the methodology and software platform. In particular, we will present case studies and report on our experiences of using both in software engineering education.

Chapter III

BACKGROUND AND RELATED WORK

The design and development of wearable community systems touches on a number of research areas including wireless ad hoc networking, peer-to-peer computing, and communityware. This chapter provides an overview of the issues and challenges in these areas as they relate to wearable communities systems. The main contribution of this chapter is a survey of systems to support face-to-face interactions. Although the enabling technologies to support face-to-face interactions among co-located individuals have become available over the last years only a few systems have been realized. The main outcome of this chapter is an understanding of the technical issues of wearable community systems.

III.1 Wireless Ad-Hoc Networking

Wireless ad hoc networks and wireless personal area networks are the most important enabling technology for wearable communities. A wireless ad hoc network is a self-organizing network comprised of autonomous nodes that cooperate in order to dynamically establish communications. Nodes may be highly mobile or stationary, and may vary widely in terms of their capabilities and uses. Each node in a wireless ad hoc network functions as both a host and a router, and the control of the network is distributed among the nodes. The network topology is in general dynamic, because the

connectivity among the nodes may vary with time due to node up and downtimes and the possibility of having mobile nodes.

Wireless personal area networks (WPAN) are a special class of ad hoc networks. They are used for interconnecting devices centered around an individual person and are characterized by short-range communication links and a low cost, low power design.

Ad hoc networks were initially developed for use in battlefields under the name packet radio networks (Tornow 1987; Leiner and Nielson 1987). Today, there are two major types of wireless ad hoc networks:

- A *mobile ad hoc network* (MANET) is an ad hoc network composed of mobile nodes (IETF 2002). Since nodes can move, the network topology may change rapidly and unpredictably. The network is decentralized, where all network activity including discovering the topology and delivering messages is executed by the nodes themselves. A mobile ad hoc network is typically used in connection with mobile, handheld and wearable computers. Applications made possible by mobile ad hoc networks include mobile patient monitoring, emergence response, and distributed command and control systems.
- A *smart sensor networks* consists of sensors spread across a geographical area. Each sensor has wireless communication capability and sufficient intelligence for signal processing and networking of the data. Smart sensor networks are used by the military to detect enemy movements, in environmental sciences to monitor environmental changes (e.g., air quality), and in traffic control to monitor vehicle traffic on highways or in a congested parts of a city.

In the following discussion, we will focus on mobile ad hoc networks as they are the primary enabling technology for wearable communities.

Mobile ad hoc networks represent an alternative to the traditional infrastructure model of mobile communication. The objective of this new network architecture is to achieve increased flexibility, mobility and ease of management relative to infrastructure

wireless networks such as cellular voice and data networks. Compared to these networks, mobile ad hoc networks have the following advantages:

- No required infrastructure: mobile ad hoc networks do not rely on wired base-stations and for that reason can be deployed in places without existing infrastructures. They can be created spontaneously and on as needed basis, because they require little configuration to setup. As result, they can be deployed in situations where no other communication infrastructure exists or where such infrastructure cannot be used because of security, cost, or safety reasons.
- Self-organization: In a wired network the connection topology of nodes is determined by the physical cabling and thus is fixed. This restriction is not present in mobile ad hoc network: as soon as two nodes are within hearing distance of each other, an instantaneous link between them is automatically formed. As a consequence, the network topology of a mobile ad hoc network reflects the relative distance of its nodes and is continuously reconfigured as nodes come within reach of each other.
- Fault tolerance: The self-organizing nature of mobile ad hoc networks and the fact that they don't rely on dedicated base stations makes them fault-tolerant. In a traditional cellular network, a fault in the base station will impair all nodes in its cell. In mobile ad hoc networks, a malfunction in one node can be easily overcome through network reconfiguration.

The advantages of ad hoc networks come with certain drawbacks. Because transmission range of ad hoc networks is limited by power constraints, frequency reuse and channel effects, store-and-forward packet routing is required over multiple-hop wireless paths. Because communication hosts can move freely and independently of one another, routing is a difficult challenge and has become the major research topic related to ad hoc networks.

III.1.1 Wireless Communication Technologies for Mobile Ad Hoc Networks

Mobile ad hoc networks can be implemented on a variety of communication technologies including the IEEE 802.11 (IEEE 1999) and HiperLAN (ETSI 2002) standards for wireless local area networks (WLAN), the IEEE 802.15 standard for wireless personal area networks (WPAN) (IEEE 802.15) and future ultra-wideband wireless networks (UWBWG 2002).

IEEE 802.11 Local Area Networks

The IEEE 802.11 standard (IEEE 1999) is a family of specifications created by the Institute of Electrical and Electronics Engineers Inc. for wireless local area networks. The specifications specify a shared media "over-the-air" interface between wireless nodes and address both the Physical (PHY) and Media Access Control (MAC) layers. The IEEE 802.11 standard is based on a single channel, broadcast media and is the wireless equivalent of Ethernet. The IEEE 802.11 standard currently defines four separate but backward compatible specifications: 802.11, 802.11a, 802.11b, and 802.11g. The most recently approved standard, 802.11g, offers wireless transmission over relatively short distances (25 meter) at up to 54 megabits per second (Mbps) in the x-GHz range of the radio frequency (RF) spectrum compared with the 11 megabits per second of the 802.11b standard. The range of 802.11b can exceed 500 meters but is typically no more than 150m outdoors and significantly less indoors.

802.11 networks can be operated in two different modes:

- *Infrastructure mode* is the most common operating mode. It connects the wireless node to a wired network through one or more wireless access points (bridge). If the range of the different access points overlap each other it is possible to move from one cell to another without losing network connection (roaming).

- The *independent mode* (or *ad hoc mode*) realizes a standalone ad hoc wirelesses network in which all nodes share the same medium and are directly connected. An access point is not required and communication is possible between all nodes within direct transmission range.

HiperLAN

HiperLAN (ETSI 2002) is a set of wireless local area network communication standards primarily used in European countries and provides features and capabilities similar to those of the IEEE 802.11 WLAN standards. There are two specifications: HiperLAN/1 and HiperLAN/2. HiperLAN/1 provides communications at up to 20 Mbps in the 5-GHz range of the radio frequency (RF) spectrum. HiperLAN/2 operates at up to 54 Mbps in the same RF band. Similar to 802.11 networks, HiperLAN is based on single channel, broadcast based wireless media.

Ultra-Wideband Radio

Ultra-wideband (see UWBWG 2002) (also known as UWB or as digital pulse wireless) is a wireless technology for transmitting large amounts of digital data over a wide spectrum of frequency bands with very low power for a short distance. Unlike conventional radio systems including that operate within a relatively narrow bandwidth (e.g., IEEE 802.11, HiperLAN, and Bluetooth) ultra-wideband operates across a wide range of frequency spectrum by transmitting a series of very narrow and low power pulses. Ultra-wideband radio not only can carry a huge amount of data over a distance up to 80 meters at very low power (less than 0.5 milliwatts), but has the ability to carry signals through doors and other obstacles that tend to reflect signals at more limited bandwidths and a higher power. The combination of broader spectrum, lower power and pulsed data means that ultra-wideband causes less interference than conventional narrowband radio solutions, and delivers wire-like performance in an indoor wireless environment.

Wireless Personal Area Networks

A wireless personal area networks (WPAN) is a short-range wireless network for interconnecting devices centered around an individual person. The most prominent WPAN example is Bluetooth (Bluetooth 2002), which was used as the basis for a new IEEE 802.15 standard (IEEE 2002). Bluetooth is based on a low-cost, low-power transceiver chip that transmits and receives data in the 2.45 GHz frequency band at a maximal rate of 2 megabit per second. Transmission range is limited to 10 meters.

Unlike IEEE 802.11 and HiperLAN networks, which are based on single channel, broadcast based wireless media, Bluetooth is based on a frequency hopping physical layer. This fact implies that hosts are not able to communicate unless they have previously discovered each other by synchronizing their frequency hopping patterns. In contrast, for single channel networks in ad hoc mode the distance relationship between the nodes implicitly (and uniquely) determines the topology of the ad hoc network. In a Bluetooth network, only those nodes which are synchronized with the transmitter can hear the transmission, even if all nodes are within direct communication range of each other.

Bluetooth supports point-to-point, point-to-multipoint, and multi-hop communication over wireless medium (any-to-any communication will be part of a future release). Point-to-multipoint connections are referred to as piconets. A piconet consists of one master (the owner or creator of the network) and up to seven slaves. Several piconets can be established and linked together spontaneously creating what is called a scatternet. A scatternet represents a Bluetooth-based mobile ad hoc network, albeit one whose overall topology is divided into several piconets.

To support any-to-any communication among a set of Bluetooth devices that have no knowledge of their surroundings requires to build a scatternet in which pairs of nodes (which can communicate with each other) form a connected graph. Several topology creation algorithms have been devised to achieve this goal (Salonidis et al. 2001a; Salonidis et al. 2001a; Záruba et al. 2001; Groten and Schmidt 2001).

III.1.2 Research Issues

The two most prominent research areas in ad hoc networking are routing and multicasting.

Routing

Communication between arbitrary hosts in a mobile ad hoc network requires routing over multi-hop wireless paths because of the limited propagation range of wireless radios (Figure 5). Conventional routing is inadequate in ad hoc networks, as the mobility aspect can cause rapid and frequent changes in network topology. The main difficulty arises because (1) without a fixed infrastructure these paths consist of wireless links whose end-points are likely to be moving independently of one another (2) the capacity of a wireless link varies over time, and its utilization is often low due to noise, interference and contention (3) nodes typically have very limited energy supplies (batteries) and must transmit and receive sparingly to conserve energy (4) nodes may temporarily disconnect from the network for a "sleep" period to save energy when traveling out of network range. These issues make routing in an ad hoc network a difficult problem which cannot be solved with current Internet routing protocols. Routing protocols for ad hoc networks are responsible for maintaining and reconstructing the routes in a timely manner as well as establishing the durable routes. In addition, routing protocols are required to perform these tasks in a manner that is efficient in bandwidth and energy consumption.

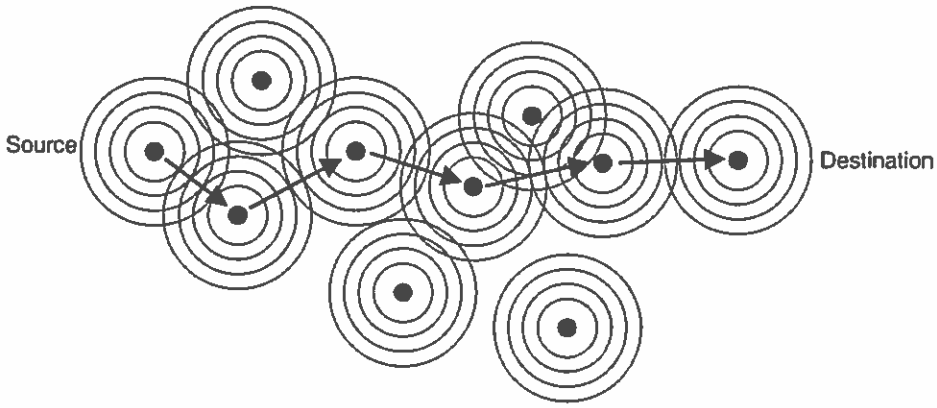


Figure 5. Multi-hop routing in an ad hoc network

Routing protocols proposed for mobile ad hoc wireless networks can be divided into three categories according to the routing strategy.

First, there are distance vector protocols. Protocols of this type include Wireless Routing Protocol (WRP) (Murthy and Garcia-Luna-Aceves 1996), Destination Sequence Distance Vector (DSDV) routing protocol (Perkins and Bhagwat 1994), and Least Resistance Routing (LRR) (Pursley and Russel 1993).

Second, there are protocols that are based on link state algorithms. Protocols such as Global State Routing (GSR) (Chen and Gerla 1998), Fisheye State Routing (FSR) (Pei et al 2000a), Adaptive Link-State Protocol (ALP) (Garcia-Luna-Aceves and Spohn 1998), Source Tree Adaptive Routing (STAR) (Garcia-Luna-Aceves and Spohn 1999), and Landmark Ad Hoc Routing (LANMAR) (Pei et al. 2000b) fall into this category.

Third, there are on-demand routing protocols (Maltz et al. 1999) that are proposed for ad hoc networks only. On-demand routing protocols do not maintain route to each destination of the network on a continual basis. Instead, routes are established on demand by the source. When a route is needed by the source, it floods a route request packet to construct a route. Upon receiving route requests, the destination selects the best route based on route selection algorithm. Route reply packet is then sent back to the source via the newly chosen route. In on-demand routing protocols, control

traffic overhead is greatly reduced since no periodic exchanges of route tables are required. Numerous protocols of this type have been proposed including Lightweight Mobile Routing (LMR) (Corson and Ephremides 1995), Dynamic Source Routing (DSR) (Johnson and Maltz 1996), Ad-Hoc On Demand Distance Vector (AODV) routing (Perkins and Royer 1999), Associativity-Based Routing (ABR) (Toh 1997), and Multipath Dynamic Source Routing (MDSR) (Nasipuri and Das 1999).

Multicasting

In a typical ad hoc environment, network hosts work in groups to carry out a given task. Therefore, multicast plays an important role in ad hoc networks. In addition, multicasting is important for efficient dissemination of data throughout a network. For example, service discovery often relies on disseminating service advertisements to interested partners. Multicast protocols used in static networks (e.g., Distance Vector Multicast Routing Protocol (DVMRP) (Deering and Cheriton 1990)) do not perform well in wireless ad hoc networks because multicast tree structures are fragile and must be readjusted as connectivity changes, yielding excessive processing and network overhead. Hence, the tree structures used in static networks must be modified, or a different topology between group members (i.e., mesh) need to be deployed for efficient multicasting.

Many different protocols for multicasting in mobile wireless networks have been proposed in recent years. Acharya and Badrinath (Acharya and Badrinath 1996) were the first to address the issue of wireless multicast. However, their protocol assumes a fixed wired network component and mobile hosts that can only receive multicast packages but not send. Many multicasting protocols have been recently proposed for ad hoc networks. The Reservation-Based Multicast (RBM) routing protocol (Corson and Batsell 1995) builds a core (or a Rendezvous Point) based tree for each multicast group. RBM is a combination of multicast, resource reservation, and admission control protocol where users specify requirements and constraints. The Ad

hoc Multicast Routing protocol utilizing increasing id numbers (AMRIS) (Wu and Tay 1999) builds a shared-tree to deliver multicast data. Each node in the multicast session is assigned an ID number and it adapts to connectivity changes by utilizing the ID numbers. A multicast extension of Ad Hoc On-Demand Distance Vector (AODV) routing protocol has been newly proposed in (Perkins and Royer 1999). Its uniqueness stems from the use of a destination sequence number for each multicast entry. The sequence number is generated by the multicast grouphead to prevent loops and to discard stale routes. Another multicast algorithm based on neighbor relationships is described in (Lee and Kim 2000).

III.2 Peer-to-Peer Computing

Peer-to-peer computing has gained popularity thanks to the recent emergence of file sharing applications over the Internet. Peer-to-peer computing leverages available computing performance, storage, and bandwidth found on systems around the Internet. With distributed computing applications like SETI@home (Korpela et al. 2001) millions of users contribute their computing resources to work on a common computational analysis. Instant messaging services enable users to communicate and collaborate with their peers in real time. In addition, file-sharing applications like Napster, Gnutella (CLIP2 2002) (Kan 2001), and Freenet (Clarke et al. 2001) offer a compelling and intuitive way for Internet users to find and share files directly with each other.

The term "peer-to-peer computing" can be interpreted in at least two different ways. First, it can emphasize the users' perspective and refer to a system where users are at the same time users and providers of information. Such systems are based on the principle that users who are willing to share files or resources with others may gain benefits in the long run. Second, it can refer to the architecture of a distributed system. This view emphasis the distinction of peer-to-peer computing from traditional client-server computing by insisting that in order to qualify as a peer-to-peer system it must

not have a central server or coordinator and communication between hosts must be pair wise. Proponents of the user-centered view see a peer-to-peer system as technology that supports sharing and collaboration among independent users. This view was popularized by Napster, a peer-to-peer file sharing application that enables pair wise interactions between users although architecturally it relies on a central server. Proponents of the technical point of view see a peer-to-peer architecture as a way to tackle issues like system scalability, reconfigurability and user privacy and anonymity. Ultimately, peer-to-peer is about overcoming the barriers to the formation of ad-hoc communities, whether of people, of programs, of devices, or of distributed resources. In this dissertation we follow a definition of peer-to-peer computing expressed by (Bolcer et al. 2000) who describe peer-to-peer as "any relationship in which multiple, autonomous hosts interact as equals. An autonomous host is useful in its own right even in the absence of others. The peering relationship implies that additional functions are available to other peers *collectively* as a consequence of their collaborations with other hosts. Known as the *network effect*, the value and extent of these added powers increases dramatically as the number and variety of peers grows" (Bolcer et al. 2000, 3).

Although the concept of peer-to-peer computing is far from new, its application to modern distributed computing systems and applications brings in new problems and research challenges including networking protocols, decentralized algorithms, middleware, software architectures, coordination models, and security.

III.2.1 Mobile Peer-to-Peer Computing

Traditionally, mobile devices have been designed as thin clients as part of a client-server system. For example, cell-phones and Internet-enabled PDAs use wireless connections to gain access to resources such as data and computation provided by large central servers. With the advent of wireless ad hoc networks, it becomes possible to design mobile systems as peer-to-peer systems.

A mobile peer-to-peer system (MP2P) (Mascolo et al. 2001; Gold and Mascolo 2001; Kortuem et al. 2001; Kortuem 2002; Charas 2001) is a distributed mobile system that consists of mobile hosts that continuously change their physical location and establish peering relationships among each other based on proximity. Hosts interact during brief physical encounters thereby engaging in short-haul wireless exchanges of data. Mobile P2P applications take advantage of resources -- storage, cycles, content, human presence – provided by mobile devices in the immediate physical proximity. Such systems closely match Bolcer's definition of peer-to-peer computing: mobile devices are autonomous and provide great benefits to their users even when not connected to other devices. However, as soon as two devices come within reach and a communication link is established, they enter into a short-lived, yet mutually beneficial partnership by exchanging data and accessing each other's services.

Mobile peer-to-peer systems differ from client-server based distributed mobile systems in the following respects:

- Mobile P2P systems are decentralized: there is no central node and all peers have the same roles and responsibilities.
- Communication links are highly transient: disconnections and reconnections occur frequent and unpredictably.
- Since mobile hosts can move frequently and independently of one another, mobile P2P systems have rapidly and unpredictably changing topologies.

Similarly, there are important differences between traditional (non-mobile) and mobile peer-to-peer system. (Bolcer et al. 2001) describe (non-mobile) peer-to-peer computing as the natural and desirable outcome of three profound and pervasive trends:

- The ease of interconnection
- The expansion of bandwidth
- The wealth of cycles

"... in a world where (network access) interconnection is universal, (network) bandwidth is plentiful, and (processor) cycles are inexpensive, peering among physical unequals is both natural and desirable" (Bolcer et al. 2001, 3).

Traditional peer-to-peer systems such as Napster, Gnutella and Freenet, which are intended to run on stationary hosts in wired networks, have been designed with this view in mind. However, two of these trends do not apply to mobile peer-to-peer systems: while ad hoc networks provide ease of interconnection, there is much less expansion of bandwidth in wireless networks than there is in wired networks, and much less wealth of cycles in mobile devices than there is in desktop units.

III.2.2 Challenges

The unique character of mobile peer-to-peer systems represents a significant challenge for the designer. Not surprisingly, mobile peer-to-peer computing exhibits the combined issues and challenges of ad hoc networking, mobile computing, distributed computing and peer-to-peer computing. In the following, we will summarize these challenges.

Mobile Device Limitations

Mobile devices present a more constrained computing environment compared to workstations and desktop computers. Because of fundamental limitations of battery life and form factor, mobile devices tend to have less powerful CPUs, less memory, smaller and less reliable storage, restricted power consumption, smaller displays, and missing or restricted input devices. The limitations in processing power and storage make it harder to implement algorithms for distributed coordination and cryptographic security measures.

Communication

Mobile peer-to-peer systems exhibit all the limitations (and advantages) of ad hoc networks. Because of fundamental limitations of power, available spectrum, and mobility, ad hoc networks tend to be low bandwidth and long latency. As bandwidth increases, the device's power consumption also increases further draining the already limited battery life of mobile devices. Thus, even as wireless networks improve their ability to deliver higher bandwidth, the power availability still limits the effective throughput.

As discussed above, communication links in ad hoc networks are prone to unexpected interruptions. Consequently, mobile peers must anticipate frequent network failures and handle them gracefully. In addition, peer applications should provide for disconnected operations (Kistler and Satyanarayanan 1992) such that a peer remains operational even without network connection. The instability of multi-hop paths and the limited lifetime of routes in ad hoc networks have a negative impact on the performance on peer-to-peer routing. For example, the Gnutella file-sharing software uses routing as a way to return search results to the peer that initiated a search. Search results travel backwards on the same route the original search query took. If one of the intermediate peers disappears from the network before the search result could be returned, the search result will be lost. In our own experiments using Gnutella over the Internet we experienced that about 2% of all search results could not be returned. In an ad hoc network, this number would increase dramatically.

Naming

Traditional (non mobile) peer-to-peer systems are characterized by an increasing decentralization and autonomy of hosts. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, peer-to-peer systems often operate outside the DNS system.

The same must be true for mobile peer-to-peer systems. Additional reasons for not relying on the DNS system are:

- In ad hoc networks, access to a central DNS server cannot be assumed
- Not all mobile devices support IP networking and thus do not have IP addresses
- Collaborative applications require the ability to identify not only devices, but also users.

Resource Discovery

One of the things that makes current peer-to-peer system so powerful is that they take advantage of resources -- storage, cycles, content, human presence -- available at the edges of the Internet. In a mobile peer-to-peer system we want to take advantage of resources provided by peers running on mobile devices in the immediate physical proximity. Because of the unpredictable physical mobility of mobile devices, discovering resources becomes a challenge.

The highly dynamic nature of mobile peer-to-peer systems requires similarly dynamic mechanisms for device and resource discovery. In ad hoc networks, device discovery is part of the network; resource discovery, however, is the task of the peer system. We need algorithms through which a computing device can detect the presence of neighboring devices, share configuration and service information with those devices, and notice when devices become unavailable. Resource discovery must be timely (in order to detect moving devices) and efficient (so not to overload the network). One such algorithm has been developed as part of the DEAPspace project (Nidd 2000).

In contrast to peer-to-peer systems that are targeted at fixed networks, decentralization is not a mere option for mobile peer-to-peer networks, but a necessity. Even seemingly decentralized peer-to-peer systems such as Gnutella rely on centralized servers for some tasks. For example, Gnutella clients use a central host cache for determining initial entry points into the Gnutella network. In mobile peer-to-peer systems all functions need to be decentralized.

Data Sharing and Synchronization

A mobile peer-to-peer system is basically a highly dynamic, decentralized distributed system with weakly connected mobile hosts. In order to cooperate, peers need to be able to share and synchronize data. The extreme decentralization and unpredictability of mobile peer-to-peer system together with the fact that peers always only establish pair-wise connections leads to the following conflicting requirements:

- High availability: mobile peers needs to be as autonomous as possible and they should be able to perform computations even in the absence of connections with other peers. This requirement can be achieved with a replicated data storage scheme where each peer maintains a local copy of shared data object.
- Consistency: Replicated data storage introduces the problem that local copies can be updated independently by different peers and thus might become inconsistent over time. This requires a synchronization mechanism that is able to handle weakly connected hosts and unpredictable network links.
- Timeliness: Any solution to the consistency problem has to deal with the fact that data might be shared across a group of peers that rarely come together. Even more, there is no guarantee that a specific group of peers will ever be connected to each other at the same time. Consequently, updates must be propagated throughout a network by passing information from peer to peer. Because of the unpredictability of host mobility, it is impossible to guarantee that each host receives update information in a timely manner.

Trust and Security

A particular security aspect of mobile peer-to-peer systems relates to the question "how do we know we can trust somebody on the network?" In systems with a centralized component, this problem can be handled by public key certificates issued and cryptographically signed by globally trusted certification authorities (CA). A public

key certificate proves that its holder is trustworthy simply because the issuer, the trusted CA, has signed it and can therefore vouch for the holder's credentials. A chain of CAs, each trusting the next CA in the chain, may sometimes be necessary when a certificate signed by an unknown CA is presented.

In a mobile peer-to-peer system global trust authorities are difficult or even imposable to maintain, because connectivity is to such an authority cannot be guaranteed. Thus a mobile peer-to-peer system requires distributed authentication protocols. This, however, is made difficult by the fact that this must occur in a completely decentralized environment with no or intermittent connection to a trusted authority. Possible solutions might include the use of reputations (Schneider et al. 2000; Abdul-Rahman and Hailes 2000).

In order to engineer a fully secure system it becomes necessary that the device is able to authenticate the user. Otherwise digital certificates can easily be stolen by taking the device itself. Depending on the required security level this might require biometric identification.

Privacy

Privacy is the right of individuals to control collection and use of personal information about themselves. Unlike security, which deals with safeguarding of information from unauthorized users, privacy is concerned with the amount of information known about an individual. One of the main privacy concerns is protecting a user's anonymity. Monitoring network traffic or gaining access to confidential personal data can compromise a user's anonymity. Not only must a system prevent spying and monitoring, but users must also be given control what information is disclosed, to whom, and when. In particular, it must be possible for an individual to stay anonymous if so desired.

Building mobile peer-to-peer systems requires a variety of skill-sets ranging from collaboration paradigms based on opportunistic proximity-based interactions to highly dynamic wireless networks. As a consequence, the software infrastructure to support mobile peer-to-peer applications is becoming critically important. Recognizing this need researchers have started to develop mobile peer-to-peer platforms. Among them are platforms that exclusively focus on data sharing aspects (XMIDDLE (Mascolo et al. 2001), Lime (Murphy et al. 2001)), mobile groupware platforms (Pocket DreamTeam (Roth 2002)), Bluetooth-specific platforms (BlueTalk (Pocit Labs 2001)), and general purpose platforms (JXTA for J2ME = JXME (Arora et al. 2002)). We will discuss these platforms in more detail in Chapter VIII.

III.3 Communityware

With the advent of online communities, an interest in *communityware* emerged. Communityware, a term coined by Ishida (Ishida 1998a; Ishida 1998b), is software for supporting and enabling communities. Communityware is intended to support diverse and amorphous groups of people and focuses on the process of organizing people who are willing to reach some mutual understanding.

Examples of communityware are:

- Recommendation systems and collaborative filtering systems (e.g., book recommendations at Amazon.com)
- Matchmaking software (systems that identify similarities in user's interests with the purpose to facilitate direct communication)
- Collaborative information collection tools with awareness and notification support (e.g. document repositories, shared bookmark applications)
- Comprehensive solutions for building community web sites

Communityware is related to the more familiar class of systems known as *groupware*. Peter and Johnson-Lenz, who are credited by many as coining the term

groupware (Johnson-Lenz and Johnson-Lenz 1981), define groupware as "intentional group processes plus software to support them" (Johnson-Lenz and Johnson-Lenz 1982, 4). This definition properly excludes multi-user databases and electronic mail that are not designed particularly to enhance the group process. According to (Johnson-Lenz and Johnson-Lenz 1981), a complete groupware infrastructure has three dimensions: communication, collaboration (shared information and building shared understanding), and coordination (delegation of task, floor control, etc.).

Communityware is intended to support social processes of diverse and amorphous groups of people. This is in stark contrast to groupware which typically focuses on work-related activities of already organized people, such as teams. Teams are composed of members who know each other and collaborate to achieve a common goal while community members have just common interests or preferences. Thus communityware aims to support the process of organizing people who are willing to reach some mutual understanding. In other words, communityware focuses on the early stage of collaboration, which includes group formation, contact facilitation, finding people, and building a common understanding.

The differences between groupware and communityware are summarized in Table 4.

Table 4. Differences between Groupware and Communityware

	Groupware	Communityware
User population	Small	Potentially large
Degree of cooperation among users	High	Low-high
Motivation for participation	External pressure, Common goal	Shared interests, Self interest
Relations between users	Everyone knows everyone else	Not everyone knows everyone
Collaboration means	Shared artifacts	Occasional exchange

III.4 Systems to Support Face-to-Face Interactions

The notion of wearable communities defines a research framework for investigating computer support for social interactions in face-to-face settings. Such interactions are often spontaneous (driven by chance encounters of mobile people) and highly situated (embedded in the real world context in which people interact). In the literature, spontaneous unstructured interactions are referred to as *informal communication* (Whittaker et al. 1994; Fish et al. 1990; Fish et al. 1993). Typically short, expressive, and frequent, informal communication may be initiated by one party or serendipitously occur through chance meetings. (Fish et al. 1990) describe informal communication as a social event that takes place ad hoc when there is an opportunity for communication. It can be strongly related to work (Bergqvist et al. 1999; Kraut et al. 1990; Luff and Heath 1998; Belotti and Bly 1996), but it can also be purely social. Informal communication has no pre-established agenda, and no pre-established topics are discussed. Informal interactions play an important part in our social life and are vital for coordination of work activities at the office.

Despite the importance of face-to-face and informal communication in our daily lives information technologies to support interactions between people have mostly been targeted at planned rather than spontaneous communication, in settings that are virtual (shared virtual space) or artificial (instrumented meeting rooms) rather than part of people's everyday. For example, the CoLab (Stewart et al. 1999) project features an electronic meeting room that enables a group of users to control a shared display at the front of the room. The Pebbles project (Myers et al. 1998), investigates the use of handheld Personal Digital Assistants (PDAs) as portable input devices for a single shared display. In addition, up until recently research aimed at supporting informal communication has focused on distributed (Dourish and Bly 1992; Fish et al. 1993; Fitzpatrick et al. 1998; Nakanishi et al. 1996) rather than co-located groups.

With the advance mobile and wearable technology some researchers have started to recognize the social potential of this technology and have build systems to

support face-to-face communication. Among the early proposals are match-making technologies (e.g. LoveGety (CNN 1998)), awareness devices to provide roaming groups with a sense of connectedness (e.g. HummingBird (Holmquist et al. 1999)), ad hoc games and gaming platforms that seek to make real-world group mobility part of digital entertainment (e.g. Pirates (Björk et al 2001), Mercantile (Pering and Pering 2001), Pervasive Clue (Schneider and Kortuem 2001)), educational software tools (e.g., Geney (Danesh et al. 2001)) and messaging devices that adopt 'word of mouth'-metaphors for proximity-based passing of information (e.g. ThinkingTags (Borovoy et al. 1996), MemeTags (Borovoy et al. 1998), iBalls (Borovoy et al. 2001)).

The main characteristics of these systems are summarized in Table 5.

These systems represent important precursors of wearable community systems. In our survey we limit our attention to systems that make use of mobile or wearable devices. This excludes, for example, systems like AgentSalon (Sumi and Mase 2001) where users interact in front of a shared display without devices on their own. In the following we will provide a short description of the overall design and purpose of these systems and then discuss technical and architectural aspects.

III.4.1 Design Overview

Thinking Tags

The Thinking Tag (Borovoy et al. 1996) is a small badge used to initiate communication among co-located people at events involving large crowds. The purpose of Thinking Tags is to determine if two users share common interests. The badge itself consists of five LED signs and an IR-transceiver. Each user defines his or her interests during an initial setup procedure by connecting the tag to a terminal and answering a number of yes/no questions related to the event, e.g. "Are you interested in the Y2K problem?" The tags compare the user preferences and indicate, by lighting one or

several LED signs, if the two persons have anything in common. The idea behind Thinking Tags is that users can use the knowledge about shared interests to initiate a conversation.

Table 5. Systems to Support Face-to-Face Communication: Overview

	User Population	Interaction Type	Objectives
MemeTags	Community (strangers)	Unplanned	Creating common ground, contact facilitation
ThinkingTags	Community (strangers)	Unplanned	Creating common ground, contact facilitation
iBalls	Children		Entertainment, creating common ground
Hummingbird	Group (friend, Co-workers)	Unplanned, opportunistic	Group awareness, contact facilitation
ProxyLady	Co-workers	Opportunistic	Group awareness, contact facilitation
NewsPilot	Co-workers	Opportunistic	Group awareness, contact facilitation
Pervasive Clue	(no characteristics)	Unplanned, opportunistic	Entertainment
PIRATÉ s	(no characteristics)	Unplanned, opportunistic	Entertainment
Mercantile	(no characteristics)	Unplanned, opportunistic	Entertainment
Geney	Children	Planned	Entertainment, Education

Meme Tag

The Meme Tag (Borovoy et al. 1998) is a wearable display device worn around the neck, its display facing a conversation partner rather than its wearer. The purpose of Meme Tags is to propagate memes, contagious ideas, opinions or messages, throughout a network of Meme Tags worn by participants of a large scale event. If one Meme Tag has a meme that the other does not have, the wearer of that other device is offered to accept a transfer of that message. The idea is that users will only accept memes they like or agree with. Similar to Thinking Tags, the idea is to initiate conversations by letting people know if they subscribe to the same memes or not.

A Meme Tag is proactive in that it initiates communication with other Meme Tags without explicit user action, but ultimately the user authorizes the message transfer by explicitly pressing a button. Hence, no messages can propagate over the "network" unless users specifically agree to host them and, in fact, this is the main idea behind the Meme Tag application.

Hummingbird

The Hummingbird (Holmquist et al. 1999) is a small wearable device equipped with a short-range radio transceiver, through which it broadcasts its identity and receive information about other Hummingbirds in the vicinity. The Hummingbirds were developed to "give members of a group continuous aural and visual indications of which other group members are in the vicinity" (Holmquist et al. 1999, 2). Whenever two or more Hummingbirds are close enough to communicate, the devices give a subtle audio signal and display the identity of the other devices in the proximity. The devices are functionally self-contained, i.e. non-dependent of surrounding infrastructure. A Hummingbird extends the range of our ordinary senses, allowing the user to know that a colleague is nearby even though he or she is not close enough to be directly seen. Inspired by what is often within "shouting distance", the communication range is set to

approximately 100 meters (depending on the number and nature of obstacles like people, walls etc.).

Hummingbirds are an example of an Inter-Personal Awareness Device (IPAD) as defined by Holmquist (Holmquist et al. 1999). In contrast to a communication device like the mobile phone, an IPAD facilitates contacts instead of mediating them. An important basis for the IPAD concept comes from the observation that informal communication may occur whenever people are in the same place, but that it does not necessary matter which place they happen to be in. In this sense, IPADs seek to support spontaneous interactions between people in vicinity.

ProxyLady

Proxy Lady (Dahlberg et al. 2001) is a mobile system which objective is to foster opportunistic face-to-face communication, for example during unexpected encounters in corridors and coffee rooms. An opportunistic meeting is an event anticipated by one but not all parties involved (Kraut et al. 1990). Proxy Lady enables user to associate *information items* such as email messages and text files with people, called *candidates for interaction*. When a candidate is in the proximity of the proxy lady's users, the device notifies its owner, which can initiate a face-to-face conversation with this person and (possibly) start a brief conversation or meeting relating to the information item. The information items are downloaded to the PDA during synchronization with a desktop computer. The use of email and files as trigger for interactions implies that users are acquainted or have communicated prior to the encounter. Typically, this limits Proxy Lady to use in workgroup and work-related informal communication.

In its basic conception Proxy Lady is very similar to Hummingbirds, but provides more flexibility. It extends the basic IPAD concept by supporting opportunistic meetings, i.e. meetings expected by one but not the other party. In response to privacy concerns voiced by users of Hummingbirds, Proxy Lady provides

an 'invisible mode'. If a device is in invisible mode, nearby users are unable to detect its presence. The range of the radio transmitters is fixed at approximately 20 meters.

NewsPilot

The NewsPilot (Dahlberg, Redström, and Fagrell 1999; Dahlberg, Fagrell, and Redström 1999) is an Inter-Personal Awareness devices based upon Palm handheld devices. Very similar to ProxyLady, NewsPilot uses radio transceivers to detect and inform the user about nearby NewsPilot users. As triggers NewsPilot uses to-do action items stored in a person's calendar. Each to-do item in the to-do list of the Palm handheld device can be associated with one or several person. When one of those persons is nearby, the to-do item is shown and a small beep is emitted.

Pervasive Clue

Pervasive Clue (Schneider and Kortuem 2001) is a live-action role-playing game based loosely on Hasbro's classic board game Clue augmented with short-range radio frequency (RF) PDA devices. The goal of Pervasive Clue is to discover who killed the host, Mr. Bauer, where it was done and what was the murder weapon. Solving the murder is done through the discovery of clues, when a player feels they can solve the crime they are allowed to make an accusation. If any of the crime facts (murderer, location or weapon) are incorrect the player is eliminated. Players are each equipped with a *Cluefinder* (Figure 6), an RF enabled PDA device with a large magnifying glass attached. Although the magnifying glass is entirely cosmetic its design illustrates the function of the device, how it is used and promotes use of the Cluefinder as a role-playing prop. Each clue has a physical representation (i.e. knife, book, candlestick) as well as a hidden short range RF beacon <1 foot, broadcasting its clue. Players find game clues by searching a room with the Cluefinders and coming within 1 foot of the beacon. Players may also gain clues by exchanging them with other players. The rules of the game do not restrict a player's ability to give or trade clues with other players.

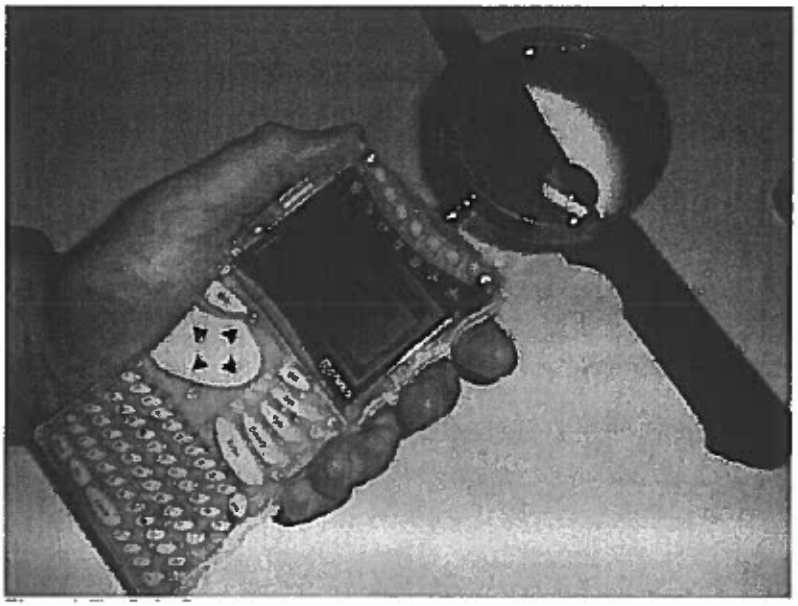


Figure 6. Clue Finder

The Cluefinder PDA accomplishes its task of promoting sociability by: identifying nearby people, semi-automatic support for trading clues and items, capturing and disseminating reputation information about people for use in evaluating a player's trustworthiness based on his or her behavior (like cheating or lying) in past games.

Geney

Geney (Danesh et al. 2001) is a collaborative problem solving application to help children explore genetics concepts. The goal is to encourage players to collaborate and share knowledge in order to complete the task. In Geney, students work together to produce a fish with a particular set of characteristics by exchanging fish with their friends through the handheld's infrared port. Each Geney handheld contains a single pond of fish and students can view the genetic traits of fish in their pond. Two fish in a pond can be mated and the offspring will have genetic traits that are derived from their parents' genes. The goal of the game is to breed a specific creature. Only by

collaborating with other students playing the game can the group achieve the desired goal.

Pirates

Pirates (Björk et al. 2001) is a multi-player computer game that takes place in a fantasy archipelago setting, where each player is the captain of a ship. Game objectives include solving different missions, such as finding treasures, trading with commodities found on the islands, and fighting other players in sea-battles. As missions are completed, the players gain experience points that translate into rank, and as goods are traded for money, ships are upgraded to sturdier ones.

The game is implemented on handheld computers connected in a Wireless Local Area Network (WLAN). A local server maintains and controls the virtual game environment, and keeps a record of what events take place in the game. In addition to the WLAN adapters, each handheld device is fitted with custom-made proximity sensors, used to determine the players' location in physical space. Although the game is played on handheld computers and is maintained by a server, the players must roam a physical environment, the game arena, in order to explore the virtual game environment. To engage in a virtual battle with co-players, they must walk up to them, forcing the players to not only watch the computer screen but also to look at other players and the real world.

Mercantile

Mercantile (Pering and Pering 2001) is a mobile game designed to support face-to-face interaction within an already existing social context, such as the office workplace. In the game, each player represents a trading empire and the computer system requires them to meet face-to-face to negotiate and affect a trade, instead of allowing them to trade in a purely virtual world. The mobile device itself possess no inherent display (e.g., LCD), and instead uses short-range wireless communication to

provide access to a user's data and applications through already-existing nearby displays, such as a desktop workstation or internet kiosk. Even in the virtual world, players can only "see" the holdings of other players if they are physically co-located, encouraging them to actually move around and seek out other players. The overall concept intends to explore mobile computing and distributed gaming in a truly mobile context by focusing on co-located social interaction, rather than relying on the distributed virtual interaction typically supported by mobile game consoles.

III.4.2 Implementation and Architecture

Table 6 summarizes the technical attributes of the systems we examined. It lists the device characteristics, the communication technology, the type of data exchanged between devices, the proximity sensing technology and the overall system architecture.

Devices

The devices used in these systems fall into two categories: proprietary devices and commercial off-the-shelf devices like PalmOS PDAs, Nintendo GameBoys and Cybikos. The proprietary devices use by ThinkingTags, MemeTags are extremely small and computationally not very powerful, while the "Personal Server" used by the Mercantile project is in the same class as commercial PDAs. Most devices have regular displays capable of displaying text and graphics, while others have very simple (LEDs, ThinkingTags) or no (Mercantile) output capabilities at all.

Table 6. Systems to Support Face-to-Face Communication: Technology

	Device	Data Exchange between devices	Communication Technology	Proximity Sensing Technology	System Architecture
Thinking Tags	Small, proprietary, output only, 5 LEDs function as display, no input	Interests (small text snippets representing predefined by answers to five predefined opinion questions;	IR (range < 5m)	(communication)	Decentralized P2P
MemeTags	Small, proprietary, output only, 2x16 character display	Memes (small, user defined text snippets)	IR (range < 5m)	(communication)	Decentralized P2P
iBalls	Key-chain sized game devices from SEGA DreamCast	Software objects with data and behavior	RF (?), range < 5m	(communication)	Decentralized P2P
Hummingbird	Nintendo GameBoy	User id	RF, proprietary, range < 5m	(communication)	Decentralized P2P
ProxyLady	Casio Cassiopeia E-105 PDA	User id	RF, 802.11 WLAN (no direct communication between devices), range < 5m	(communication)	Centralized (for managing database of user ids and email)
NewsPilot	PalmOS PDA	User tasks	RF, proprietary (?),range < 5m	Dedicated RF transmitter	Centralized
Pervasive Clue	Cybiko wireless device	User IDs, clues	RF, built into device, range < 5m	(communication)	Decentralized P2P
Pirates	HP Jornada PDA	User photo	RF (802.11 WLAN), range < 5m	Dedicated RF transmitter	Centralized
Mercantile	"Personal Server"	Trading items,	RF (Bluetooth), range < 5m	(communication)	Decentralized client-server

	(StrongARM-based proprietary mobile device) w/o display	holdings			(Each device is server that connects to shared public display)
Geney	Palm OS PDA	Fish data object	IR (IRDA), range < 5m	(communication)	Decentralized p2p

Communication Technology

All devices are communication enabled, either via infrared (IR) or radio frequency (RF) technology. Some systems use communication standards like IRDA, IEEE 802.11b and Bluetooth, while the rest uses proprietary solutions. The maximum transmission range varies from just under 5 meters to up to 100 meters.

Data Exchange

All systems support data exchange between devices either directly from device to device or indirectly with the help of additional infrastructure. All but one system exchanges data objects such as user ID and text strings: the i-Ball system allows the transfer of small software object, i.e. programs that encapsulate data and behavior.

Proximity Sensing Technology

All devices have the ability to detect the presence of proximate devices. The technologies used to determine if two devices are co-located fall into two categories. Two system (Pirates, NewsPilot) use dedicated proximity sensors, while others combine communication and proximity sensing. With a combined solution two devices are considered to be co-located if they are able to communicate. Since Pirates and NewsPilot are the only systems that do not use a short-range communication technology, dedicated proximity sensors are a necessity. However, dedicated sensors

provide additional advantages: it is easier to modify the distance two devices have to be apart before they are considered co-located.

System Architecture

The systems use three basic architectures:

- Centralized: all devices rely on and communicate with a central system component, such as a database or email server.
- Decentralized, peer-to-peer: there is no centralized system component and all device interaction occur peer-to-peer (i.e. without intermediary).
- Decentralized client-server: the Mercantile system uses an unusual architecture. Mercantile devices do not have any display, but require users to walk up to a shared public display. Only users who are connected to the same display can exchange data. This architecture is decentralized because there is no central system component. At the same time it is client-server, because devices act as servers that provide computational services to a "dumb" display client.

The decentralized peer-to-peer solution has the advantage that the system does not require external communication or computation infrastructure. Thus such a system can bet set up at any place, inside and outside. Compared to a centralized architecture, a decentralized system is usually more difficult to build but provides significantly more freedom for users. The decentralized client-server solution of Mercantile makes it possible to simplify the device (it doesn't need to have a display), but users can only interact in the presence of a shared display.

III.4.3 Problems

The fundamental problem in the development of system to support face-to-face communication is that there is a *semantic gap* between the application layer and system

layer. As of today, there is little or no direct support for the variety of features that such applications require. Likewise, there are no programming and building abstractions for developers to leverage off when designing wearable community applications. This results in a lack of generality, requiring each new application to be built from the ground up in a manner dictated by the underlying network technology and device platform. This direct dependency results in the following problems:

1. *Lack of network transparency.* Applications must be tailored towards specific network technologies and device platforms. As result, the designer must have intimate knowledge of the underlying technologies and deal with low-level aspects.
2. *Limited portability.* For the same reason, applications are difficult to port to other device or network technologies.
3. *Lack of reuse.* There is no code sharing among applications. Instead, each application must be build from scratch requiring the programmers to solve the same problems all over again.
4. *Inability to evolve applications.* If underlying network and device platforms evolve so must applications. Currently it is nearly impossible to rearchitect or evolve applications to incorporate changes in network and hardware or to exploit new features.
5. *Lack of interoperability.* There are no established standards with regard to communication in ad hoc networks and context awareness. Without standardization every application is an island unto itself and applications created by different developers are not able to talk to each other.
6. *Limited feature set.* Due to the time and skills required to build applications they tend to be rather simple and only provide a very rudimentary set of functionality.

This set of problems makes clear that there is a need for the uniform support for designing, building and executing these types of applications.

III.5 Summary

The two fundamental technologies for wearable communities are ad hoc networks and peer-to-peer computing. Despite that fact that ad hoc networking and peer-to-peer computing deal with similar issues such as discovery and routing, there is not much overlap in the research. Most research on ad hoc networks focuses on the lower layer of the protocol stack including the link layer, network layer and transport layer. On the other hand, current peer-to-peer systems are designed for an Internet-like network infrastructure in which stationary hosts are connected by high bandwidth links. The assumptions on which these peer-to-peer systems are built are no longer valid in dynamic ad hoc networking environments. The unique characteristics of such networks require highly adaptable peer-to-peer systems that can react to changes in connectivity and resource availability in a timely and ongoing manner.

Emerging ad hoc and personal area networks open a rich field for innovative groupware and communityware solutions for supporting face-to-face communication. Yet as of today, only a small number of systems that exhibit important characteristics of wearable community systems exist. Current systems suffer from a series of technical shortcoming which must be overcome for the rapid and efficient development of wearable community systems. The dynamic environment created by mobile ad hoc networks together with the social and collaborative nature of wearable community applications makes it hard to architect such systems. The fundamental problem is the *semantic gap* between the application layer and system layer, and the lack of an appropriate development infrastructure. Without it, each new application must be built from the ground up in a manner dictated by the underlying network technology and device platform. In the following we describe the WearCoM wearable community methodology and associated Proem platform that have been designed to address the problems of building wearable communities applications.

Chapter IV
WEARABLE COMMUNITY METHODOLOGY

The fundamental problem this dissertation addresses is how to build software to support wearable communities. In Chapter II, we argued that it is beneficial to divide the software part of a wearable community system into two layers: a generic distributed software infrastructure, also called a wearable community platform, and community-specific application software. A wearable community platform implements common functionality, and provides services and capabilities required by a wide range of wearable community application.

In this chapter, we present the WearCoM methodology for the design and development of wearable community applications. The goal of the methodology is to provide a framework that guides the design and development activities from initial conception to implementation and deployment of an application on the users' devices.

The methodology consists of three components: (1) a conceptual model that defines terminology and an abstract architecture; (2) a design language that addresses the specification of important analysis and design decisions and enables developers to specify key aspects of the application design; and (3) a development process that outlines a sequence of development steps that result in the creation of specific artifacts. These artifacts include specification documents and software code. The WearCoM methodology is supported by the Proem peer-to-peer platform described in Chapters V and VI. An overview of the methodology is depicted in Figure 7.

Our primary goal in developing the WearCoM methodology and the Proem platform is to enable an exploratory design approach based on rapid prototyping. Thus, our main focus is a reduction of the complexity of the development process to a point

where it becomes possible for developers without extensive prior knowledge of wireless networking and wearable computing to build wearable community applications. For that purpose, we opted to employ an informal design language partially inspired by the Unified Modeling Language (UML) (see OMG 2001). The idea is that be making it easier to develop wearable community application, we will facilitate the exploratory creation and investigation of wearable communities.

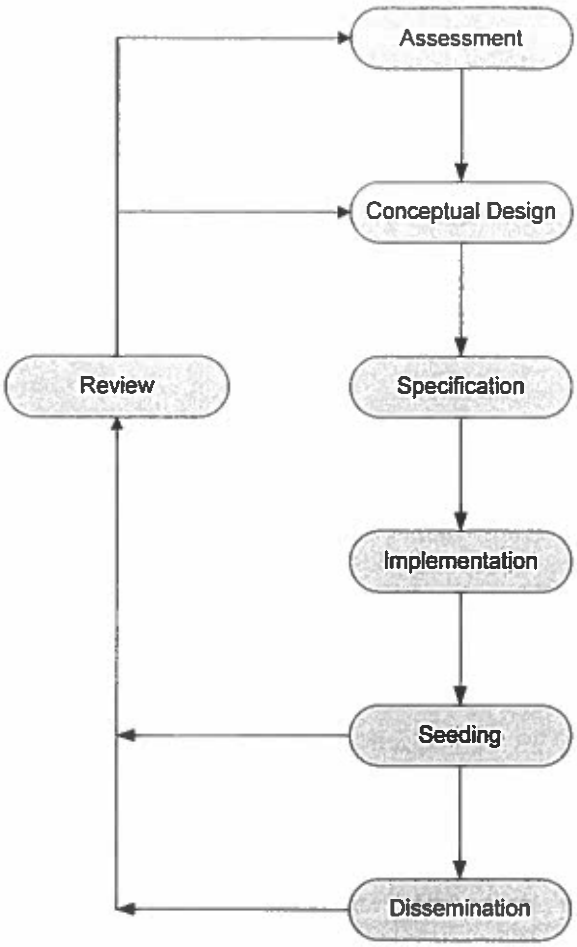


Figure 7. WearCoM methodology overview

IV.1 Conceptual Model

The conceptual model defines the key concepts and components of a wearable community application. These include: community agent, user profile, and community language.

IV.1.1 Community Agents

A wearable community application involves two types of *actors*: *users* and *community agents*. A community agent (*agent*, for short) is a software application that runs on a user's wearable device. An agent functions as intermediary between a user and a community. It has two primary tasks: it announces the presence of its user to nearby agents and enables (indirect) communication between users. It does that by discovering nearby agents and autonomously exchanging data with them. Each agent is community-specific. A user 'owns' one agent for each community he or she is a member of and only agents belonging to the same community can communicate.

We use the term 'agent' to emphasize autonomy both at design time and runtime, and cooperation. An agent is not an interactive application that sits idle until the user interacts with it by means of a graphical user interface. Instead, an agent is an application that exhibits autonomous behavior and acts on behalf and in the interest of its user without requiring direct user intervention: it is proactive and takes the initiative when circumstances are opportune. For example, an agent may autonomously interact with agents of nearby community members and may inform its user about nearby users. In sum, a community agent is software that is personal, proactive, presence-aware, communicative and to a certain extent autonomous. It is predictable and can be fully controlled and monitored by its user.

IV.1.2 Encounters

The second fundamental concept is an *encounter*. We define an encounter as a situation where

- two or more individuals are in close physical proximity to each other;
- the agents of these individuals have discovered each other's presence; and
- the agents are able to communicate.

This definition does not say anything about how agents discover each other, or how close users have to be for discovery to happen. Similarly, this definition does not say whether discovery and communication are independent functions or can be combined into one. Discovery and communication might be based on two different technologies: for example discovery could use dedicated proximity sensors while communication might rely on wireless communicating networks.

Encounters have several important properties:

- Encounters can occur between two, three or more individuals.
- Encounters are situations that last a certain time interval, not momentary events: encounters can be short and last only a few seconds, or they can be long-lasting and going on for hours. For example, the encounter between two individuals passing each other in a hallway might last just a few seconds. Yet, two or more people in a lengthy meeting encounter each other for the full duration of that meeting.
- Encounters are not reflexive: if A encounters B, then B does not necessarily encounter A.
- Encounters are non-transitive: if A encounters B, and B encounters C, then A does not necessarily encounter C.

IV.1.3 User Profiles

A *user profile* is a typed data item that defines the identity of a user within a community. It contains information a user willingly discloses to other community members and may include any information that is defined useful or necessary to identify or describe a person. For example, it may include a user's (real or assumed) name, contact information, a list of hobbies and a photo. While some interactions may be anonymous, most communities depend on an exchange of personal information to introduce community members to each other.

Each community uses a different *profile template* and all user profiles used within one community have the same structure. The profile template standardizes the structure and content of user profiles used within a community.

During an encounter, agents exchange profiles thereby disclosing their users' identity to the other agent. Agents may perform different operation depending on the content of a received profile: they might inform their owner about the presence of a particular person, update an internal database that keeps track of encounters or simply cache the profile for later use.

IV.1.4 Community Language

The most important concept of our system model is the *community language*. A community language defines a way for actors to interact (communicate). It consists of two components:

1. The *community vocabulary*. is a collection of messages types that actors can exchange. A *message type* is a data structure that that provides a template for *messages* that actors can exchange. A message is a single data object that is sent by one actor (the sender) to another actor (the recipient). When an actor sends a

message, it has expectations about how the recipient will respond to the message. Those expectations are not encoded in the message itself.

2. The *community protocol* defines the conversations actors can have. A conversation is a pattern of message exchange that two or more actors are bound to follow in communicating with one another. Typically, actors do not engage in single message exchanges but they have more or less extended conversations, i.e., sequences of message exchanges with a definite beginning and end. A community protocol can be thought of as formalizing *rituals* that actors can engage in

A conversation can contain two types of interactions:

1. *Agent-Agent interactions*: agents communicate by sending messages over a wireless link.

2. *Human-agent interactions*: users interact with their agents, for example in order to query the agent for information or to modify the agent's behavior. Agents interact with users, for example in order to inform the user about the status of ongoing operations or about the presence of nearby users.

Agent-agent interactions can only occur during an encounter, because they are dependent upon a direct wireless link between the users' wearable computers and because we assume the use of short-range wireless networks. Human-agent interactions can occur before, during or after an encounter.

The scope of a community language is shown in Figure 8.

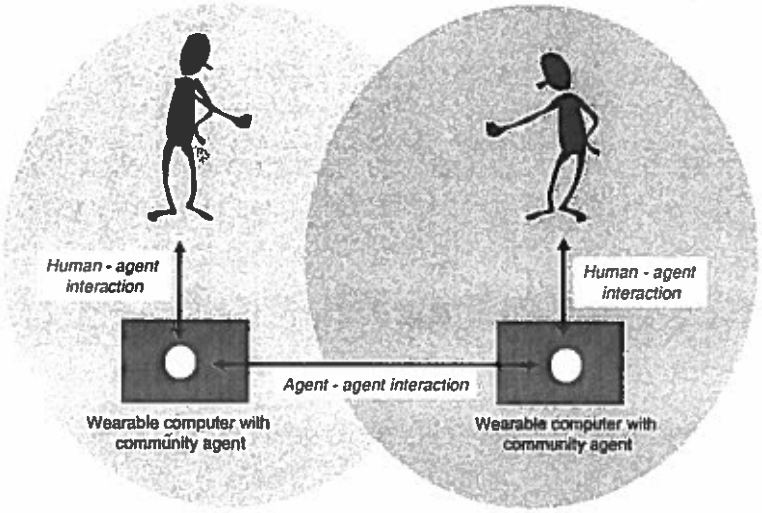


Figure 8. Scope of a Community Protocol

Human-human interactions happen outside of the scope of the wearable community system. Direct exchanges between users are thus not part of a conversation and are not covered by the community language. A community protocol creates rather than constrain social protocols.

In groupware, there has been a longstanding controversy whether social protocols should be determined only by the users, only by the software, or somewhere in between. We believe that social protocols should not be determined by software, but nevertheless should be reflected in the design.

A community language shares some similarities with a communication protocol such as TCP/IP or an agent communication language (ACL) such as KQML (Genesereth and Ketchpel 1994; Finin et al. 1994) or the Interagent Communication Language (ICL) (Martin et al. 1999). First, it differs from both in that it regulates communication between four entities, two of which are computational and two of which are human users. Second, it differs from a communication protocol in that it does not deal with the mechanism of communication but with its content, specifying the vocabulary as well as the structure of interactions. Finally, it differs from an agent

communication language in that it is not concerned with providing a semantic foundation for communication but only with the structure.

Examples of community languages and protocols are proximity-based awareness of other users, 'word of mouth' message passing, and ad hoc voting in temporarily connected groups – all these create opportunities for more additional social interaction rather than narrowing the space of interactions.

Table 7. Main Concepts of the Wearable Community System Model

Community Agent	A software application running on a wearable computer that connects a user to a community
Encounter	A situation in which two or more people are in close physical proximity and their respective agents are aware of each other and able to communicate
User Profile	A typed data item that defines the identity of a user within a community
Message	A typed data item exchanged by actors
Conversation	A sequence of message exchanges among actors
Community Vocabulary	A collection of message types that actors of a community can exchange
Community Protocol	A set of rules that defines the valid conversation within a community
Community Language	Community vocabulary + community protocol
Wearable Community Application	community agent + community language + user profile

IV.1.5 Summary

The conceptual model as defined above enables us to formalize the notion of a wearable community. From a system point of view, a wearable community is a collection of users plus the collection of the users' community agents plus a community language. Within a wearable community, each user has exactly one community agent. The community language describes the interactions between users and their agents and between agents. The main concepts of the system model are summarized in Table 7.

In the following we will introduce a design language for specifying the key aspects of a wearable community and outline a development process that defines the activities from initial design to implementation of wearable community software.

IV.2 Design Language

IV.2.1 Purpose

The wearable community design language (WCDL) is a semi-formal notation for specifying the key aspects of the design of wearable community software. The WCDL provides the modeling language for:

- Scenarios
- User profile templates
- Community vocabularies
- Community protocols

IV.2.2 Scenarios

A *scenario* summarizes the essential elements of a community from the users' perspective. Each scenario is described with the following pieces of information:

- NAME (unique name of community)
- PURPOSE (short description of the purpose of the community)
- POPULATION (short description of user population as it relates to purpose of community)
- DEVICE (short description of required device capabilities)
- BEFORE (short description of actions performed by individual users before an encounter)
- ENCOUNTER (short description of interactions that take place during an encounter of two community members)
- AFTER (short description of actions performed by individual users after an encounter)

Example

Let's look at a concrete example. *Genie* is a simple wearable community system whose purpose it is to improve the knowledge exchange within a dispersed group of wearable computer users, for example employees working in a particular office building or students attending the same college. The goal of Genie is to make the combined expertise of the group available to every group member. Some people of the group might be experts on soccer while other might by hobby gardeners with an extensive knowledge on horticulture. If one member of the group needs to find the answer to a particular question (such as "Where do I find x?", "How do I do y?"), it is often not immediately obvious who is able to answer the question in a given situation. Thus Genie members use their WPAN-equipped wearable computers to automatically find fellow community members who are willing and able to answer particular questions.

Figure 9. Genie wearable community (Step 1)

This is done in the following way: each user defines a set of questions which are stored by the user's Genie community agent. Whenever two or more wearable users meet, their agents exchange their users' questions. This may happen during informal chance encounters occurring normally throughout a day (for example, during coffee break, lunch, or an official company meeting). After receiving a question an agent alerts its user and displays the question on the wearable computer's display, allowing the user to indicate if he or she knows the answer and is willing to talk to the person looking for an answer. The response is relayed back in real-time from agent to agent and eventually to the user (Figure 9). After a successful exchange among the user's community agents, the users can approach each other and discuss the topic of interest in a personal conversation (Figure 10). In order to facilitate contact agents also exchange personal user information including the users' names and photos (photos facilitate contact by people who do not know each other by name).

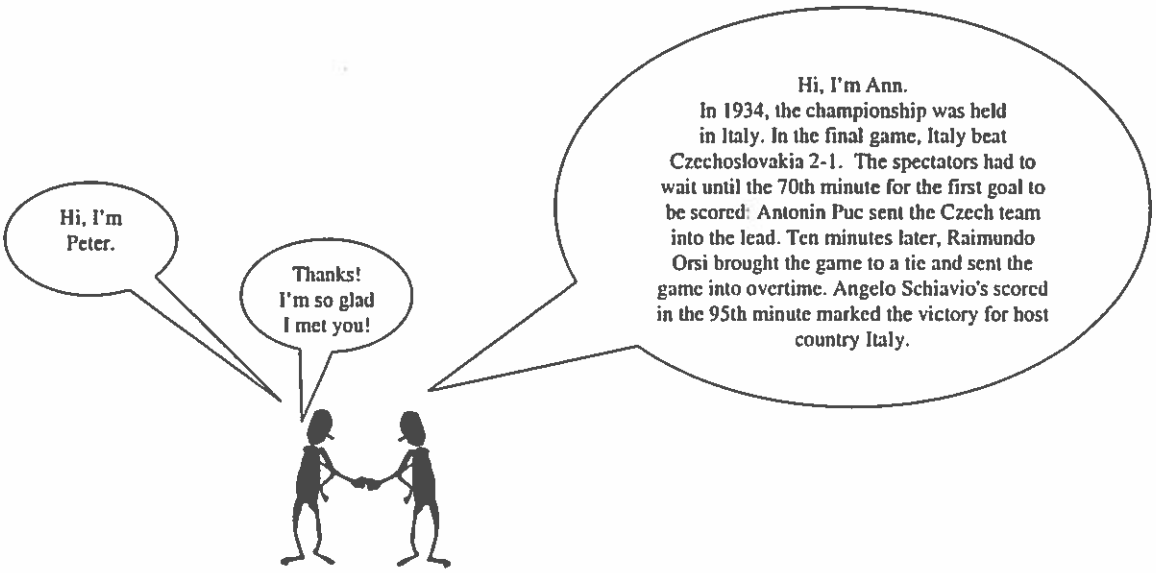


Figure 10. Genie wearable community (Step 2)

The scenario description for the Genie is shown in Table 8

Table 8. Genie scenario

Name	Genie
Purpose	To improve knowledge exchange within a group of mobile people by enabling them to find people who are willing and able to answer particular questions
Population	University students of a specific course OR employees of a company OR members of a project team OR a group of friends
Device	Device must allow users to input questions as text or speech and to indicate that they are willing to talk to the person who poses a question (via a button press or a verbal command)
Before	Users define questions and input them into their computers where they are stored persistently
Encounter	Agents exchange questions and pose the other user's question to their own user. Users who know the answer and are willing to talk to the first user, indicate this willingness to their computer. This information is relayed back to the original user via both user agents together with contact information (name and picture) which allows both users to find each other
After	-

IV.2.3 Profile Templates

The profile template of a community is described by a list of attribute descriptors each of which contains the following fields:

- Attribute Name
- Implementation Type
- Description

The syntax of the attribute name and attribute type is defined as follows:

```
<attribute_name>       ::=   <attribute_name>*                  |
                               A | B | … | Z | 0 | … | 9 | _ | -

<implementation_type> ::=     <primitive_type>                  |
                            [ <primitive_type> ]          |
                              <implementation_type>*          |
                              <implementation_type>+          |
                              <primitive_type> ; <implementation_type>

<primitive_type> ::=        built-in type
```

An implementation type can either be a primitive type, an optional primitive type (indicated by square brackets), a possibly empty list of implementation types (indicated by *), a non-empty list of implementation types (indicated by +) or a sequence of types (a list of types separated by ;).

A primitive type specifies the implementation and refers to the name of programming language-dependent variable type. For example, the name of a Java class or built-in type is a valid primitive type.

Example

The profile template for the Genie community is shown in Table 9.

Table 9. Genie profile template

Attribute Name	Implementation Type	Description
Name	String	A real or assumed name of the user, e.g. "Ann"
Subject	String*	A possibly empty list of subject fields the user is willing and able to answer questions about. For example: "soccer", "architecture"

In the Genie community, members disclose two pieces of personal information: their name and a list of subjects they feel knowledgeable about and are willing to answer questions. The name, which is mandatory and of type String, need not be the user's real name, but might be an assumed name similar to screen names people use in online chat rooms. The subject is represented as a string. In the example, Ann's subject of expertise is soccer.

IV.2.4 Community Vocabulary

A community language consists of two components: the vocabulary and the protocol. A vocabulary is a list of message types each of which is described by three fields:

- Message Type Name
- Description
- Implementation

The specification of the implementation is a list of attribute type specifications as described in Chapter IV.2.3.

Example

The vocabulary of the Genie Community contains three types: question, positive and negative. Their definition is shown in Table 10.

Table 10. Genie vocabulary

Message Type	Description	Implementation		
		Attribute Name	Implementation Type	Description
Question	A message of this type contains the subject of the question and the text of the question.	Subject	String*	The subject field this question refers to. For example "soccer" or "architecture"
		Text	String	The actual text of the question. For example "Who won the 1936 soccer world cup?"
Positive	A message of this type indicates that a user does know the answer and is willing to be contacted	void (a message of this type does not carry any data)		
Negative	A message of this type indicates that a user does not know the answer to a question.	void (a message of this type does not carry any data)		

IV.2.5 Community Protocol

A community protocol defines a set of allowed conversation each of which is a finite sequence of message exchanges. Protocols are specified by *community protocol diagrams* (CPD). A CDP models two aspects of a protocol:

- The states a conversation can be in ("not yet started", "finished", ...)
- The flow of messages during a conversation.

A CDP emphasizes message sequences, i.e. the time ordering of messages.

Notation

A CDP is a graph with the following elements:

- *A state* is a conditions in which a conversation can reside during its lifetime (visualized by labeled circles)
- A *transition* is a relationship between two states indicating that the conversation can move from the first state to the second state. A transition is shown as a rectangle connected by solid arrows to the *source state* and the *target state*. Transitions are labeled with *messages types* indicating a communication event that occurs at the time of the transition. On such a change of state the transition is said to "fire".
- States are organized into *realms*. Each realm is assigned to one actor and indicates responsibility for the continuation of the conversation (visualized by boxes)
- The *initial state* is the default starting place for a conversation (visualized by a black circle)

Example

Figure 11 shows a simple CDP for a hypothetical community protocol (we will discuss the protocol for the Genie community below). The vertical axis of the diagram indicates time, while the horizontal axis indicates the actor which is responsible for the continuation of the conversation. The diagram defines the following sequence of interactions:

The conversation is initiated by Agent A, who sends a message of type t1 to Agent B. Agent B sends a message of type t2 to User B who responds with message of type t3. Agent B then sends a message of type t4 to Agent A who contacts its user by sending message of type t5.

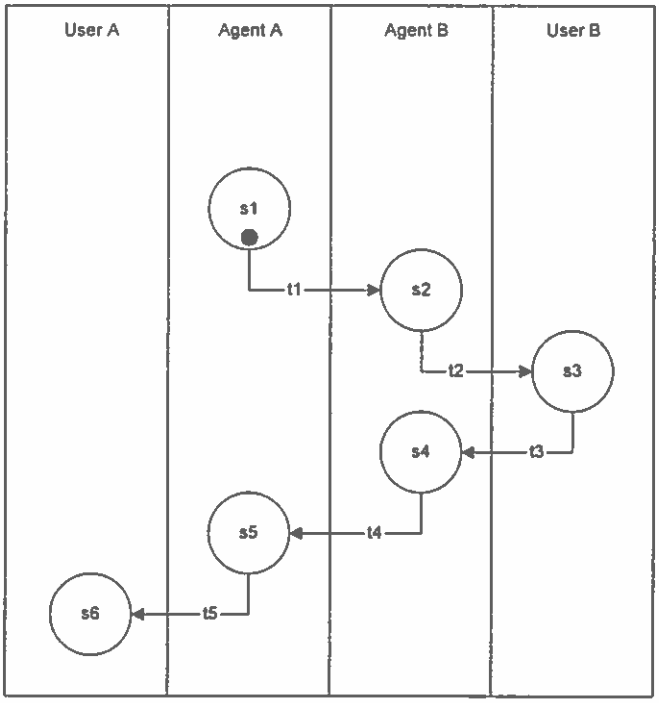


Figure 11. Sample community protocol diagram

A transition simultaneously indicates two related aspects of a protocol:

- Temporal succession: if a transition leads from state s1 to State s2 it means that state s1 strictly occurs before state s2.
- Messaging: if a transition labeled t1 leads from state s1 in realm r1 to state s2 in realm r2 it means that actor r1 sends a message of type t1 to actor r2.

Restrictions

The following additional restrictions apply to CDPs:

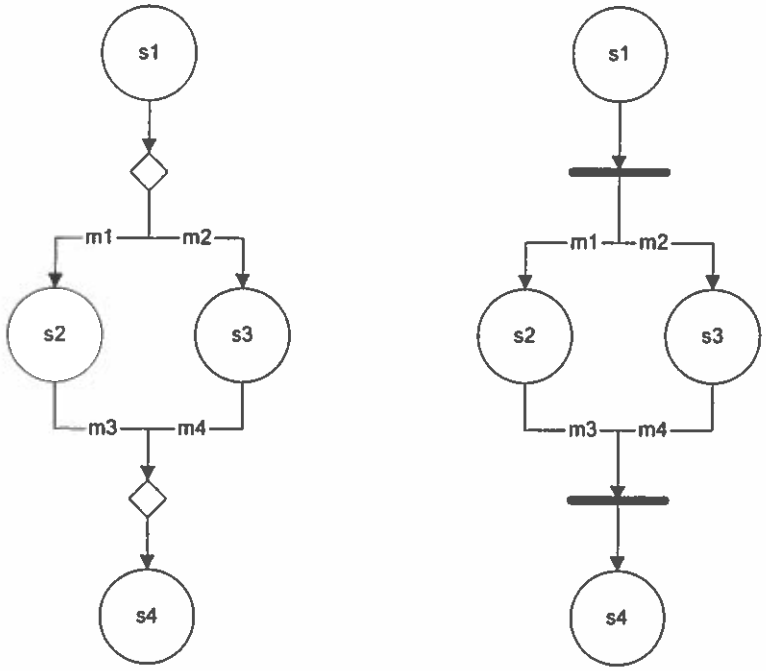
- There are exactly four realms, one for each of the four actors: *User A, Agent A, Agent B* and *User B*.
- Realms are ordered as follows: User A, Agent A, Agent B, User B.
- Transitions may only connect states of consecutive realms (User A <-> Agent A, Agent A <-> Agent B, Agent B <-> User B)
- Each state has only one outgoing transition (but may have multiple incoming transitions)
- There is exactly one start state
- State labels are unique
- Transition labels need not be unique

Complex Transitions

A CDP is intended to model not just one possible conversation, but the space of all allowed conversations at once. Thus, we introduce four types of *complex transitions* (Figure 12):

- A *branch* indicates the splitting into two or more alternate paths. It can be interpreted as logical OR. It is visualized by a triangle with one incoming transition and multiple outgoing arrows. Only outgoing arrows carry labels.

- A *merge* indicates the convening of multiple alternate paths. It is visualized by a triangle with multiple incoming transitions and one outgoing arrows. Only incoming arrows carry labels.
- A *fork* represents the splitting of a single flow of control into two or more concurrent flows of control. It can be interpreted as logical AND. It is visualized by a horizontal bar with one incoming transition and multiple outgoing arrows. Only outgoing arrows carry labels.
- A *join* represents the synchronization of two or more flows of control into one sequential flow of control. It has multiple incoming transitions and one outgoing arrows. It is visualized by a horizontal bar with multiple incoming transitions and one outgoing arrows. Only incoming arrows carry labels.



a. branch / merge **b. fork / join**

Figure 12. Complex transitions

A complex transition is enabled when the source states is occupied. After a complex transition fires its destination state is occupied.

Example

The protocol diagram for the Genie community is shown in Figure 13. It captures two possible conversations:

The exchange is initiated by Agent A, who sends user A's question to Agent B (s1 -> s2). Agent B forwards the question to User B (s2 -> s3). If user B does not know the answer she indicates this by sending a 'no_answer' message to her agent (s3 -> s4). Agent B relays this negative result to Agent A (s4 -> s5). As User A should not be interrupted with a negative response, no further message is send to User A and the conversation has come to an end. If, however, User B knows the answer to User A' question, then she sends an 'answer' message to her Agent B (s3 -> s6). Agent B then informs Agent A (s6 -> s7) which in turn alerts User A that an answer has been found (s7 -> s8).

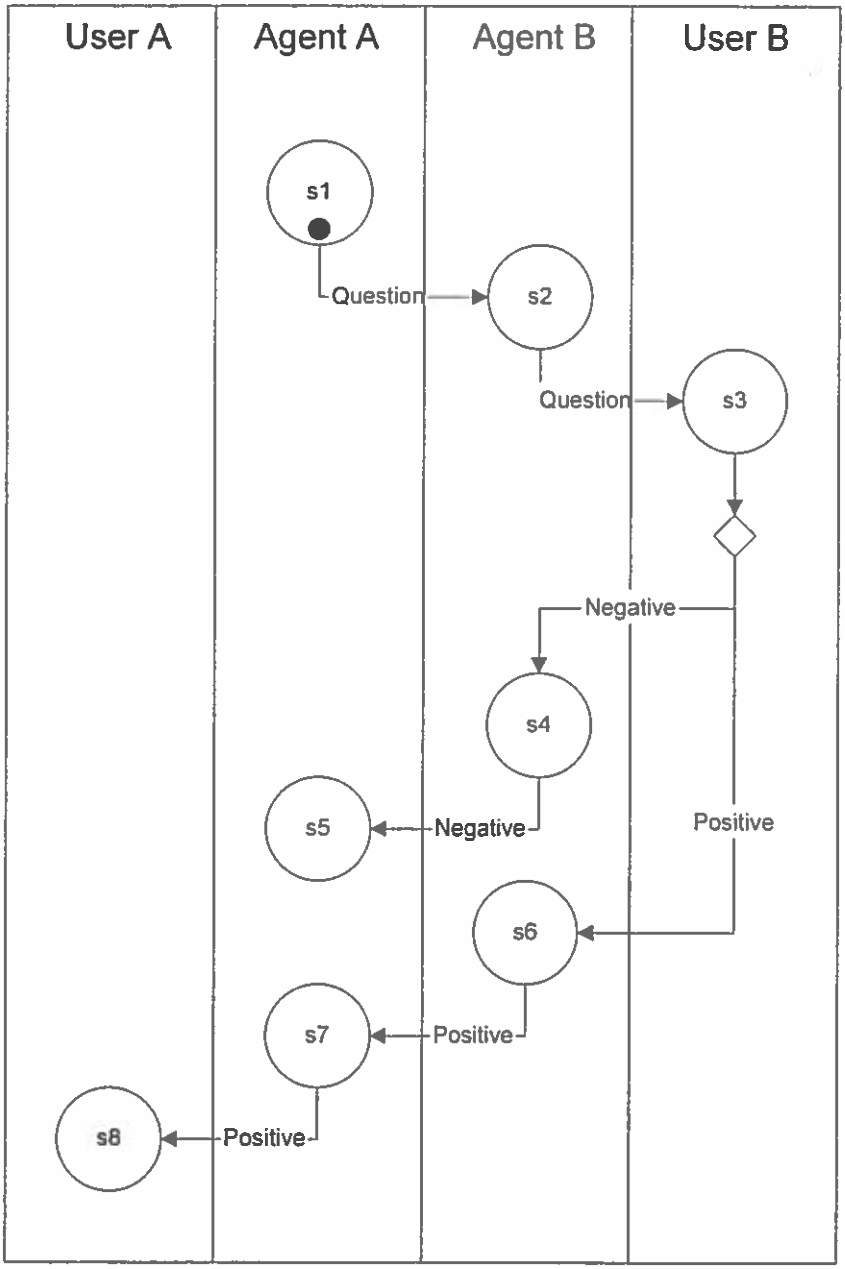


Figure 13. Genie community CPD

IV.3 Development Process

IV.3.1 Purpose

The purpose of the wearable community development process (WCDP) is to guide the activities the lead to the implementation of a wearable community application and its deployment on user devices. The description of the development process is divided into roles and development phases.

IV.3.2 Roles

The development process defines three roles: *designer*, *developer* and *user*.

- Designers define the overall purpose of a wearable community. They are concerned with the social interactions that are to take place among members of a wearable community.
- Developers create the software that runs on the users' wearable computers. Their main concern is the implementation of specific wearable community applications.
- Users are the persons who use wearable computers to engage fellow community members.

At least two different models of community development are possible. In the first model, designers, developers and users are distinct individuals. Designers and developers together develop a wearable community application that is then distributed to users. This model is similar to the traditional software development model. In an alternative model, users are at the same time designers and developers. A group of individuals may collectively specify and design the application, but different individuals may create their own interoperable implementation. This distributed development

model is possible because a community language defines a standard that promotes interoperability.

IV.3.3 Development Phases

The design process guides design and development activities ranging from the initial design to deployment of application software on a user's device. The process is an iterative, cyclic process divided into seven phases: *assessment*, *conceptual design*, *specification*, *implementation*, *seeding*, *dissemination* and *review* (Figure 7). Each phase defines activities performed by users, designers or developers; the outcome of some phase is a specific set of design artifacts. The relationship between roles, phases and artifacts is depicted in Figure 14.

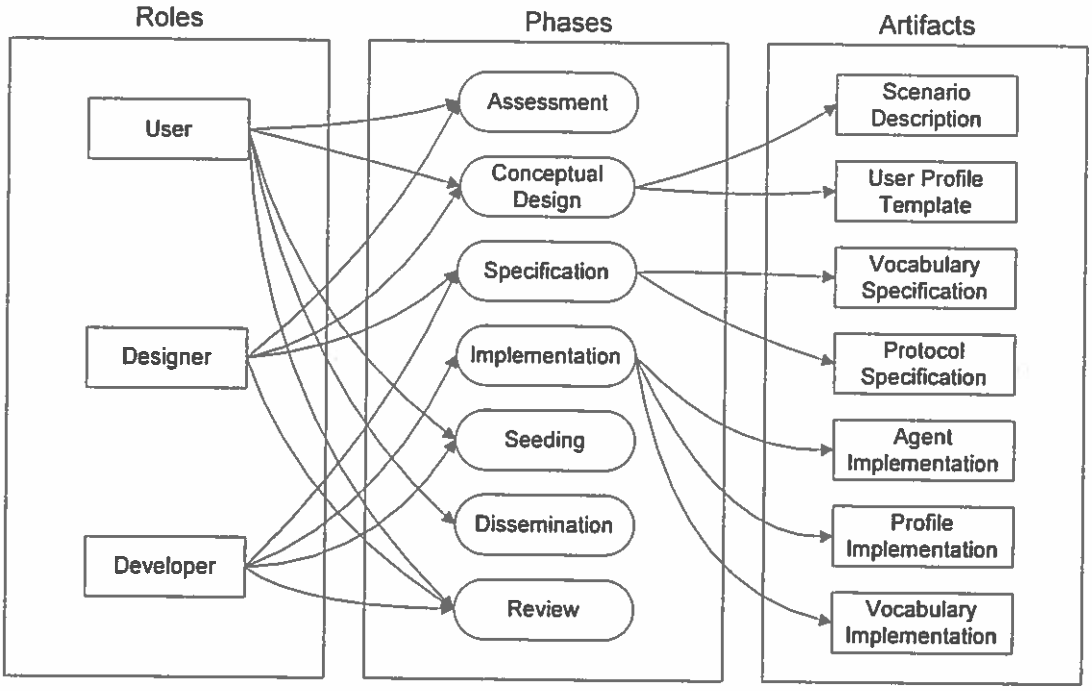


Figure 14. Relationship between WearCoM roles, phases and artifacts

The seven process phases are defined as follows:

Phase 1. Assessment

The assessment phase is performed by designers and developers in cooperation with users. The goal of this phase is to explore the purpose of a wearable community and the user requirements.

The activities of the assessment phase include:

- Identify the user population
- Identify the user needs
- Define the overall purpose of a wearable community
- Identify high-level requirements of the wearable community application

The outcome is an understanding of the goal, purpose and scope of a wearable community and wearable community application.

Phase 2. Conceptual Design

The design phase is performed by designers, possibly in cooperation with users. The goal of this phase is to capture the user experience of interacting within a community. The design activities are:

- Define scenarios that describe all possible interactions within a community
- Identify which personal information people are willing to disclose about themselves.

The artifacts produced by the conceptual design phase are:

- A scenario description (Chapter IV.2.2)
- A user profile template (Chapter IV.2.3)

Phase 3. Specification

The specification phase is performed by designers and developers. The goal of this phase is to specify the interactions that can take place during an encounter. The activities of this phase are:

- Specify the conversation that can occur between actors (users and agents)

The artifacts produced by the specification phase are:

- A community vocabulary (Chapter IV.2.4)
- A community protocol diagram (Chapter IV.2.5)

Phase 4. Implementation

The implementation phase is performed by developers. The goal is to create a software implementation of the community agent, the messages and the user profile. The particular activities include:

- Implement the community agent
- Implement data structures representing the messages
- Implement data structure representing a user profile

Artifacts are:

- Agent implementation (source code and executable code)
- Profile implementation
- Vocabulary implementation (i.e. implementation of message types)

The community protocol is implemented by the community agent.

Phase 5. Seeding

The seeding phase is performed by users or developers. Seeding is the act of distributing the agent to a small number of initial users. This can be done in a public way by creating a community web site that enables users to download the agent or in a more personal way by approaching prospective community members. Seeding

enables individuals to gradually start a new community by involving just a few people. Seeding involves only one essential activity:

- Install the application on a user's device

Phase 6. Dissemination

The dissemination phase is performed by the user. Dissemination is the rumor-like spreading of copies of an agent throughout a user population during face-to-face encounters. An individual who is already member of a particular community transfers a copy of an agent to an individual who is not yet a member, but wishes to become one. Therefore, all members will eventually share the same agent implementation which originates/stems from the original seeders. This phase is essential for a grass-root like growing of an initially small community and represents an important and novel aspect of wearable communities. We envision an ecosystem of wearable community agents, each one created by a different developer, all together competing for the user's attention. Only those agents that are beneficial to users (because it connects them to a useful or fun community) will be propagated to friends, family and colleagues. Agents that are not circulated will eventually die out and disappear. The individual activities of the dissemination phase are:

- Negotiate about access to the agent
- Transfer the agent from one device to another
- Install the agent on the user's device
- Remove an agent from the user's device

Phase 7. Review

The review phase is performed by designers in cooperation with users. The goal is to evaluate:

- the technical quality of a community application prototype
- the success or failure of the design of a community application prototype

- the success or failure of the wearable community created by the application prototype

As discussed in Chapter II, social and technical concerns of wearable communities are difficult to separate. A wearable community application creates a technical foundation from which a community might or not emerge. The success of an application depends on the success of the community it creates. Thus, designers need to simultaneously evaluate technology and social behavior. This can only be done through empirical studies of real users using the technology in a real context.

If the results of the review phase are not satisfactory, a new development cycle needs to be started beginning with either the assessment or the conceptual design phase.

The review phase is one of the least understood phases

IV.4 Tool Support

To help developers use the outlined methodology and to enable rapid development of wearable community applications, infrastructure and tool support is essential. For that purpose, we advocate the use of a *wearable community platform*, a set of software technologies for designing, implementing and deploying wearable community applications. The platform must be distributed in nature and provide high-level abstractions and services for building applications. In particular, it must provide three major support functions:

1. *Development support*: to manage this complexity during the application development the platform must provide architectural and functional abstractions for developers to leverage off when implementing wearable community applications. These abstractions must closely match the concepts introduced by the methodology.

2. *Decentralized system support*: to enable the spontaneous formation of wearable community systems, the platform must provide infrastructure support for presence-awareness and communication.
3. *Runtime support*: to support the dissemination of agents throughout a group of users, the platform needs to enable transmission and guarantee safe execution of community agents.

IV.4.1 Development Support

The primary task the wearable community platform is to simplify the construction of community applications by providing programmers with commonly needed functionality and appropriate abstractions. Thus, the platform should provide an *application framework*, i.e., a set of libraries and application programming interfaces (APIs) for constructing applications. Based on the discussion of the conceptual model at the beginning of this chapter and of the challenges of wearable community systems in Chapter II, we can identify the following required framework features:

- *Identity Management* – enable applications to define and manipulate user identities; in order to address privacy concerns, applications (and thus users) should at all times be in full control over the information given out to other users.
- *Presence awareness* (situational context) – make applications aware of nearby devices and users; to reduce the need for active and repeated inquiry by applications, the framework should support a notification mechanism for "presence events".
- *Human-centered Communication* – enable human-addressable messaging (in addition to device-addressable messaging); the framework should provide flexible addressing modes including direct messaging between two individuals as well as community-wide broadcasting of messages.

- *Relationship Management* (social context) – enable applications (and thus users) to define personal relationship between users. Behavior of applications might change depending on whether a friend, a colleague from work or a total stranger is being encountered.
- *Episodic Memory* (historical context) – make applications aware of the historical context; to reduce the need for applications to record individual events, the framework should automatically record encounters and interactions among users in a persistent database; applications should be able to query this database using user identities and time as search keys.
- *Trust Management* (social context) – enable applications (and thus users) to evaluate the trustworthiness of other users; for example, this may be done on the basis of subjective personal experiences, aggregated experiences of the community as a whole (reputation). Although not all application scenarios and wearable communities require an explicit measure of trust (for example, Genie knowledge community), some applications may benefit from it (for example, the mBazaar "wearable eBay" community).

By addressing these features in a framework, we will enable programmers to more easily and more rapidly build applications. Developers will be able to leverage of the framework, rather than be forced to provide their own custom support in an ad hoc manner.

IV.4.2 Decentralized System Support

A community system is a decentralized system in which hosts interact in a peer-to-peer fashion. Thus, many of the requirements on the system level are similar to those of mobile peer-to-peer systems. They include:

- Presence awareness – dissemination of presence information throughout a system; network-based discovery of nearby devices, resources and individuals through exchange of metainformation.
- Peer-to-peer communication – multi-hop message delivery in highly dynamic ad hoc networks.
- Security – establishing of secure communication links between devices[3].
- Transfer of applications from device to device – as support for the dissemination phase, applications need to be passed from device to device (application transport need not occur during runtime).

IV.4.3 Runtime Support

The dissemination phase requires that applications be transferred between two user's devices in a peer-to-peer manner. In order to ensure safe execution of applications, a secure runtime environment with the following properties is required:

- Verification of application code – to unsure integrity of application code
- Safe execution of application code (sand boxing) – to prevent malicious code from causing harm
- Persistent storage of application state – to enable reliability in the wake of system failures

---

[3] Although security is a requirement for a wearable community platform, we will not address security in this dissertation. Security in dynamic decentralized mobile systems and wireless ad hoc networks is a dissertation topic in itself.

IV.5 Summary

The methodology described in the chapter defines a conceptual model, a design language and a development process for wearable communities. In order to enable an exploratory design approach based on rapid prototyping of wearable community applications and thus wearable communities, we need infrastructure and development support in a variety of areas. To address these needs, we advocate the development of a wearable community platform that focuses on the information needs of applications and provides developers with high-level programming abstractions. The primary component of such a platform is an application framework for building wearable community applications. Although individual solutions exist for the identified requirements there is currently no platform that combines the development, system and runtime support required for adequate support of the WearCoM methodology. In the following section, we describe the Proem platform that was designed to remedy this shortcoming.

Chapter V

THE *PROEM* PLATFORM

In this chapter, we present a solution to the development problem in the form of the *Proem* peer-to-peer platform. Proem is a platform for building presence-aware, ad hoc collaborative applications. The term "presence-aware" refers to the use of knowledge about availability and reachability of people while "ad hoc collaborative" refers to the fact that interactions between people are spontaneous and transient. This type of applications increasingly gains importance as personal mobile devices and wireless ad hoc networks become more popular. Wearable community applications are one important example of ad hoc collaborative applications and Proem has been designed to tightly integrate with the WearCoM development methodology. It offers an extensive API that provides access to common application functionality and enables developers to rapidly build feature-rich wearable community applications. By providing a common ground for the development and execution of wearable community applications, the Proem platform serves as an effective catalyst for wearable communities.

The main components of the platform are:

1. The Peerlet application framework, a collection of libraries and APIs for the rapid development of presence-aware, ad hoc collaborative applications.
2. The Proem Runtime System, a software environment for hosting and executing applications built with the Proem application framework.

3. The Proem protocols, a set of peer-to-peer protocols that define the way in which Proem peers communicate and cooperate over the network.

Proem is implemented in the Java programming language and runs on a variety of mobile and wearable computers including PocketPC-based handheld devices like the Compaq iPAQ series. The runtime and application framework consist of 135 Java classes that have a total size of 75KB (250KB source code). Proem is independent of network transport protocols and can be implemented on top of TCP/IP, HTTP, Bluetooth, and many other protocols. This means that applications built with Proem function in the same fashion when the system is expanded to a new networking environment or to a new class of devices, as long as the Proem runtime system is ported to the new operating environment.

In this Chapter, we describe the overall architecture of Proem and discuss the runtime system and the communication protocols. In Chapter V, we describe the application framework and discuss how the platform integrates with the WearCoM development methodology. In Chapter VI, we discuss case studies of applications built on top of Proem.

V.1 Architecture

V.1.1 Proem Network

Proem is an infrastructure for building decentralized peer-to-peer systems. It is based on a family of peer-to-peer protocols that define the way in which autonomous hosts, called *peers*, communicate and cooperate over a network. A collection of cooperating Proem peers is called a *Proem network* (Figure 15). It builds, on the application level, a virtual network with its own routing mechanisms on top of an existing network infrastructure. Depending on the type of the communication network, the virtual network topology may or may not match the underlying network topology.

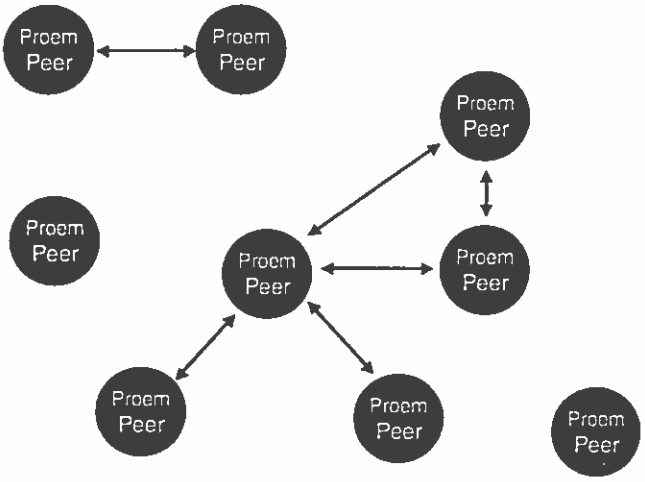


Figure 15. Proem Network

Proem is primarily aimed at wireless proximity networks, i.e. local area and personal area networks. This implies that hosts must be co-located in physical space in order to establish a peer-to-peer relationship.

In a Proem network, each peer operates independently and asynchronously of any other peer. Yet peers may choose to cooperate for a variety of purposes. On the most fundamental level, they may act as mobile routers and cooperatively propagate messages across the network. Cooperation in a Proem network is entirely voluntary and there is no mechanism to enforce cooperation.

A Proem network is a dynamic, self-organizing, ad hoc network. In general, it is partitioned and consists of multiple clusters. Due to the rapid and unpredictable movement of hosts a Proem network does not have a stable topology; communication links are repeatedly created and destroyed as side effect of host movement causing peers to enter and leave each other's transmission range. In a Proem network visibility is limited: not every peer can communicate with every other peer. The set of peers that are visible from a particular peer is called this peer's *horizon* (Figure 16). At a minimum, the horizon includes all immediate neighbors of a peer, i.e. all peers to which

it is connected by a direct network link. In addition, the horizon may include peers that are two or more hops away. Due to the dynamic nature of a Proem network the horizon is constantly changing: peers may unexpectedly and unpredictably enter or leave a peer's horizon at any moment. The horizon is thus mostly a theoretical concept and it is impossible for a peer to know or compute its horizon with certainty.

V.1.2 Key Concepts

The architecture of a Proem network is based on a few fundamental concepts, namely *peer*, *peerlet*, *peerlet engine, protocol and message*.

A *peer* is an autonomous, wireless host such as a mobile or wearable computer. We assume peers to be personal devices that are owned and operated by users. A person can simultaneously use any number of peers, but each peer belongs to only one individual.

Each peer may host one or more peer-to-peer applications, called *peerlets*. Peerlets are network enabled applications whose primary task is to interact with peerlets running on other peers. Each peerlet implements an application-specific *peerlet protocol* and only peerlets that support the same protocol are able to communicate. All peerlet protocols are asynchronous messaging protocols. Proem developers specify and implement peerlet protocols; they are not restricted to a set of built-in protocols.

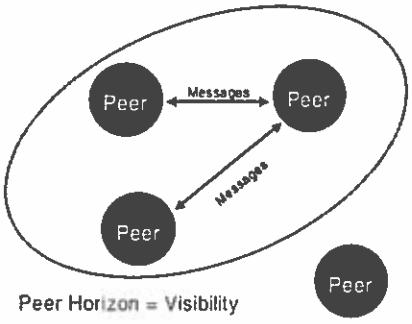


Figure 16. Peer Horizon

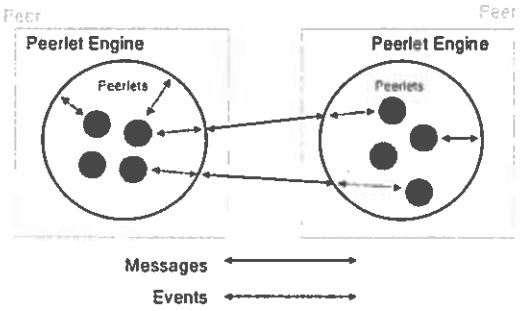


Figure 17. Peers, peerlets, peerlet engines and peerlet protocols

Peerlets are hosted and managed through their lifecycle by the *peerlet engine*. A peerlet engine is the Proem runtime system and is installed on every Proem peer. It provides core services such as discovery, communication and presence awareness. The peerlet engine ensures interoperability and makes sure that peerlets can be installed and safely executed on any peer.

The relationship between peers, Peerlet engine, Peerlets and protocols is depicted in Figure 17.

V.1.3 Assumptions

Proem is an infrastructure for building collaborative peer-to-peer systems, i.e. the ultimate goal is to support interactions and exchanges between people, not just devices. In addition, Proem is primarily aimed at mobile network environments. In its design we made the following assumptions about the operating environment:

Peers are personal mobile or wearable personal devices that are owned and operated by people. They are battery-operated and posses capabilities for data processing, data storage and communication. They are not fault-tolerant and may crash, be turned off or run out of battery power at any time. Hosts do not need to be identical and in general use a variety of hardware and operating system platforms. In contrast to

most mobile computing research, we do not assume that mobile hosts are severely limited in processing power, memory capacity or storage. This reflects the fact that speed of innovation in mobile device technology has increased so dramatically over the last years that we can expect mobile devices soon to become as powerful as high-end workstations were a few years back. For example, we expect that in the near future mobile processors will run at speeds of several gigahertz and mobile storage solutions will provide hundreds of gigabytes of storage, thereby effectively eliminating some of the limitations currently associated with mobile devices. In sum, we assume mobile devices that match or exceed the specifications of today's PalmOS and PocketPC handheld devices.

Short-range and ad hoc wireless communication technology is still relatively new and significant progress can be expected for the future. As we want to accommodate networks based on single channel, broadcast based wireless media (such as 802.11 or IR LAN) and on multi-channel wireless media (such as Bluetooth) we make only few assumptions about the capabilities of the wireless network. The fundamental requirement on the communication technology is that it enables seamless data communication via short-range links both inside and outside of buildings, and allows connecting devices in an ad-hoc fashion (i.e., not requiring setup procedures or user intervention). In particular, we do not make any assumptions about reliability, routing capabilities and link topology of the network:

- The network may not support high-level network protocols such as IP. Communication links may be unreliable, message transmission times may be variable and message may be lost or corrupted during transmission. In particular, there is no guarantee that replies will reach their destination.
- The network may not support multi-hop routing and there is no guarantee that routes between arbitrary hosts exist, even if these hosts are co-located and within transmission range. In addition, routes that do exist may disappear at any moment.

- Communication links may be point-to-point, point-to-multipoint or broadcast-based.
- Communication links may be unidirectional or bi-directional.

Unidirectional links are unknown in wireline networks but are common in wireless networks. Links may be unidirectional due to the *hidden terminal problem* (Tanenbaum 1996) or due to disparity between the transmission power levels of the nodes at either ends of the link. Node A may be able to receive messages from node B as there may very little interference in A's vicinity. However, B may be near an interfering node and, therefore, be unable to receive A's messages. As result, the link between A and B is unidirectional and directed from B to A. Link unidirectionality can be a persistent or transient phenomenon and it is possible for a link to quickly and repeatedly transition between unidirectional and bidirectional state (Prakash 1999).

Wireless communication and networking technologies that are covered by the above definition include existing standards like Bluetooth, IrDA, Home RF, IEEE 802.11 and HIPERLAN, and experimental network technologies like Ultra-wideband radio technology.

V.1.4 Distributed Object Identifiers

Peers manage, communicate about and manipulate distributed objects, which we call *entities*. Within the architecture, entities have first-class status. The *Proem* architecture defines six different *entity types*, namely user, peer, peerlet, peerlet protocol, message type, message and user. Each entity is globally identified by a uniform resource name (URNs) (see Moats 2001), i.e. a persistent and location-independent identifier.

Proem does not provide a built-in naming system and does not guarantee that each distributed entity has a unique ID. A mobile peer-to-peer system that does not make use of central resources and in which peers interact during random encounters

does not have a global state. Thus there is no absolute way to guarantee uniqueness across an entire community that may consist of a large number of peers. However, because of the large identifier space and the high locality of interactions with a Proem network, we do not expect many name clashes.

Proem entities may have multiple URNs. In particular, we do not require that users and peers are known throughout a network by only one name. Multiple names provide for pseudo-anonymity: since there is no central name repository it is impossible to determine whether two different names refer to the same or two different identities. Thus a user who wants to obfuscate his identity may switch to a different URN for himself and his peer. The reason to allow multiple names for users and peers is to allow users to change their identity at will and to make tracking of individual users more difficult. Note, however, that certain collaborative applications depend on stable and unique identities and switching URNs makes these collaborations impossible. It is up to the designer and ultimately the user to make the tradeoff between privacy and functionality.

The syntax of Proem URNs is defined as follows:

<Proem-URN> ::= "urn:proem:" <Proem-Type> ":" <Proem-UID>

<Proem-Type> indicates the type of the entity identified and <Proem-UID> is a numerical identifier that is unique within each type. The syntax definition for the proem type and uid are as follows:

<Proem-Type> ::= "peer" | "user" | "protocol" |
"messagetype" | "message" | "peerlet"

<Proem-UID> ::= <symbol> [1,63<symbol>]

<symbol> ::= <number> | <upper> | <lower>

<number> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<upper> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" |
"I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" |

"Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" |
"Y" | "Z"

<lower> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
"i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
"q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
"y" | "z"

Example

The following strings are valid Proem IDs:
"unr:proem:peer:12345678", "unr:proem:user:00789",
"unr:proem:protocol:genie".

V.2 Peerlet Engine

The peerlet engine is the heart of the Proem platform. It provides a runtime environment for peerlets and is responsible for

- Discovery of peers and users
- Hosting and execution of peerlets
- Enabling communication between peerlets
- Managing peerlet user interfaces

The peerlet engine realizes a Proem virtual machine that ensures interoperability between peers and makes sure that peerlets can be installed and safely executed on any peer. In addition, it provides core services to peer-to-peer application including discovery, presence awareness, and messaging.

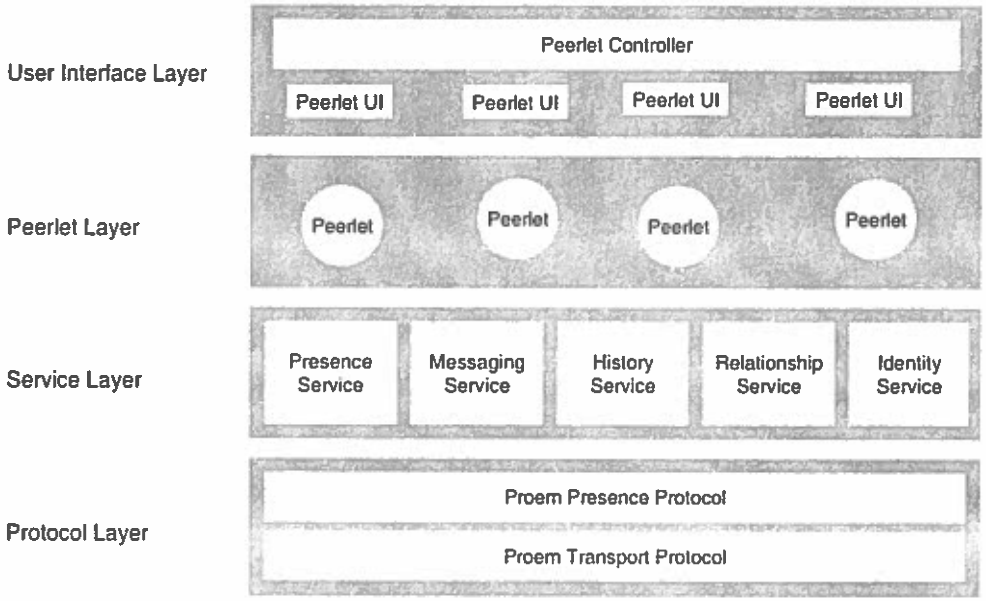


Figure 18. Peerlet Engine Software Architecture

The architecture of the peerlet engine is shown in Figure 18. It is divided into four functional layers:

Protocol Layer

The protocol layer encapsulates essential peer-to-peer computing functions. Its task is to connect peers in a Proem network and to enable cooperation between them. It is implemented as a protocol stack:

- The Proem Transport Protocol is the common messaging protocol for all Proem peers. It hides the underlying network by creating a virtual network view. Similar to the Internet Protocol (IP), the Proem Transport Protocol is an unreliable, best-effort connectionless protocol. It assumes little from the underlying network mechanisms, only that messages will "probably" (best-effort) be transported to the addressed peer.

- The Proem Presence Protocol is implemented on top of the Proem Transport Protocol. It is used by to advertise and discover information about peers and users in a Proem network. As such it is the mechanism for realizing presence awareness.

  The Proem protocols are discussed in more detail in Chapter V.3.

Service Layer

The Service Layer implements a set of common application services. The most important services are the *identity service*, *presence service*, *communication service* and *history service* (searching and indexing, directory, storage, file sharing, distributed file systems, resource aggregation and renting, protocol translation, authentication, and PKI services). The primary goal of this layer is to provide developers with the right abstractions for easy development of wearable community applications.

The service layer provides a set of common application services. These include the *identity service*, *presence service*, *messaging service*, *history service* and *relationship service*. The primary goal of this layer is to provide developers with the right abstractions and a complete set of services for developing presence-aware ad hoc collaborative applications.

- The identity service provides an API for specifying a user's public identity in the form of a public *profile*. A profile is a data structure similar to an electronic business card which is used to advertise a user within a network. A user can define one profile for each peerlet.
- The presence service provides information about nearby users. Applications can query the service and get the profiles of all users who are reachable across the network.
- The history service realizes an episodic memory and provides information about past encounters and interactions. For example, applications can ask "When did I

last encounter user x?" The service automatically captures relevant information about encounters thus simplifying the implementation of applications.

- The messaging service provides a message-based, network-independent communication API that is used to send messages to peerlets on remote peers.
- The relationship service enables applications to define relationships that exist between the user and other individuals. These relationships are stored persistently and are available even after a restart of the peerlet engine.

The functionality of these services and their APIs are described in Chapter VI.5.

Peerlet Layer

This peerlet layer represents the actual execution environment for peerlets. It is responsible for storage of peerlet binaries and the management of peerlets throughout their lifecycle. The development of peerlets and their lifecycle are discussed in Chapter VI.2.

User Interface Layer

The user interface layer manages the interaction users and peerlets. Its main component is the peerlet controller, a simple graphical user interface management system. It displays the user interfaces of individual peerlets and allows users to switch between running peerlets.

In the following two sections we describe the network and user interface layers in more detail. The peerlet and service layers are presented in Chapter VI.

V.3 Proem Protocols

The Proem architecture defines two communication protocols, the *Proem Transport Protocol* and the *Proem Presence Protocol*. These protocols are the foundations for realizing presence-awareness ad hoc interactions of Proem-based devices.

- The Proem Transport Protocol (PTP) is an unreliable, connectionless, asynchronous messaging protocol that defines a common communication substrate for Proem peers. All communication among peers is based on this protocol. PTP is designed as lightweight XML-based messaging protocol. Its design makes minimal assumptions about the underlying communication infrastructures and can easily be implemented on top of a variety of mobile network protocols. It is a connectionless, asynchronous messaging protocol and supports point-to-point and multi-hop messaging.
- The Proem Presence Protocol (PDP) is used to announce and discover peers and users within a Proem network. PDP uses the PTP as messaging layer.

Both protocols must be supported by Proem peer. The definition and use of these protocols guarantees interoperability between different implementations of the *Proem* platform.

V.3.1 Proem Transport Protocol (PTP)

The Proem Transport Protocol is the common messaging protocol for all Proem peers. It hides the underlying network protocol by creating a virtual network view and enables universal understanding of how to perform data exchanges between peers. Similar to the Internet Protocol (IP), the Proem Transport Protocol is an unreliable,

best-effort connectionless protocol. It assumes little from the underlying network mechanisms, only that messages will "probably" (best-effort) be transported to the addressed peer. As result of using a common transport protocol Proem peers can be implemented in any programming language and communication among Proem peers can occur over a wide variety of network technologies.

Proem Transport Protocol (PTP) is a connectionless asynchronous messaging protocol. Data is passed from peer to peer in one atomic unit, called a message. Peers communicate by exchanging one-way messages. The same messages may be sent or received more than once during the course of a protocol exchange. When an unreliable transport protocol is used messages may be delivered more than once, may not arrive at all, or may arrive in a different order than sent. In addition, message delivery is not guaranteed: a sent message may or may not reach its destination. The simplicity of the protocols implies that no protocol states are required to be maintained at either the sending or receiving end.

Addressing

Addressing is based on unique *Peer IDs*. A Peer ID is transport independent and uniquely identifies a peer, even if it uses different lower level network addresses.

Messages

The information transmitted between peers is packaged as XML-encoded messages. Messages are fundamentally one-way transmissions from a sender to a receiver. A Proem message is an XML document that consists of a header, delivery hints and a body:

- The header contains a collection of XML elements indication among other things the source and destination of a messages.

- The delivery hints are a collection of XML elements that define how a message should be delivered to its destination. Delivery hints are interpreted by intermediate peers that forward messages.
- The body is an application specific container of data intended for the application on the destination host.

The header is mandatory, while delivery hints and message body are optional. The XML representation of a message looks as follows:

```
<?xml version="1.0" >
<Header>
...
</Header>
<Hints>
...
</Hints>
<Body>
...
</Body>
```

Message Header

The message header contains the following elements:

Element	Occurrence	Type	Purpose
<Message-id>	1	Proem ID	A unique message id
<Source-id>	1	Proem ID	The id of the peer where this message originated
<Destination-id>	1	Proem ID	The id of the destination peer
<Application-id>	1	Proem ID	The id of the application this message belongs to
<Message-type>	1	Proem ID	The type of this message

- The message id is a (semi-unique) identifier for this message instance.
- The source id indicates the peer from which this message originates. Because messages may be forwarded, this is not necessarily the sender.
- The destination id indicates the peer to which this message should be delivered to
- The application id indicates the application this message belongs to. It is used to determine to which peerlet application a message should be dispatched to once it reaches its destination.
- The message type identifies the type of the message as defined by the peerlet protocol.

Delivery Hints

The delivery hints consists of three components which are discussed below.

<TTL>	1, optional	Integer	Indicates the message's time-to-live
<Hop-count>	1, optional	Integer	Indicates the current hop count of this message
<Expires>	1, optional	Date	Indicates expiration time of this message (used in connection with broadcast or communitycast)

Example

A complete message might look as follows:

```
<?xml version="1.0">
<Header>
        <Message-id>urn:proem:message:568767</Message-id>
        <Source-id>urn:proem:peer:0123487</Source-id>
        <Destination-id>urn:proem:peer:0123487</Destination-id>
        <Protocol-id>urn:proem:appplication:6565675 </Protocol-id>
        <Message-type>urn:proem:application:6565675:2</Message-type>
</Header>
 <Hints>
        <TTL>3</TTL>
        <Hop-count>0<Hop-count>
</Hints>
<Body>
        -
</Body>
```

Cooperative Message Delivery

Message delivery from the source to the destination is done cooperatively, that is with the help of intermediate peers. If a message destination is one hop away, it is delivered directly to the destination peer. If that is not the case, the message is forwarded to peers that are within reach of the sender. To that effect, messages may carry a set of *delivery hints* that indicate how messages should be handled by intermediate peers. The propagation of messages that cannot directly be delivered to their destination is controlled by two delivery hints. The time-to-live element (TTL) indicates how often a message should be forwarded. If a TTL is used, the hop-count element must be present as well. The hop count indicates the number of times a message has been forwarded. Each time a message is forwarded, the TTL is decreased by one and the hop-count is increased by one. A message with a TTL of 0 must not be forwarded.

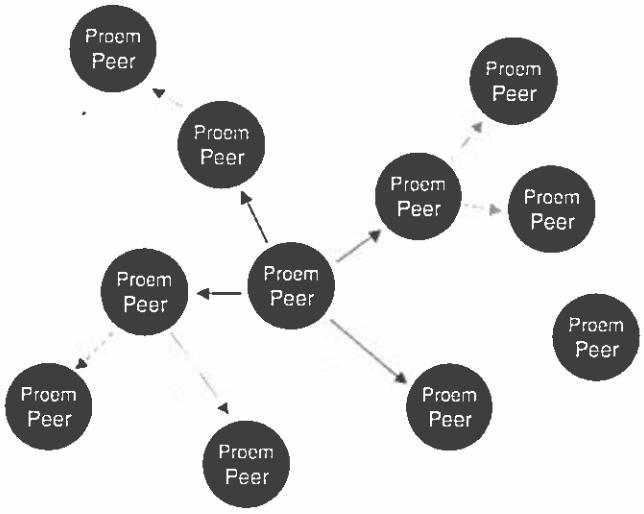


Figure 19. Cooperative Message Delivery

In a Proem network cooperation is voluntary. No peer is required to forward messages, but may do so at its own discretion. A peer may decide to drop a message that is not addressed to itself. Similarly, delivery hints are mere hints: there is no mechanism to guarantee that peers interpret them correctly and act accordingly. In particular, intermediate peers can choose to ignore delivery hints entirely. This behavior is a direct consequence of the fact that peers are independent and may act only in their own (or their users') best interest. Because of bandwidth, energy or privacy concerns, peers might refuse to carry network traffic that does not concern them. Thus, there is no guarantee that messages will be delivered beyond a peer's horizon. In the worst case, a peer might only be able to see and communicate with its direct neighbors.

Avoiding Transmission Loops

The network flooding algorithm used for delivering messages can result in a peer receiving the same message more than once from two different peers. Such a situation indicates a transmission loop. In order to prevent loops intermediate peers, i.e. peers that forward messages, must maintain a cache holding the ids of messages that they have received in the past. If a received message is already in the cache, it must be dropped. The cache size can be fairly small because due to the high locality of a Proem

network loops, if they exist, are small and duplicated messages should arrive "fairly soon" after the first arrival.

Retransmissions

It is valid for a message to be sent more than once. If a peer suspects that a message might not have been received and needs to be retransmitted it may do so. However, because of the way message are handled the message-id of the original message and retransmitted message must be different.

Communitycast and Broadcast

In some situations it is necessary to send the same message to multiple destinations, for example if one wants to disseminate a piece of information throughout a network. Thus, Proem supports three addressing modes:

- Unicast: this is the default addressing mode. Such a message is destined for exactly one peer.
- Broadcast: a broadcast message is intended to be delivered to every peer in the network.
- Communitycast: a communitycast message is intended to be delivered to each peer that supports a specified application.

A broadcast message is identified by using the special Proem id proem:peer:broadcast as destination. A communitycast address is identified by using an application id as destination. For example, the destination proem:application;029809 indicates that this message should be delivered to all peers supporting application 029809.

The intention of the broadcast and multicast addressing modes is that a message be delivered to as many qualifying peers as possible. Network flooding as described above is not appropriate as delivery method: first, flooding would lead to congestion because traffic is no longer local. Second, Proem networks are typically divided into

many small clusters. The reach of flooding is limited to a single cluster and messages thus would not be delivered to peers in a cluster different from the one the sending peer is in.

Thus communitycast and broadcast are implemented by periodically sending the message to all currently connected neighbors. The transmission rates used by peers are independent: each peer can decide how often a message should be repeated. Each community or broadcast message carries an expiration date (delivery hint 'expires') which indicates when retransmission should be stopped.

Implementation

The implementation of the Proem Transport protocol may vary depending on the capabilities of the underlying network. In particular, message delivery is different whether the network supports one-to-one, one-to-many, broadcast or multicast. In the current IP-based implementation, messages to direct neighbors are transmitted via IP unicast while broadcast messages are transmitted via IP multicast.

The current implementation of the Proem Transport Protocol does not support message routing along a particular path, but uses a controlled form of network flooding. If source and destination peer are connected by a direct link the source peer sends the message to the destination. If source and destination are not directly connected, the source peer may forward its message to all peers it is connected to and ask them to forward them on its behalf. Intermediate peers may, but are not required to forward messages until a message finally reaches its destination. This effectively floods the network. The reason for using this approach is twofold:

- Proem networks are dynamic ad hoc networks with potentially rapidly changing topologies. In such a network, the cost of calculating routes can be prohibitive.
- Most or all communication within a Proem network is local with source and destination being only a few hops away. Thus flooding does not incur a large overhead.

V.3.2 Proem Presence Protocol (PPP)

The Proem Presence Protocol defines a decentralized discovery mechanisms for peers and users, i.e. the process of advertising metadata describing peers and users and allowing for their discovery.

One of the most important goals of the Proem platform is to realize proximity awareness. Proximity awareness, in contrast to location awareness, describes a system's ability to determine the presence or absence of entities (users and devices) within the immediate physical space surrounding the system. For wearable communities we are interested in two form of proximity awareness: awareness of people and awareness of devices. Proximity awareness of people can also be described as proximity-based presence awareness.

Proximity awareness can be realized in at least two ways:

- Sensor based: Using dedicated sensor technology such as RFID tags and readers
- Network based: deriving locality information from topology information of communication networks.

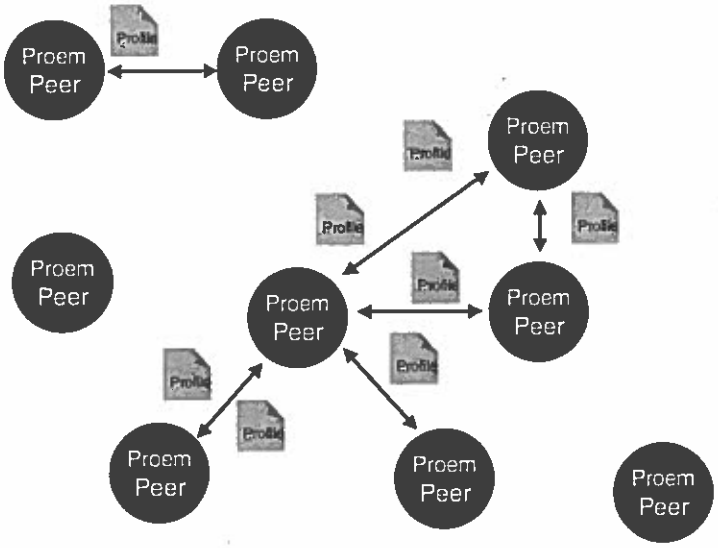


Figure 20. Distribution of Profiles in the Proem Presence Protocol

A network-based approach has the advantage that it is essentially for free as it does not require additional hardware components but instead relies on existing communication infrastructure. The disadvantage of this approach is that in most cases the topology of communication networks is independent of spatial arrangements of communication nodes. This is not true, however, for short-range communication technologies. Here, the existence of a direct link between two nodes implies their physical nearness (the converse is not necessarily true, however).

Proem is agnostic about the method for determining proximity: the presence service API is technology-independent and the presence service can be implemented in various ways.

Discovery

The *Proem Presence Protocol* defines a decentralized discovery mechanisms for Proem peers and users. It defines message formats and a process of advertising metainformation throughout a network and allowing for its discovery.

The basic data unit of the presence protocol is the *profile*. A *user profile* contains a description of a user's identity with regard to a particular peerlet protocol. A *peer profile* is a list of user profiles and describes the protocols a peer supports and the associated user identities.

The presence protocol supports two discovery modes:

- Push: in this mode each peer periodically broadcasts its peer profile. The broadcast is done uncoordinated and asynchronously. The rates with which peers send out broadcasts are independent and may vary widely among peers.
- Pull: in this mode a peer periodically broadcasts a *query*. A query is a request to peers to broadcast their peer profiles. Upon receiving a query peers may, but are not required to, respond by sending a peer profile.

Push mode is the default discovery mode. It allows peers (and ultimately users) to control and adapt the broadcast rate. A low rate may be used to minimize bandwidth

usage or energy consumption associated with sending of messages; a high rate may be used to guarantee discovery of fast moving peers.

The simple push mode has the problem that it provides information about the fact that two peers are within transmission range, but not when they are leaving transmission range. A peer profile indicates the presence of a peer at the particular moment when it is sent and received, but it doesn't say anything about the time immediately following the transmission. Because of the movement of peers communication links are transient and short-lived. Two peers that were visible to each other at one moment may be far apart only a few seconds later. In order to allow peers to predict if they have lost contact with another peer, each peer profile contains a number indication the expected future time interval between two consecutive transmissions (EFTI = expected future transmission interval). A value of 10000 indicates that the sending peer promises to send the next message no later than 10 seconds from now. This information enables the receiving peer to predict when it should receive subsequent messages. If within the specified time interval it does not receive another message, it can assume that it has lost contact with the sending peer. This may either be because the sender has traveled out of transmission range or it was shut down.

Peers can change their EFTI at will. A long EFTI may be used if (1) energy and bandwidth conversation is the primary goal (2) the peer is stationary and visibility relationships are somewhat stable (3) it is not important to discover each and every single peer. The decision about the most appropriate EFTI cannot be made by the system alone, but must be determined in connection with the user's intentions and preferences. At present, the Proem system uses a fixed EFTI that can be adjusted manually by the user.

The pull mode can be used in situations where it is crucial to have the most accurate information. If a peer's EFTI is very long (> 1 minute), it is difficult for other peers to make accurate predictions. Thus if a peer needs to know for sure which peers

and users are in its immediate vicinity it can switch to pull mode and actively query its environment.

Messages

The presence protocol utilizes the PTP transport protocol to route messages.

PeerProfile

The peer profile message contains three elements:

- The id of the peer that is described by this profile (represented as URN)
- An unsigned number indicating the EFTI (in seconds).
- A list of user profiles

Element	Occurrence	Type	Purpose
<peer-id>	1	Proem ID	ID of the peer whose identity is described by this message
<update-interval>	1	Unsigned int	The approximate expected time interval between two identity messages originating from the same peer
<user-profile>	*	User profile	A profile describing the user's identity

A user profile describes a user's identity with regard to one application. A user may use different identities for each application. The peer profile message contains one profile per application.

A user profile contains three elements:

- The user id represented as URN
- The application id this profile relates to (represented as URN)
- An application specific attribute block containing attributes describing the user.

Element	Occurrence	Type	Purpose
<user-id>	1	Proem ID	ID of the peer whose identity is described by this message
<application-id>	1	Proem ID	ID of the application this profile is associated with
<attributes>	*		Application specific attributes

Example

A complete identity message might look as follows:

```
<?xml version="1.0">
<Header>
        …
        <application-id>urn:proem:protocol:0</application-id>
        <message-type>urn:proem:protocol:0:0</message-type>.
        …
</Header>
<Body>
        <peer-profile>
          <peer-id>urn:proem:peer:12345</peer-id>
          <efti>20</efti>
          <user-profile>
            <user-id>urn:proem:user:0123487</user-id>
            <application-id>urn:proem:protocol:0987</protocol-id>
            <attributes>
                 <name>Peter Pan</name>
                 <age>29</age>
              </attributes>
          </user-profile>
        </peer-profile >
</Body>
```

Query

The query message contains two elements:

- The id of the peer from which this request originates (represented as URN)
- A list of protocol ids

Element	Occurrence	Type	Purpose
<peer-id>	1	Proem ID	The id of the peer from which this request originates
<protocols>	*	Proem ID	Indicates the applications the sending peer is interested in.

Example

A query message might look as follows:

```
<?xml version="1.0">
<Header>
        …
        <application-id>urn:proem:protocol:0</application-id>
        <message-type>urn:proem:protocol:0:1</message-type>.
        …
</Header>
<Body>
        <query>
          <peer-id>urn:proem:peer:12345</peer-id>
          <application-id>urn:proem:protocol:234567</application-id>
          <application-id>urn:proem:protocol:19876</application-id>
        </query>
</Body>
```

Delivery Hints

The following delivery hints can be used in connection with the presence protocol:

- TTL: If a peer is only wants to interact with its immediate neighbors, the TTL should be set to 0.
- Expiration Time: this hint may not be used with identity or query messages

V.4 Peerlet Controller

The peerlet controller is a graphical user interface manager for peerlets. In its functionality it is similar to a desktop window manager, yet it has been designed for the requirements of mobile users and mobile and wearable computers. The main goal is simplify the interaction with multiple running applications and to enable mobile users to operate applications while on the go.

The user interface of each peerlet consists of a list of *peerlet panels* where each panel is a fixed-size rectangle displaying common graphical user-interface widgets such as text boxes, menus etc. Panels cannot overlap and only one panel is visible at once (Figure 21 shows the PocketPC implementation).

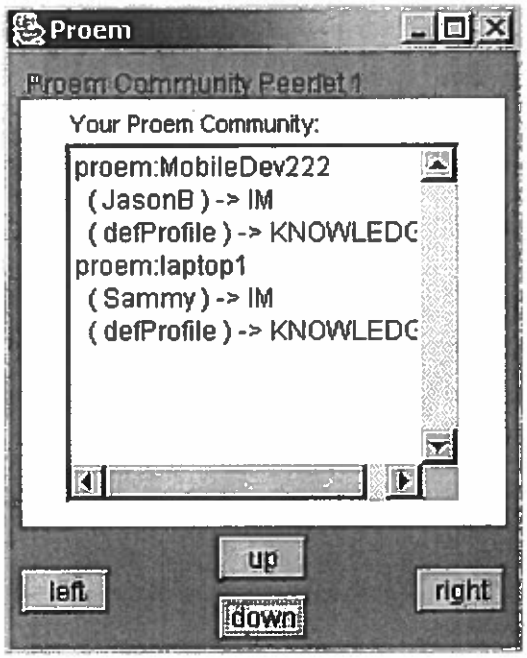


Figure 21. Peerlet controller user interface with one visible panel

The peerlet controller organizes the panels of all peerlets in a 2-dimensional grid (Figure 22). The panels belonging to a peerlet are organized horizontally while peerlets are organized vertically. Four buttons (up, down, left, right) allow the user to scroll through the grid: using the left and right buttons the user can scroll through all panels belonging to one peerlet; using the up and down buttons the user can switch between peerlets. For each peerlet, the peerlet controller remembers the current panel. The current panel is the one that is visible when the user selects a peerlet. If the user switches between peerlets using the up and down buttons, the peerlet controller jumps from one peerlet's current panel to the next peerlet's current panel. When the user moves left or right, the left or right panel becomes the peerlet's current panel.

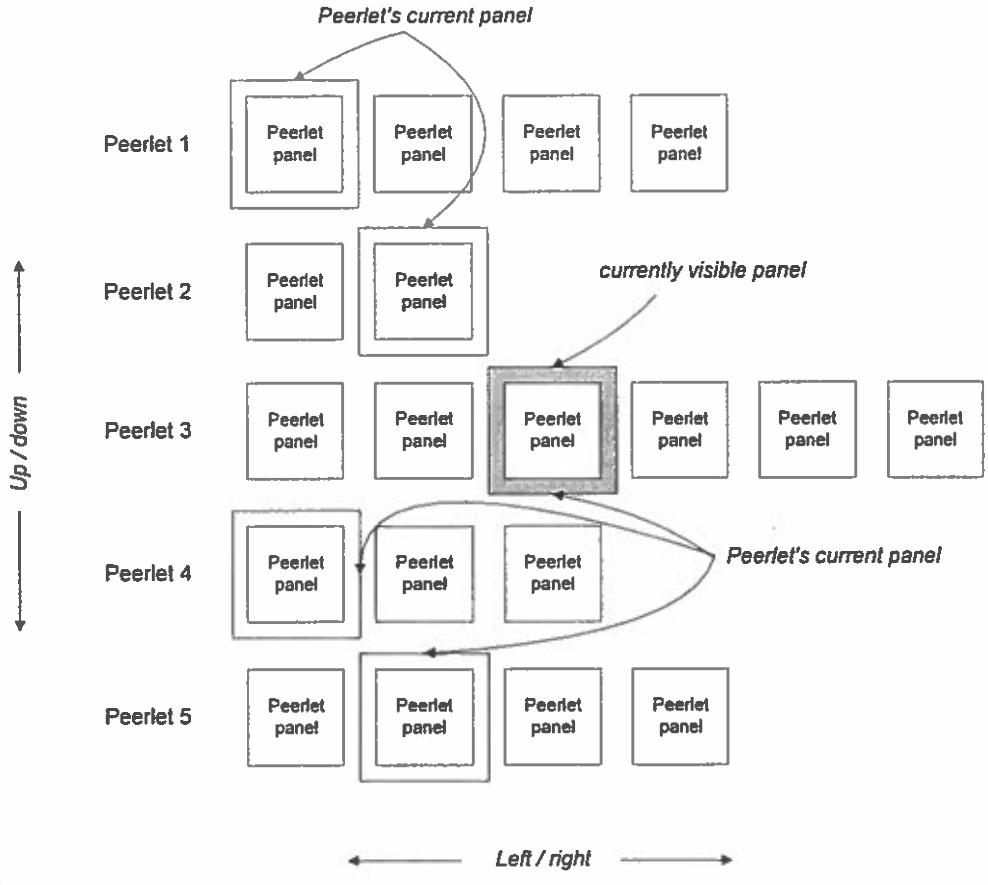


Figure 22. Peerlet panel grid

A peerlet's panel list can be static or dynamic. Peerlets may programmatically create new panels or destroy existing ones. In addition, a peerlet may request from the peerlet manager that a particular panel becomes the current panel and is displayed in the main window. For example, a temporary alert box can be realized by creating a new panel and making it the current panel. After the user has acknowledged the panel, the peerlet may remove the panel from the list.

V.5 Summary

This chapter described the architecture of the Proem peer-to-peer platform. In particular, we described the Proem protocols and the structure of the peerlet engine. The two Proem protocols, the Proem Transport Protocol and the Proem Presence Protocol protocols, define the way in which Proem peers communicate and cooperate over the network. A collection of Proem peers supporting these protocols form a Proem network. The Proem Transport Protocol defines a common, ad hoc communication substrate for Proem peers. It is programming language and network independent and can be implemented on top of a variety of lower-level network protocols. The Proem Presence Protocol defines a standard way for peers to discover each other in the network and to exchange metainformation. Most importantly, it supports the exchange of user profile as required by the WearCom methodology.

A peerlet is a Proem peer-to-peer application. The community agents defined in the WearCoM methodology is implemented as Proem peerlet. Peerlets are hosted by the peerlet engine. The peerlet engine provides runtime services for peerlets for messaging, discovery and presence-awareness and managing of user identities, interaction histories and relationships.

Thus far, the discussion of the Proem platform has focused on architecture, protocols and algorithms. In the next chapter, we will discuss how to implement peerlets and thus wearable community agents using the Peerlet application framework

Chapter VI

THE PEERLET FRAMEWORK

The Peerlet Framework is a Java class library for the rapid development of peerlet applications. It consists of classes and interfaces for the construction of applications and includes a service API providing common functionality for presence aware ad hoc collaborative applications. This chapter presents the Peerlet Framework and illustrates how to implement, deploy and run peerlet applications.

VI.1 Application Model

A peerlet application is separated along functional lines and consists of the following required components:

- A peerlet class (derived from class GenericPeerlet[4])
- A user interface controller class (derived from class GenericPeerletController)
- A set of message classes (derived from MessageBody)
- A user profile (an XML file describing the user's identity)

The peerlet is the main component of an application; it defines the application logic and its communication behavior. The peerlet controller manages the interaction

[4] Class names with the prefix 'Generic' indicate abstract classes that implement an associated interface. For example, proem.GenericPeerlet implements proem.Peerlet. Such classes define default behavior that makes using them more convenient than the associated interface.

with the user; it is responsible for creating the user interface. The message classes define the data that can be exchanged between peerlets on different hosts and between a peerlet and its controller. The user profile is used to advertise a user's identity throughout a Proem network. For easy distribution a peerlet suite may be placed inside a Java Archive (JAR) file.

VI.2 Peerlets

The central abstraction of the Peerlet API is the peerlet. All peerlets implement the Peerlet interface either directly, or more commonly, by extending the class GenericPeerlet that implements the interface.

The peerlet interface is defined as follows:

```
interface Peerlet
{
        /*
         * the getPeerletID method returns the id of this peerlet
         */
        public PeerletID getPeerletID ();

        /*
         * the getProtocolID method returns the protocol supported by this peerlet
         */
        public ProtocolID getProtocolID ();

        /*
         * the init method initializes the Peerlet and puts it into service
         */
        public void init ();

        /*
         * the destroy method is called just before unloads the Peerlet.
         */
        public void destroy ();

        /*
         * the handleProemEvent method gets called when a system event occurs.
         *
         */
        public void handleEvent(proem.Event event);
}
```

All methods defined in the Peerlet interface are called by the Peerlet engine at various stages of the peerlet's lifecycle. They are called in the following sequence:

1. getProtocolID: The peerlet engine constructs the peerlet and calls the getProtocolID methods to determine which protocol this peerlet implements[5]. If the peerlet engine already hosts a peerlet that implements the same protocol, the start-up process is terminated.
2. init: The calling of this method signals to the peerlet that it has entered an active state. This allows the peerlet to initialize data structures.
3. handleProemEvent: this method signals an event that should be handled by the peerlet. The method is called exactly once for each event occurrence. Calls to this method may occur concurrently and it is the responsibility of the programmer to make sure that the code is reentrant.
4. destroy: this method signals that the peerlet instance is about to be destroyed. It enables the peerlet to release resources and save persistent state information.

VI.2.1 Event Handling

Peerlets receive and react to events generated by the Peerlet engine. The Peerlet Framework defines three event types (Figure 23):

1. *Communication events*: events of this type indicate a message from a remote peer
2. *Discovery events*: events of this type indicate that a new peer has been discovered or that a previously encountered peer has disappeared
3. *User interface events*: events of this type indicate input from the user

Each event type is represented by a Java interface.

---

[5] All ID classes (PeerletID, ProtocolID, UserID, ...) are simple wrapper classes that encapsulate a Java String object.

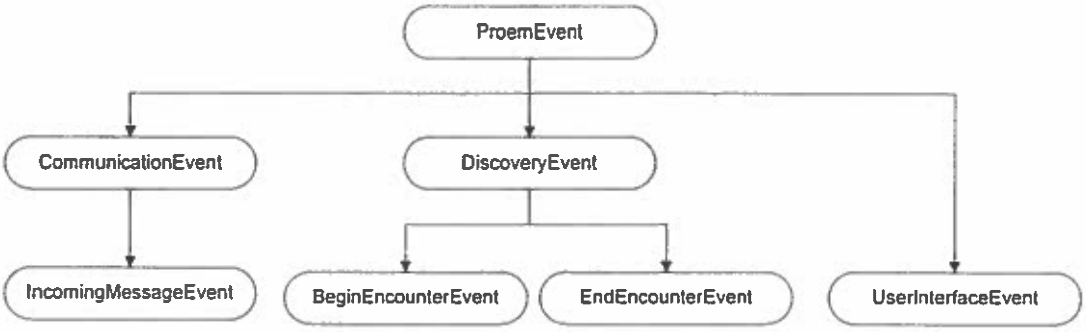


Figure 23. Proem event type hierarchy

The IncomingMessageEvent Interface

An event of type IncomingMessageEvent is fired whenever the peerlet engine receives a message that belongs to the same protocol as supported by the peerlet. The event encapsulates the incoming messages which can be extracted using the getMessage method (the structure of messages and the mechanisms for sending messages are discussed in Chapter VI.5.3).

```
public interface IncomingMessageEvent extends proem.Event
{
        public proem.Message getMessage();
}
```

The UserInterfaceEvent Interface

An event of type UserInterfaceEvent is fired whenever there is new input from the user. The event encapsulates a message object which can be extracted using the getMessage method.

```
public class UserInterfaceEvent implements proem.Event
{
	public proem.Message getMessage()
}
```

The BeginEncounterEvent Interface

An event of type BeginEncounterEvent is fired whenever a new peer has been discovered in the vicinity that supports the same peerlet protocol. The event encapsulates an object of class Encounter which can be extracted using the getEncounter method. An encounter object contains information about the peer and user encountered when the encounter happened (encounter objects are discussed in Chapter VI.5.4)

```
public interface BeginEncounterEvent extends proem.Event
{
	public proem.Encounter getEncounter()
}
```

The EndEncounterEvent Interface

An event of type EndEncounterEvent is fired whenever a previously encountered peer disappears i.e. is no longer visible. The event encapsulates an object of class Encounter which can be extracted using the getEncounter method.

```
public class EndEncounterEvent implements proem.Event
{
	public proem.Encounter getEncounter()
}
```

VI.2.2 Example

The following code segment illustrates the basic structure of event handling code in a peerlet.

```
public void handleEvent(proem.Event event)
{
        if (event instanceof BeginEncounterEvent)
        {
          Encounter encounter = ((BeginEncounterEvent)event).getEncounter();
          UserProfile profile = encounter.getProfile();
          UserID user = profile.getUserID();
          // action 1
        }
        else if (event instanceof EndEncounterEvent)
        {
          Encounter encounter = ((BeginEncounterEvent)event).getEncounter();
          UserProfile profile = encounter.getProfile();
          UserID user = profile.getUserID();
          // action 2

        }
        else if (event instanceof IncomingMessageEvent)
        {
          Message in = ((IncomingMessageEvent).event).getMessage();
          if (in.getBody() instanceof A)
          {
              //action 3
          }
          else if (in.getBody() instanceof B)
          {
              //action 4
          }
        }
        else if (event instanceof UserInterfaceEvent)
        {
          Message in = ((UserInterfaceEvent).event).getMessage();
          if (in.getBody() instanceof C)
          {
              //action 5
          }
          else if (in.getBody() instanceof D)
          {
              //action 6
          }
        }
}
```

The handleProemEvent consists of a sequence of if-then-else statements differentiating between four cases. If the event indicates an encounter (BeginEncounterEvent), the peerlet extracts the encounter object encapsulated in the event; it then extracts the user's profile from the encounter and the user's id from the profile. Knowing the user's identity the peerlet may now perform any action such as sending a message or alerting the user (action 1). The code is almost identical if the

event indicates the end of an encounter (EndEncounterEvent) instead of the beginning. Of course, the actions performed are different (action 2).

If the event indicates a message (IncomingMessageEvent), the peerlet first extracts the message object from the event and may then, depending on the type of the message, perform any number of actions (action3 and 4). User interactions are handled in exactly the same way as messages received from remote peers (UserInterfaceEvent). All user input is represented and send to a peerlet in form of a message which can be extracted from the event object. The message type, which indicates the nature of the user input, can be used to decide which action should be performed (action 5 and 6).

VI.3 Peerlet User Interfaces

The Peerlet Framework strictly separates the user interface code from the main application logic. The peerlet implements the application logic while the *peerlet controller* implements the user interface. The controller is responsible for creating the user interface, displaying of information and handling of user interactions. The interface between peerlet and controller is message based and hides the details of the user interface implementation. In particular, a peerlet is unaware of the user interface technology. A controller may implement a graphical or speech-based user interface and the peerlet code need not be changed.

The user interface component of a peerlet application implements the PeerletController interface which is almost identical to the peerlet interface (it is missing the getProtocolID method).

```
interface PeerletController
{
        /*
         * returns the id of the peerlet this controller belongs to
         */
        public PeerletID getPeerletID ();

        /*
         * the init method initializes the controller
         */
        public void init ();

        /*
         * the destroy method is called just before unloads the Peerlet.
         */
        public void destroy ();

        /*
         * the handleEvent method is called to handle messages from the peerlet.
         *
         */
        public void handleEvent(proem.Event event);
}
```

All methods are called by the peerlet engine at various stages of the application lifecycle.

- getPeerletID must return the id of the peerlet this controller belongs to. This information is used by the peerlet engine to match up peerlets and user interface objects.
- The init method is called during initialization and is used to build the concrete user interface.
- The handleEvent method is called to handle messages sent by the peerlet to the user interface. The only event type that controllers must handle is incomingMessageEvent (the sending of messages is discussed in Chapter VI.5.3).
- The destroy method is called by the peerlet engine to signal that the controller instance is about to be destroyed. It enables the controller to release resources and destroy the user interface.

VI.4 Application Lifecycle

Peerlet applications are managed throughout their lifecycle by the peerlet engine which controls how they are loaded, initialized, and how they are taken out of service. Each running application is an isolated process, defined by the peerlet, with possibly multiple threads of execution. Applications running within the same peerlet engine are not aware of other and cannot communicate.

Peerlet applications are continuously running processes within the peerlet engine that are not started and stopped by the user. Instead, applications are loaded and started when the peerlet engine starts and they are terminated when the peerlet engine shuts down (see also Chapter VI.6).

The application lifecycle consists of four distinctive phases:

1. Loading and Instantiation

At system startup the peerlet container loads the peerlet and peerlet controller classes of a peerlet suite (using normal Java class loading facilities) and creates one instance of each class.

2. Initialization

Initialization consists of three steps:

1. The peerlet engine calls the peerlet's and methods. The peerlet engine initializes the peerlet by calling its getPeerletID, getProtocolID and init methods. The last call enables the peerlet to read persistent configuration data, initialize costly resources and perform other one-time activities.
2. The peerlet engine initializes the controller by calling its getPeerletID and init methods. The later enables the controller to create the user interface.

3. Finally, the peerlet engine reads the user profile file and hands it to the discovery manager. The discovery manager then distributes the profile to other peers in order to advertise the user's identity.

3. Event Handling

After an application is initialized it is ready to handle events. Event handling occurs concurrently: the peerlet engine may send an event to a peerlet even if the peerlet has not finished handling the previous event. To handle concurrent events properly the developer of the peerlet must make adequate provisions for concurrent processing.

4. Termination

The peerlet container keeps an application loaded until the engine itself shuts down. The peerlet engine terminates an application in the following way:

- The peerlet engine first calls the destroy method of the peerlet controller.
- It then calls the destroy method of the peerlet to allow the peerlet to release any resources it is using and to save any persistent state. Before the peerlet engine calls the destroy method, it allows any threads that are currently running in the handleProemEvent method of the peerlet to complete execution, or exceed a predefined time limit. Once the destroy method has been called on a peerlet instance, the engine does not route other events to it.
- After the destroy methods complete, the peerlet engine releases the peerlet and controller instance so that they are eligible for garbage collection.

VI.5 Proem Services

The Proem service API is a collection of interface and classes that provide common application functionality. These services are the presence service, messaging service, history service, relationship service, and reputation service.

Each service is defined by a Java interface and implemented by a single global service object. The service objects are accessible through static methods in the Proem system object:

```
class Proem
{

    static public PresenceService getPresenceService();
    static public CommunicationService getCommunicationService();
    static public HistoryService getHistoryService();
    static public RelationshipService getRelationshipService();
    static public ReputationService getReputationService();


}
```

VI.5.1 Identity Service

The identity service enables applications to define and change a user's profile. This profile is used by the discovery mechanism to advertise a user throughout a network.

```
public interface IdentityService
{

    public UserProfile createProfile();
    public void activate(UserProfile profile);
    public UserProfile getActiveProfile(ProtocolID protocol);

}
```

- The createProfile method returns a new empty profile.
- The activateProfile method activates the supplied profile, i.e. it makes it the current profile for the protocol associated with the profile. After activation, the profile will be distributed by the discovery mechanism to other peers.

The UserProfile interface is defined as follows:

```
public interface UserProfile
{
        public UserID getUserID ();
        public void setUserID ();
        public ProtocolID getProtocolID ();
        public void setProtocolID ();
        public void setAttributeValue ( String attributeName, java.io.Serializable
attributeValue );
        public java.io.Serializable getAttributeValue ( String attributeName );
        public void deleteAttribute ( String attributeName );
}
```

The getUserID returns the id of the user described by this profile while the getProtcolID returns the protocol this profile is associated with. Each profile carries a list of attributes which are name-value pairs. The last three methods set and get an attribute value, and delete an attribute entirely.

Example

The following example code segment illustrates how to define and activate a profile.

```
…
IdentityService is = Proem.getIdentityService();
UserProfile profile = is.createProfile();
Profile.setUserID(new UserID("urn:proem:user:01234"));
Profile.setProtocolID(new UserID("urn:proem:protocol:genie"));
Profile.setAttribute("name", "Howard");
is.activateProfile(profile);
…
```

VI.5.2 Presence Service

The presence service provides information about proximate peers and users. It allows peerlets to determine which peers and users are visible and can be communicated with. The service is defined by the PresenceService Interface:

```
public interface PresenceService
{

    public Collection getPeers(ProtcolID protocol);
    public Collection getUsers(ProtcolID protocol);
    public boolean isVisible(PeerID peer);
    public boolean isVisible(UserID user);
    public void forceUpdate(ProtcolID protocol, long wait);
    public UserProfile getUserProfile(UserID user);

}
```

- The getPeers method returns a list of PeerID objects each one identifying one nearby peer. Similarly, getUsers returns a list of UserID objects each one identifying a currently visible user. The result only contains peers and uses related to the specified protocol. The protocol argument might be null in which case all peers or users are returned.
- The overloaded method isVisible enables a peerlet to test whether a particular peer or user is currently visible. It true if the specified peer or user is visible, false otherwise.
- The information maintained by the presence service and returned by these methods is not guaranteed to be up to date. Because of the constant movement of peers visibility can change rapidly. Two peers that can see each other one moment may be too far apart to communication the next moment. The forceUpdate enables peerlets to actively query their environment for the presence of peers supporting a particular protocol. This method forces the discovery mechanism to perform one pull operation, i.e. to send out one single query message. In order to see the effect of this method, a peerlet should call getPeers or getUsers right after a call to forceUpdate. The first parameter indicates the

protocol that peers must support. The second argument indicates how many milliseconds should pass before a call to this method returns.

- The getUserProfile method returns the profile of a user that has been encountered in the past or is currently visible. User profiles are automatically cached by the presence service so that peerlets do not have to do it.

The UserProfile interface is defined as follows:

```
public interface UserProfile
{
        public UserID getUserID ();
        public ProtocolID getProtocolID ();
        public void setAttributeValue ( String name, Serializable value );
        public java.io.Serializable getAttributeValue ( String attributeName );
        public void deleteAttribute ( String attributeName );
}
```

- The getID method returns the id of the entity described by this profile. The remaining three methods set and get an attribute value, and delete an attribute entirely.

Example 1

The following example code segment illustrates how to force an update operation.

```
static ProtocolID pid = new ProtocolID("proem:protocol:example99");
…
PresenceService ps = Proem.getPresenceService();
ps.forceUpdate(pid, 10000); // wait for 10 seconds
Collection peers = ps.getPeers();
…
```

Example 2

The following code segment illustrates how to find and print the name of every currently visible user (assuming the profile of the particular application includes a name attribute represented as String).

```
…
PresenceService ps = Proem.getPresenceService();
Collection users = ps.getAllUsers();
Iterator it = users.iterator();
UserID id = null;
UserProfile profile = null;
while (it.hasNext())
{
        id = it.next();
        ps.getUserProfile(id);
        if (profile != null)
        {
          System.out.println("The user's name is: " + profile.getAttribute("name"));
        }
        it++;
}
…
```

VI.5.3 Messaging Service

The Messaging Service enables peerlets to send messages to remote peers. Messaging is also used for communication with the user interface.

The Service Interface

The service interface defines three basic methods for sending messages, namely send, broadcast and community. The interface does not contain any method for listening for or receiving messages as messages are delivered to peerlets automatically via event notification.

```
interface MessagingService
{

        public DeliveryHint newDeliveryHint( long ttl, Date expires);
        public Message newMessage( MessageTypeID mtype,
                        ProtocolID pid,
                              DeliveryHints hints,
                        MessageBody body);
        public Message newMessage( MessageTypeID mtype,
                        ProtocolID pid,
                              MessageBody body);
        public void send(PeerID destination, Message mes);
        public void send(UserID destination, Message mes);
        public long broadcast(Message mes);
        public long communitycast(ProtocolID destination, Message mes);
        public void cancel(long);
}
```

```
interface Message
{
        public MessageTypeID getMessageType();
        public ProtocolID getProtocolID();

        public MessageBody getBody();
        public DeliveryHints getHints();
        public PeerID getDestination();
        public PeerID getSource();
        public UserID getSender();
        public void setBody();

}
```

```
interface DeliveryHint
{
        public long getTTL ();
        public Date getExpirationDate ();
}
```

- newMessageHint creates a new delivery hint object. The time-to-live indicates how many hops a message should be forwarded, while the expiration time indicates until time a broadcast or communitycast message remains valid.
- newMessage creates a new message object. The programmer must specify the message type, the protocol ID, and a message body. The delivery hints argument is optional and may be null.
- Send initiates the transmission of a unicast message to the destination peer. The ttl delivery hint, if present, indicates that this message should be forwarded from host to host. However, there is no guarantee that receiving peers interpret this hint or act upon it.
- Broadcast sends a message to all peers in a network. If no expiration time is specified in the delivery hint the message is sent immediately and only once. If the delivery hint contains an expiration date and it lies in the future, the message is repeatedly sent until the expiration data has been reached. The returned is a unique number that can be used to stop a broadcast in progress (see Cancel).

- Communitycast sends a message to all peers supporting a particular application. If no expiration time is specified the message is sent immediately and only once. If an expiration date is specified and it lies in the future, the message is repeatedly sent until the expiration data has been reached. The returned is a unique number that can be used to stop a communitycast in progress (see Cancel).

Cancel stops a broadcast and communitycast which is in progress, i.e. has not reached its expiration date.

MessageBody Interface

The message body of a message represents the application specific payload. Any Java object may be used as payload. A class intended as payload must implement the (empty) MessageBody interface.

```
interface MessageBody
{
}
```

Example

The following code segment illustrates how a peerlet can send a message to a peer it just encountered.

```
/*
 * defininition of message body class
 */
class MessageBody_Example implements proem.MessageBody
{
        public   String someText;
        public   MessageBody_Example(String initialText) { someText = initiualText;
};
};


/*
 * peerlet definition (implements protocol proem:protocol:example99)
 */
class Peerlet_Example implements proem.Peerlet
{
        static MessageTypeID mtid = new MessageTypeID("proem:messagetype:01");
        static ProtocolID pid = new ProtocolID("proem:protocol:example99");

        …
        public void handleProemEvent(proem.Event event)
        {
          if (event instanceof BeginEncounterEvent)
          {
            // get id of encountered peer
            PeerID peer = ((BeginEncounterEvent)event).getEncounter().getPeerID();
            // get handle to service object
            CommunicationService cs = Proem.getCommunicationService();
            // build message object
            DeliveryHints hints = cs.newDeliveryHints(2, null);
            MessageBody_Example body = new MessageBody_Example("welcome");
            Message welcome = cs.newMessage(mtid, pid, hints, body );
            // send message
            cs.send(peer, welcome);
          }
          …
        }
        …
}
```

VI.5.4 History Service

The history service provides information about past encounters. It allows peerlets to make decisions based on historical data. For example, a peerlet may act in one way if a peer or user is encountered for the first time ever and a different way if the same peer or user has been encountered numerous times before. Similarly, a peerlet may make different decisions depending on how long ago the last encounter occurred. As another example, a peerlet may choose to interact with a particular peer only once a day and ignore all subsequent encounters.

The history service listens to event generated by the discovery mechanism. It maintains a global persistent database that is automatically updated for each event.

The Encounter Interface

The main concept of the history service is the encounter represented by the encounter interface.

```
interface Encounter
{
        public PeerID getPeerID();
        public UserID getUserID();
        public Date getDate();
}
```

An encounter object is created whenever a BeginEncounterEvent is generated by the discovery mechanism. It contains information about the identity of the involved peer and user and the starting time and date of the encounter.

The history service is defined by the HistoryService interface.

```
interface HistoryService
{
        public Encounter getFirstEncounter(PeerID);
        public Encounter getFirstEncounter(UserID);

        public Encounter getLastEncounter(PeerID);
        public Encounter getLastEncounter(UserID);

        public long getSecondsSinceLastEncounter(PeerID);
        public long getSecondsSinceLastEncounter(UserID);

        public List getAllEncounters(PeerID);
        public List getAllEncounters(UserID);

        public long getNumberOfEncounters(PeerID);
        public long getNumberOfEncounters(UserID);

        public Set getAllPeers();
        public Set getAllUsers();
}
```

- getFirstEncounter returns an encounter object representation the first ever encounter with the specified peer or user.
- getLastEncounter returns an encounter object representation the mot recent encounter with the specified peer or user.
- getSecondsSinceLastEncounter returns the seconds since the last encounter with the specified peer or user.
- getAllEncounters returns an unordered collection of all encounters with the specified peer or user.
- getNumberOfEncounters determines how often the specified peer or user has been encountered.
- Finally, getAllPeers and getAllUsers return a set of ids of all peers and users ever encountered.

The information maintained by the presence service is stored persistently and is available after a restart of the peerlet engine. However, if the amount of data becomes too large to handle it may delete parts of the history. This is done with the following guarantees:

1. The first and last encounters for each peer and user are always preserved.
2. If an encounter that dates from time t_i is deleted, than all encounters with times $t_j < t_i$, $j \neq 0$ are also deleted.
3. In other words, the service may only purge the earliest part of history (excluding the very first encounter with each peer and user), but not never the later or most resent part.

Example

The following example illustrates how information about past encounters can be used to implement different behaviors. In particular, the following code segment sends a message whenever only if the last encounter with the same person occurred more than 24 hours ago.

```
class Peerlet_Example implements proem.Peerlet
{
        static MessageTypeID mtid = new MessageTypeID("proem:messagetype:01");
        static ProtocolID pid = new ProtocolID("proem:protocol:example99");

        …

        public void handleProemEvent(proem.Event event)
        {
          HistoryService hs = Proem.getHistoryervice();
          CommunicationService cs = Proem.getCommunicationService();

          if (event instanceof BeginEncounterEvent)
          {
            UserID user = ((BeginEncounterEvent)event).getEncounter().getUserID();
            PeerID peer = ((BeginEncounterEvent)event).getEncounter().getPeerID();

            if (hs.getSecondsSinceLastEncounter(user) > 24 * 60 * 60)
            {
              // more than 24 hours: send welcome message
              MessageBody_Example body = new MessageBody_Example("welcome");
              Message welcome = cs.newMessage(mtid, pid, null, body );
              cs.send(peer, welcome);
            }
            else
            {
              //less than 24 hours: do nothing
            }
          }
          …
        }
        …
}
```

VI.5.5 Relationship Service

Our interactions with other people differ depending on the type of relationship we have with them. For example, we might be willing to trade MP3 files with almost everyone (ignoring for the moment the legal ramifications), but we might want to share our diary only with our best friends. Similarly, we probably want to receive music recommendations only from people who we know have a good taste in music.

Thus for Proem applications it is crucial to be able to reliably remember certain individuals. The relationship service enables applications to define relationships that exist between the user and other individuals. These relationships are stored persistently and are available even after a restart of the peerlet engine.

The relationship service is defined by the following interface:

```
interface RelationshipService
{
        public void defineRelationship(String name);
        public void removeRelationship(String name);
        public void addUser(String relationship, UserID id);
        public void removeUser(String relationship UserID id);
        public void isMember(String relationship, UserID user);
        public Collection getMembers(String relationship);
        public Collection getRelationships(UserID user)
}
```

A relationship is simply a named collection of users.

- defineRelationship creates a new empty relationship.
- removeRelationship deletes an existing relationship from the database.
- addUser adds a user to a relationship and removeUser removes it.
- isMember enables applications to determine if a particular user is part of a relationship.
- getMembers returns an unordered collection containing all members of a relationship.
- getRelationships returns an unordered collection containing the names of all relationships a user is part of.

Example

The following example illustrates how relationships can be used to change the behavior of a peerlet depending whether a friend or an unknown person is being encountered. In particular, the following code segment sends a message whenever a friend is encountered, but ignores strangers.

```
class Peerlet_Example implements proem.Peerlet
{
        static MessageTypeID mtid = new MessageTypeID("proem:messagetype:01");
        static ProtocolID pid = new ProtocolID("proem:protocol:example99");

          …

        public void init()
        {
          RelationshipService rs = Proem.getRelationship();
          rs.defineRelationship("My Friends");
          rs.addUser("My Friends", new UserID("proem:user:1111");
          rs.addUser("My Friends", new UserID("proem:user:2222");
        }

        public void handleProemEvent(proem.Event event)
        {
          RelationshipService rs = Proem.getRelationship();
          CommunicationService cs = Proem.getCommunicationService();

          if (event instanceof BeginEncounterEvent)
          {
            UserID user = ((BeginEncounterEvent)event).getEncounter().getUserID();
            PeerID peer = ((BeginEncounterEvent)event).getEncounter().getPeerID();

            if (rs.isMember("My Friends", user))
            {
              // this is a friend, send welcome message
              MessageBody_Example body = new MessageBody_Example("welcome");
              Message welcome = cs.newMessage(mtid, pid, null, body );
              cs.send(peer, welcome);
            }
            else
            {
              // ignore unknown user
            }
          }
          …
        }
        …
}
```

VI.6 Deployment and Distribution

VI.6.1 Bundling

A Proem application is bundled and distributed in form of a *peerlet suite*. A Peerlet suite is the collection of classes and resources that are required to run the application. The suite must contain the following items:

- A peerlet class (derived from class GenericPeerlet)
- A peerlet user interface (derived from class GenericPeerletController)
- A set of message classes (derived from MessageBody)
- A user profile (an XML file describing the user's identity)

To facilitate distribution a peerlet suite may be stored in a JAR file.

VI.6.2 Installation

A peerlet suite is installed by placing it in the application directory inside the peerlet engine installation directory of the peerlet engine. The application will automatically be started after a restart of the peerlet engine.

VI.7 Summary

This chapter described the Peerlet Framework and illustrated how to implement, deploy and run peerlet applications. The Proem platform and SDK enable developers to work in a high-level environment that hides the intricacies of the underlying network and device platform and frees developers from dealing with low-level issues unrelated to their application. A set of specially designed services for communication, presence

awareness, history information provide the building blocks for powerful peer-to-peer applications for ad hoc collaboration. Once an application is written, Proem provides a common runtime environment across many makes and models of devices, enabling developers to deploy their applications with ease and without the need of modifications. Developed applications can be shielded from future changes in technology by making necessary modifications once in the peerlet engine.

We can summarize the main features of the peerlet framework from a programmer's point of view as follows:

- *Separation of concerns*: The programming model enforces and simplifies the separation of application logic and user interface code.
- *Simple event-based programming model*: The event mechanism provides a uniform way for handling communication, discovery and user interface events.
- *A rich library of common application functionality*: The service API provides the abstractions and building blocks for the development of presence aware, ad hoc peer-to-peer applications.
- *Network and device independence*: Developers need not be aware of the specifics of the underlying network and device platforms. In particular, they are able to target multiple heterogeneous network technologies and devices platforms.
- *Tight integration with the WearCoM methodology*: Proem directly supports important concepts the WearCoM methodology: user profiles, community protocols (via peerlet protocols), community agents (via peerlets)

In the following chapter, we will describe several case studies that show how the WearCoM methodology and the Proem platform are used to build wearable community applications.

Chapter VII

CASE STUDIES:
BUILDING WEARABLE COMMUNITY APPLICATIONS

In this chapter, we present a number of wearable community applications that have been developed using the WearCoM methodology and implemented on top of the Proem platform. Our goal is to demonstrate how the methodology is used in practice to design and build wearable community applications. In particular, this chapter demonstrates two things.

1. The reduction in the complexity of the development task achieved by the peerlet framework.
2. The close relationship between design and implementation

VII.1 Overview

The wearable community applications presented in this chapter cover a large spectrum of the wearable community application space:

- *FriendFinder*: this application was inspired by Holmquist's notion of an Inter Personal Awareness Device (Holmquist et al. 1999) (see Chapter III.4). The purpose of this application is to determine affinity relationships between users based on recurring encounters.

- *Genie*: this application was introduced in Chapter IV to demonstrate the WearCoM methodology. It is one of the most simple wearable community applications (Kortuem and Segall 2003).
- *PIRATÉ*: this application is a collaborative music guide that enables users to share and collect music recommendations as side effect of encounters (see Chapter II.2.1). *PIRATÉ* was first described in (Kortuem et al. 2001).
- *mBazaar*: a wearable eBay application that facilitates contacts between buyers and sellers of goods and services (see Chapter II.2.1) (Kortuem and Segall 2003)
- *WALID*: this application was introduced in Chapter II.2.1. It represents one of the first examples of a wearable community application and assists users by negotiating the exchange of tasks with the agents of nearby community members. *WALID* was first described in (Kortuem, Schneider, Suruda, Fickas, and Segall 1999).

Each of the five applications highlights a particular specific aspect of the Peerlet API (Table 11).

Table 11. Case Studies

Application	Highlighted API Features
FriendFinder	User profiles, event handling, history service
Genie	User profiles, *simple community protocols*, *messaging service*
PIRATÉ	User profiles, community protocols, messaging service, *history service*, *relationship service*
mBazaar	User profiles, community protocols, messaging service, history service, *communitycasting*
WALID	User profiles, *complex community protocols*, messaging service, history service, relationship service

The main part of this chapter consists of a presentation of the five case studies. The description of the design and implementation of each application is organized as follows:

- *Overview*: this section describes the general application idea
- *Scenario*: this section describes the usage scenario underlying the application
- *Community language*: this section describes the community language, i.e. the community protocol and vocabulary.
- *User profile*: this section describes the personal data disclose themselves.
- *User interface*: this section describes how users interact with the application. In general, we use screen shots to highlight the key aspects of the user interface.
- *Implementation*: this section describes the peerlet implementation the community agent

VII.2 FriendFinder

VII.2.1 Overview

FriendFinder is a simple presence-aware community application inspired by the notion of Inter-Personal Awareness Devices (IPAD) (Holmquist et al. 1999). In contrast to a communication device like the mobile phone, an Inter-Personal Awareness Device facilitates contacts instead of mediating them. The earliest example of an IPAD is the Hummingbird (Holmquist et al. 1999) (see Chapter III.4). The Hummingbird gives members of a group continuous aural and visual indication of which other group members are in the vicinity. A Hummingbird broadcasts a unique signal and receives information about other Hummingbirds in the vicinity. Each Hummingbird displays a

list of currently 'visible' devices. Whenever a Hummingbird discovers a new device in its vicinity, it alerts its user with a subtle audio signal.

The Hummingbirds architecture is simple and effective, but suffers from several limitations. Most importantly, it only supports awareness between members of a closed user group. To provide social awareness between users (as opposed to devices) the Hummingbird architecture uses an implicit static mapping from devices to users. This mapping must be performed independently by each device making it difficult to add new members to a group.

FriendFinder is an extension to the original IPAD idea. Contrary to what the name might suggest, FriendFinder is not a tool for locating one's friends, but for discovering affinity relationships between users that might not be obvious to the users themselves. The idea underlying FriendFinder is that the frequency, with which two users meet is an important indicator for similarities in their taste, interests, preferences, and believes. This idea has subsequently been explored in more depth in the Social Net project (Terry et al. 2002). For example, if two people frequently attend the same events (for example, gallery openings), then this might be an indication that they share an interest in art.

The FriendFinder user profile contains a unique but arbitrary user id, a name and a photo of the user. The photo helps to relate faces in a crowed with names displayed by the application. The FriendFinder application records frequencies of encounters with other people and displays a list of 'suggested friends' ordered by the frequency with which they have been encountered.

VII.2.2 Scenario

The scenario description for the FriendFinder applications is shown in Table 12.

Table 12. FriendFinder Scenario

Name	FriendFinder
Purpose	To increase social awareness by identifying people with whom we might share interests, preferences, or believes based on frequencies of encounters
Population	Students or teenagers
Device	Device must be able to display a list of user names and photos. Device must be able to persistently store a list of encountered people. Users must be able to set up their user profile, preferably by downloading it from a PC.
Before	Users sets up their user profiles
Encounter	Devices exchange user profiles and update list of most frequently encountered users.
After	-

VII.2.3 User Profiles

The user profile template for the FriendFinder application is shown in Table 13.

Table 13. FriendFinder User Profile Template

Attribute Name	Implementation Type	Description
ID	UserID	A semi-unique identifier for the user.
Name	[String]	The user's name (this might be the user's real first name or an assumed name). This entry is optimal.
Photo	[Image]	A photo depicting the user (this allows for easy spotting of the user in a crowed). This entry is optimal.

The user profile consists of user id, user name and a photo. Name and photo are optional.

VII.2.4 Community Language

The FriendFinder application does not use a community language, but solely depends on the exchange of user profiles (which is automatically done by the discovery mechanisms).

VII.2.5 Using the Application

The FriendFinder user interface is shown in Figure 24. It displays a list of users ordered by frequency.

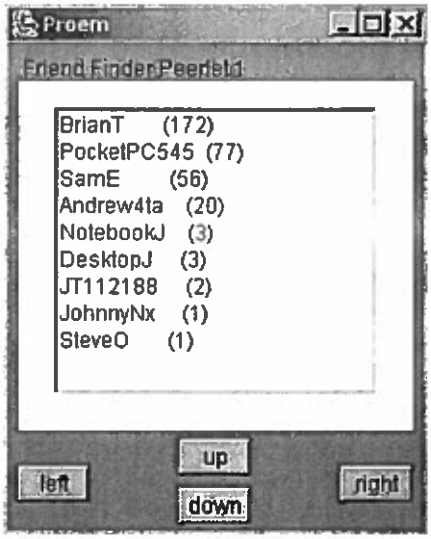


Figure 24. FriendFinder user interface

VII.2.6 Implementation

The FriendFinder application consists of two main Java classes:

- FriendFinderPeerlet is implements the community agent.
- FriendFinderUI displays the user interface.

In the following discussion, we will ignore the user interface details and limit our attention to the application logic. The FriendFinderPeerlet class is shown in Program 1.

Program 1: FriendFinder Peerlet

```
1.  public class FriendFinder_Peerlet extends GenericPeerlet
2.  {
3.      static public PeerletID getPeerletID ()
4.      {
5.        return new PeerletID("proem:peerlet:friendfinder");
6.      }
7.
8.      static public ProtocolID getProtocolID ()
9.      {
10.       return null; // no protocol
11.     }
12.
13.     public void handleProemEvent(proem.Event event)
14.     {
15.       if (event instanceof BeginEncounterEvent)
16.       {
17.         Set users = Proem.getHistoryService().getAllUsers();
18.         uimanager.updateDisplay(users);
19.       }
20.     }
21. }
```

The code of the FriendFinder peerlet consists of only three methods, two of which simply return the peerlet ID (lines 3-6) and protocol ID (lines 8-11). Since the FriendFinder application does not use a community protocol, the peerlet returns null as value for the protocol ID (line 10). The handleProemEvent method (lines 13-20) is called by the peerlet engine with an event of type BeginEncounterEvent whenever the Proem discovery mechanism determines the beginning of a new encounter. This method then calls the HistoryService to get the set of all users that have been encountered so far (line 17) and forwards it to the user interface (line 18). The user interface object sorts the set of previously encountered users and displays their information (code not shown).

VII.2.7 Summary

The FriendFinder application is a simple community application that demonstrates some of the core aspects of peerlet programming. In order to build this application, the programmer needs to understand only a few concepts, namely

- how to implement a peerlet by deriving a new class from the class GenericPeerlet;
- the basics of event handling;
- the functionality provided by the history service API.

As can be seen by the code shown in Program 1, the FriendFinder application is extremely simple. The whole application logic (excluding user-interface code and comments) comprises just 21 lines of code.

The FriendFinder case study is important because it shows how an application described in the literature can be recreated using Proem and how easy it is to extend its functionality without increasing the complexity of the implementation.

VII.3 Genie

VII.3.1 Overview

The second case study is the Genie application introduced in Chapter 3. As we already described the scenario, user profile and community protocol in Chapter IV.2, we will limit our discussion to the user interface and implementation.

VII.3.2 Using the Application

The user-interface of the Genie application consists of two screens. The first screen allows a user to define a question that will be transmitted to other devices during encounters (Figure 25). The second screen displays a question (including subject) received from another device. By pressing the 'Answer' button the user can indicate that he or she knows the answer (Figure 26).

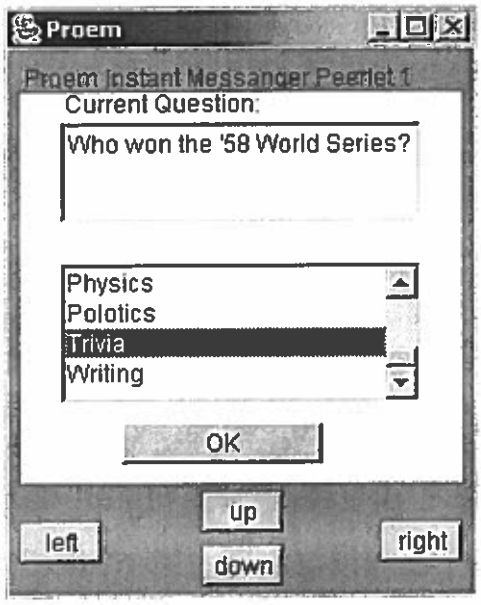


Figure 25. Genie user interface (screen 1).

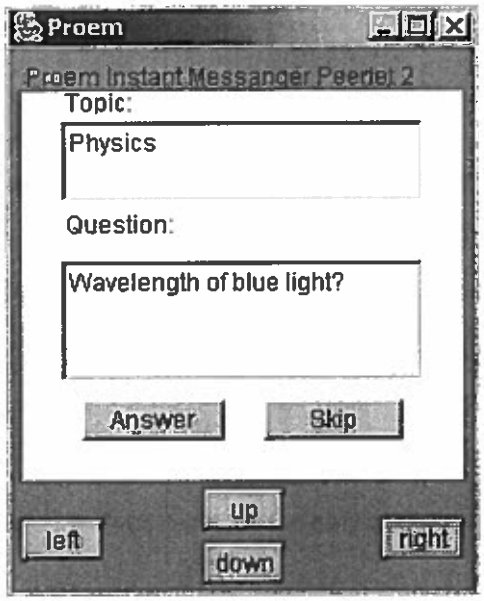


Figure 26. Genie user interface (screen 2)

VII.3.3 Implementation

The Genie application consists of several Java classes:

- Genie_UserProfile implements the user profile (Program 2)
- Genie_Question, Genie_Positive and Genie_Negative are classes representing the three message bodies of the community protocol (Program 2)
- Genie_Peerlet is derived from GenericPeerlet and implements the user agent (Program 3)
- Genie_UI displays the user interface and handles the user interaction (not shown).

In our discussion, we will ignore the implementation of the user interface and limit our attention to the agent implementation.

Program 2. Genie Message and User Profile Classes

```
1.  class Genie_MessageBody_Question implements proem.MessageBody
2.  {
3.      public String subject;
4.      public String text;
5.  };
6.
7.  class Genie_MessageBody_Answer implements proem.MessageBody
8.  {
9.      public String text;
10. };
11.
12. class Genie_MessageBody_No_Answer implements proem.MessageBody
13. {
14. };
15.
16. class Genie_UserProfile extends Proem.GenericUserProfile
17. {
18.     public String name;
19.     public String subject;
20. };
```

Program 3. Genie Peerlet

```
1.  public class Genie_Peerlet implements GenericPeerlet
2.  {
3.      Genie_MessageBody_Question question;           // this user's question
4.      MessagingService ms = Proem.getMessagingService();
5.
6.      static public PeerletID getPeerletID()
7.      {
8.        return new PeerletID("proem:peerlet:genie");
9.      }
10.
11.     static public ProtocolID getProtocolID ()
12.     {
13.       return return new ProtocolID("proem:protocol:genie");
14.     }
15.
16.     public void handleProemEvent(Event event)
17.     {
18.       if (event instanceof BeginEncounterEvent)
19.       {
20.         Encounter encounter = ((EncounterEvent)event).getEncounter();
21.         Genie_UserProfile profile = encounter.getUserProfile();
22.         remote_user = profile.getUserID();
23.         if (profile.getSubject().equals(question.subject))
24.         {
25.           Message mes;
26.           mes = ms.newMessage("proem:messagetype:question",
                                  getProtocolID(),question);
27.           ms.send(remote_user, mes);
28.         }
29.       }
30.
31.       else if (event instanceof IncomingMessageEvent)
32.       {
33.         Proem.Protocol.Message in = event.getMessage();
34.         if (in.getMessageType().equals("proem:messagetype:question"))
35.         {
36.           Genie_UI.send(in);
37.         }
38.         if (in.getMessageType().equals("proem:messagetype:positive"))
39.         {
40.           Genie_UI.send(in);
41.         }
42.         if (in.getMessageType().equals("proem:messagetype:negative"))
43.         {
44.           // do nothing
45.         }
46.       }
47.
48.       else if (event instanceof UserinterfaceEvent)
49.       {
50.         Proem.Protocol.Message in = event.getMessage();
51.         ms.send(in.getSource(), in);
52.       }
53.
54.       else if (event instanceof EndEncounterEvent)
55.       {
56.         // do nothing
57.       }
58.     }
59. }
```

The code of the Genie peerlet consists of a variable declaration section (lines 3-5) and four methods. The variables are:

- a variable for storing the user's question (line 5)
- a variable for holding a reference to the global MessagingService object.

The main component of the peerlet is the handleProemEvent() method (line 16-58). It directly reflects the community protocol as defined by the community protocol diagram in Chapter IV.2.5. The method consists of one large if-statement with four cases for handling events indicating (1) the beginning of an encounter (2) an incoming message and (3) a user interface event and (4) the end of an encounter.

Upon receiving a BeginEncounterEvent, the peerlet constructs a message containing the question and sends it to the encountered user (line 27).

Upon receiving an IncomingMessageEvent, the peerlet forwards the message to the user interface (accessible through the class Genie_UI) unless it is a negative response (line 44).

Upon receiving a UserInterfaceEvent, the peerlet forwards the message to the original sender, i.e. the user who originally sent a question (line 51).

An EndEncounterEvent is ignored by the peerlet (line 56)

VII.3.4 Summary

The Genie application illustrates how agents implement (simple) community protocols. In particular, it demonstrates that the structure of the handleProemEvent method directly reflects the structure of the community protocol. In order to build this application, the programmer needs to understand

- how to implement a peerlet by deriving a new class from the class GenericPeerlet;
- how to handle events;
- how to send messages to either the user interface or a remote agent.

VII.4 PIRATÉ Collaborative Music Guide

PIRATÉ (Kortuem et al. 2001) (see Chapter II.2.1) takes ideas from peer-to-peer file sharing applications like Napster and moves them to the wearable domain. It is a collaborative music guide that enables users to exchange MP3 play lists and music recommendations during brief random encounters.

PIRATÉ can be described as "Napster in reverse". In contrast to MP3 file sharing applications like Napster and Gnutella, PIRATÉ is not a tool for sharing MP3 files, but for discovering music one would like to listen to by trading recommendations. The bandwidth limitations of wireless personal area networks make it unfeasible to send large MP3 files across the network. PIRATÉ circumvents these restrictions by exchanging meta-information about songs instead of the songs themselves.

PIRATÉ employs methods for "collaborative filtering" or "social filtering" of information (Resnick et al. 1994) (Shardanand and Maes 1995) (Breese et al. 1998) for generating recommendations. The main idea of PIRATÉ is to automate the process of "word-of-mouth" by which people recommend music to one another. Faced with a myriad of choices when it comes to deciding which music to listen to, people will often rely on the opinions of others. These others can either be certified experts in the field, i.e. professional music critiques, or members of the social circle, that is friends, family and coworkers.

Collaborative filtering generates recommendations by identifying groups of people with similar interests, that is, it uses a collective method of recommendation. Collaborative or social filtering methods produce recommendations by computing the similarity between a user's music preferences and the ones of other people.

The task of the PIRATÉ community agents is to collect information about other people's music preferences and to generate personalized recommendations. The basic mechanism behind the PIRATÉ collaborative filtering method is the following:

- Each community agent maintains an ordered MP3 play list where each MP3 file is marked by how often the user has listened to a song within the last seven days.
- Agents exchange play lists during encounters thus giving each user access to the music preferences of a potentially large group of people.
- Using a similarity metric, the agent selects a subgroup of people whose preferences are similar to the user's preferences.
- Based on the collective taste of the subgroup, the agent recommends MP3s to the user.

In contrast to Internet-wide collaborative filtering systems like Amazon, PIRATÉ produces localized recommendations that reflect the taste of the user's local social circle. For example, with PIRATÉ students can get music recommendations based on the collective music preferences of the campus population. Such a system allows local bands that are relatively unknown to be included.

The main bottleneck with existing collaborative filtering systems is the collection of preferences. To be reliable, the system needs a very large number of people to express their preferences about a relatively large number of options. Since the system becomes useful only after a "critical mass" of opinions has been collected, people will not be very motivated to express detailed preferences in the beginning stages (e.g. by scoring dozens of music records on a 10 point scale), when the system cannot yet help them. One way to avoid this start-up problem is to collect preferences that are implicit in people's actions (Nichols 1997). For example, people who order books from an Internet bookshop implicitly express their preference for the books they buy over the books they do not buy. Customers who have bought the same book are likely to have similar preferences for other books as well. This principle is applied by the Amazon web bookshop, which for each book offers a list of related books that were bought by the same people. Another method – the on employed by PIRATÉ - uses MP3 play lists to gauge user's music preferences. People who listen to some MP3s more

frequently express their preference for the MP3s they listen to over the MP3s they do not listen to. The list of all MP3s the user has been listening to within the past few days determines a preference function for this user. The preference of each MP3 is equal to the frequency with which the user has listened to the song.

Another novel feature of PIRATÉ is a filter mechanism based on personal relationships. A user might prefer to receive music recommendations only from a few "trusted" sources, i.e. users who are known for their exceptional taste in music. PIRATÉ enables users to set up a list of trusted users and have his or her agent ignore play lists from all other users.

In sum, PIRATÉ differs in the following ways from traditional recommender systems:

- decentralized architecture
- based on localized social groups
- implicit user interface: user is not required to rate songs, exchange occurs automatically
- recommendations are constantly updated
- trusted sources, instead of anonymous recommendations

VII.4.1 Scenario

The scenario description for the PIRATÉ applications is shown in Table 14.

Table 14. PIRATÉ scenario

Name	PIRATÉ
Purpose	To generate personalized music recommendations based on listening behavior of community members.
population	Music lovers, groups of friends, classmates, students, colleagues who frequently listen to music.
Device	Wearable device that can play MP3 music files. Input: MP3 player controls (play, forward, stop, rewind) Output: display recommendation
Before	-
Encounter	Devices exchange play lists and generate personalized recommendations
After	-

VII.4.2 User Profile

PIRATÉ user profiles are simple and only contain the user's URI and a name (Table 15).

Table 15 PIRATÉ user profile

Attribute Name	Implementation Type	Description
Name	[String]	The user name, e.g. the first name or nick name
Favorites	String*	A list of favorite artists or music genres, for example "Blues", "Talking Heads".

VII.4.3 Community Language

Protocol

The PIRATÉ community protocol is depicted in Figure 27. The interactions are rather simple: when an encounter occurs, agents exchange MP3 play lists. The data is used by agents to compute a new set up recommendation which is sent to the users. Although the interactions are symmetric, the recommendations generated by agents are personalized and will differ for each user.

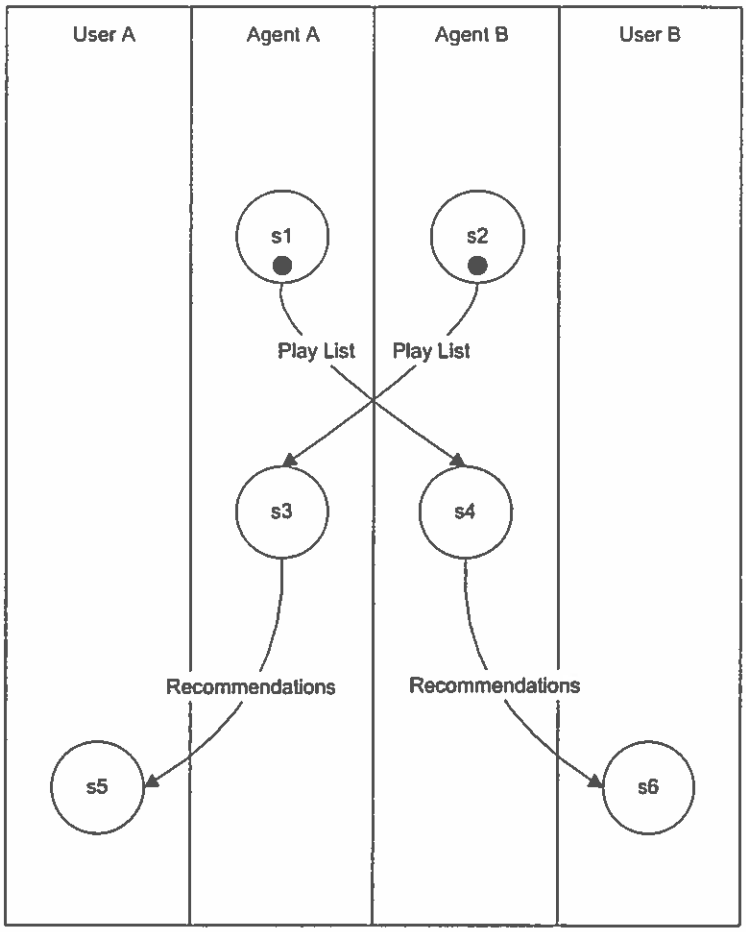


Figure 27. PIRATÉ community protocol

Vocabulary

In PIRATÉ, MP3 files are identified by ID3 tags (ID3 2002). ID3 tags are fix-sized 128-byte data fields embedded in MP3 files that carry information for identifying songs. The ID3 format v1.1 is as follows:

Song title	30 characters
Artist	30 characters
Album	30 characters
Year	4 characters
Comment	30 characters
Genre	1 byte

The genre field holds the index into a predefined table. For example genre 0 represents Blues, 1 represents Classic Rock, 2 represents Country etc. In PIRATÉ, the comment field is used for storing the rating information, i.e. the number of times a song has been listened to. The PIRATÉ vocabulary definition is shown in Table 16. Play lists and recommendations are lists of ID3 tags.

Table 16. PIRATÉ vocabulary

Message Type	Description	Implementation	
		Type	Description
PlayList	A message of this type carries a list of songs, each of which is described by an ID3 tag.	ID3Tag*	A list of ID3 tags each of which represent one song in the play list.
Recommendation	A recommendation is a list of songs each of which is described by an ID3 tag	ID3Tag*	A list of ID3 tags each of which represent one song in the recommendation list.

VII.4.4 Implementation

The PIRATÉ application consists of several Java classes.

- Two message classes (Program 4)
- The user profile class (Program 4)
- A user interface class derived from PeerletUserInterface (not shown)
- The class PIRATE_Peerlet which implements the application logic (Program 5).

Program 4. PIRATÉ message and user profile classes

```
1.  class ID3Tag
2.  {
3.      String title;
4.      String artist;
5.      String album
6.      String year;
7.      String comment; // = number of times this song has been listened to
8.      byte genre;
9.  };
10.
11. class PIRATÉ _PlayList implements proem.MessageBody
12. {
13.     public java.util.List songs;
14. };
15.
16. class PIRATÉ _Recommendation implements proem.MessageBody
17. {
18.     public java.util.List songs;
19. };
20.
21. class Genie_UserProfile extends Proem.GenericUserProfile
22. {
23.     public String name;
24.     public List favorites;
25. };
```

Program 5. PIRATÉ Peerlet Class

```
1.  public class PIRATE_Peerlet extends GenericPeerlet
2.  {
3.      PIRATE_PlayList playlist;              // the user's current play list
4.      PIRATE_Recommednation recommendations; // the user's cur. recommendations
5.      HistoryService      hs = Proem.getHistoryService();
6.      MessagingService    ms = Proem.getMessagingService();
7.      RelationshipService rs = Proem.getRelationshipService();
8.
9.      public PeerletID getPeerletID ()
10.     {
11.       return new PeerletID("proem:peerlet:PIRATÉ ");
12.     }
13.
14.     public ProtocolID getProtocolID ()
15.     {
16.       return new ProtocolID("proem:protocol:PIRATÉ ");
17.     }
18.
19.     public void init()
20.     {
21.       rs.defineRelationship("PIRATÉ ");
22.       rs.addUser("PIRATÉ ", new UserID("proem:user:peter01"));
23.       rs.addUser("PIRATÉ ", new UserID("proem:user:jim1786"));
24.       rs.addUser("PIRATÉ ", new UserID("proem:user:ann4711"));
25.     }
26.
27.     public void handleProemEvent(Event event)
28.     {
29.       UserID remote_user;
30.
31.       if (event instanceof BeginEncounterEvent)
32.       {
33.         Ecounter encounter = ((EncounterEvent)event).getEncounter();
34.         PIRATÉ _UserProfile profile = encounter.getUserProfile();
35.         remote_user = profile.getUserID();
36.         if (hs.getSecondsSinceLastEncounter(remote_user)) > 24*60*60) //24 hours
37.         {
38.           Message mes;
39.           mes = ms.newMessage("proem:messagetype:playlisy",
                                  getProtocolID(),playlist);
40.           ms.send(remote_user, mes);
41.         }
42.       }
43.
44.       else if (event instanceof IncomingMessageEvent)
45.       {
46.         Message in = event.getMessage();
47.         if (in.getMessageType().equals("proem:messagetype:playlist"))
48.         {
49.           remote_user = in.getSender();
50.           if (rs.isMember("PIRATÉ ", remote_user)
51.           {
52.             PIRATÉ _PlayList remote_playlist = in.getMessageBody();
53.             recommendations = computeRecommendations(remote_playlist);
54.             Message mes;
55.             mes = ms.newMessage("proem:messagetype:recommedation",
                                          getProtocolID(),recommendations);
56.             PIRATE_UI.send(mes);
57.
58.           }
59.         }
60.       }
61.     }
62. }
```

The code of the PIRATÉ peerlet consists of a variable declaration section (lines 3-7) and three methods. The variables are:

- a variable for storing the user's play list
- a variable for storing the recommendations generated by the agent
- a variable for holding a reference to the global MessagingService object.
- a variable for holding a reference to the global HistoryService object.
- a variable for holding a reference to the global RelationshipService object.

The init method (line 19-25) defines a personal relationship between this user and three other users. This information will be used later on to filter incoming play lists. A more flexible approach would be to implement a user interface screen for adding and removing users from the relationship.

The main component of the peerlet is the handleProemEvent() method (lines 27-61). The method consists of one large if-statement with just two cases for handling events indicating the beginning of an encounter and an incoming message.

Upon receiving a BeginEncounterEvent, the peerlet first determines if the last encounter with this particular person occurred more than 24 hours ago. If this is the case, it constructs a message containing the play list and sends it to the encountered user.

Upon receiving an IncomingMessageEvent (lines 44-56), the peerlet checks to see if the received play list is one a person with whom the user has a special relationship. If it is the case, the play list is used to generate new recommendations, otherwise it is discarded. The recommendations are sent to the user interface for display.

VII.4.5 Summary

The PIRATÉ application illustrates the use of the following APIs:

- How to set up and use relationships with the relationship service
- How to use the history service to access information about past encounters

VII.5 mBazaar

mBazaar (see Chapter II.2.1) is a wearable community application that supports students in buying, selling and swapping of personal items like CDs, books, bikes, furniture, and electronics. mBazaar employs community agents that advertise items a user wants to buy or sell to other nearby mBazaar users. At the time of an encounter between two or more individuals, for example when two or more students meet after class at a local coffee shop, the community agents identify matches between advertised and desired items and exchange contact information. Depending on the users' preferences, contact information might include email addresses or pictures of the users. The picture enables users to identify each other and to verbally negotiate a possible transaction right on the spot.

The mBazaar application is designed to highlight an application that in some important respects is different from the one employed by PIRATÉ. First, it does not use user profiles: users exchange contact information only after their agents have determined a match. Second, instead of waiting for an encounter, agents broadcast classifieds to all nearby devices using the communitycast method of the messaging service.

VII.5.1 Scenario

The scenario description for the mBazaar applications is shown in Table 17.

Table 17. mBazaar Scenario

Name	mBazaar
Purpose	mBazaar enables users to advertise products and services they wish to buy or sell and to identify other users with complimentary desires.
population	"Localized groups" such as students or employees without strong personal relationships but a high frequency of chance encounters
Device	Input capabilities for text and possible images to define classifieds and users' contact information Output: display of classifieds and contact information
Before	-
Encounter	Devices exchange classifieds, identify possible match and if approved by users' exchange contact information
After	-

VII.5.2 Community Language

Protocol

The mBazaar community protocol is depicted in Figure 28. An agent starts by sending out a classified to a nearby agent. Upon receiving a classified, an agent determines if there is a match between its user's classifieds and the incoming classified. The agent sends information about a match to the user to garner the user's opinion. If the user likes the match and is interested in contacting the other user, he or she replies with a positive answer. If not, he or she replies with a negative answer and the exchange

terminates (states s7 and s8). In case of a positive response, the agent forwards the user's contact information to the remote agent which in turn informs its user. The user can then decide if he or she wants to pursue the opportunity and contact the other person. Note, that this scheme is asynchronous: one user sends contact information and the other receives it. This scheme does not guarantee that both users have mutual knowledge of each other.

If two simultaneous interactions occur (which may or may not be the case), each user ends up receiving the other users contact information.

The asymmetry in the original exchange has the advantage that users are in full control to whom they want to disclose their contact information. They will reply to a classified if and only if their agent found a match and they are interested in pursuing it.

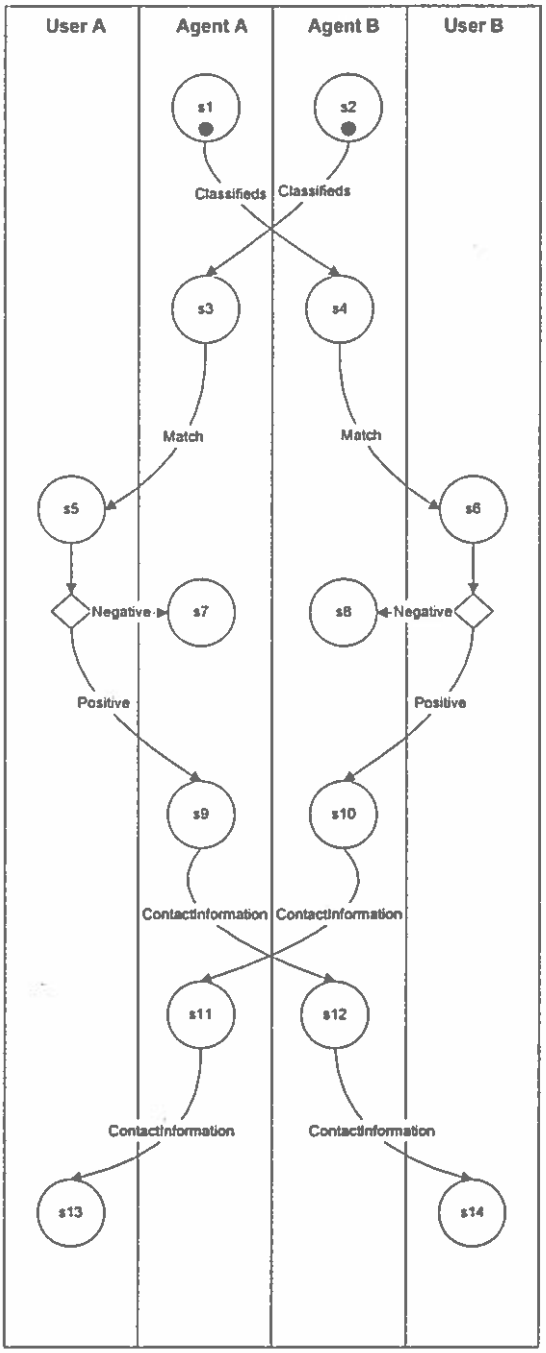


Figure 28. mBazaar community protocol

Vocabulary

The mBazaar vocabulary consists of four message types as shown in Table 18.

Table 18. mBazaar vocabulary

Message Type	Description	Implementation	
		Type	Description
Classifieds	A message of this type carries a list of individual classifieds	(boolean; String)*	Each classified consists of two fields: a boolean value indicating the classified type (buy = true; sell = false) and a String for a textual description of the product or service.
Positive	A message of this type indicates the user is interested in a match and wants to contact the other user	Void (carries no data)	
Negative	A message of this type indicates the user is not interested in a match or does not want to contact the other user	Void (carries no data)	
ContactInformation	A message of this type contains a user's contact information (e.g. email, cell phone number)	String	A string containing an unstructured textual representation of the contact information (e.g. "This is Hanna. Call me on my cell phone at 541-456 4560 to discuss a deal")

VII.5.3 Implementation

The mBazaar application consists of several Java classes.

- Four message classes (Program 6)
- A user interface class derived from PeerletUserInterface (not shown)
- The class mBazaar_Peerlet which implements the application logic (Program 7).

Program 6. mBazaar message classes

```
1.  class Classified
2.  {
3.      public boolean buy;
4.      public String description;
5.  }
6.
7.  class mBazaar_Classifieds implements proem.MessageBody
8.  {
9.      public java.util.List classifieds;
10. };
11.
12. class mBazaar_Positive implements proem.MessageBody
13. {
14. };
15.
16. class mBazaar_Negative implements proem.MessageBody
17. {
18. };
19.
20. class mBazaar_ContactInformation extends proem.MessageBody
21. {
22.     public String description;
23. };
```

Program 7. mBazaar peerlet class

```
1.  public class mBazaar_Peerlet extends GenericPeerlet
2.  {
3.      mBazaar_Classifieds classifieds;              // the user's classifieds
4.      mBazaar_ContactInformation contact; // the user's contact information
5.      MessagingService    ms = Proem.getMessagingService();
6.
7.      public PeerletID getPeerletID ()
8.      {
9.        return new PeerletID("proem:peerlet:mBazaar");
10.     }
11.
12.     public ProtocolID getProtocolID ()
13.     {
14.       return new ProtocolID("proem:protocol:mBazaar");
15.     }
16.
17.     public void init()
18.     {
19.       // set up classifieds and contact information via UI
20.       long ttl = 2;
21.       Calendar cal = Calander.instance().roll(HOUR_OF_DAY); // now + 1 hour
22.       Date later = cal.getTime(); // date indicating now + 1 hour
23.       DeliveryHint hint ms.newDeliveryHint(ttl, later);
24.       Message mes;
25.       mes = ms.newMessage("proem:messagetype:classifieds",
                              getProtocolID(),
                              hint,
                         classifieds);
26.       ms.communitycast(mes);
27.     }
28.
29.     public void handleProemEvent(Event event)
30.     {
31.       UserID remote_user;
32.
33.       if (event instanceof IncomingMessageEvent)
34.       {
35.         Message in = event.getMessage();
36.         if (in.getMessageType().equals("proem:messagetype:classifieds"))
37.         {
38.           remote_user = in.getSender();
39.           if (match(classifieds, in.getMessageBody())
40.           {
41.             mBazaar_UI.send(in);
42.           }
43.         }
44.       }
45.       else if (event instanceof UserInterfaceEvent)
46.       {
47.         Message in = event.getMessage();
48.         if (in.getMessageType().equals("proem:messagetype:positive"))
49.         {
50.           Message mes;
51.           mes = ms.newMessage("proem:messagetype:contactinformation",
                                 getProtocolID(),contact);
52.           ms.send(in.getSender(),mes);
53.         }
54.       }
55.     }
56. }
```

The code of the mBazaar peerlet consists of a variable declaration section (lines 3-5) and four methods. The variables are:

- a variable for storing the user's classifieds
- a variable for storing the user's contact information
- a variable for holding a reference to the global MessagingService object.

The init method (line 17-26) sets up the classifieds and contact information. Its most important task is to initiate a communitycast that sends out the user's classifieds to all members of the community. It defines a delivery hint object with a TTL and an expiration time. The TTL is set to 2, indicating a request to other Proem devices to forward the message to other peers. The expiration time is set to one hour in the future, guaranteeing the repeated communitycast of the same message for the next hour (the user should be able to set the expiration date of the communitycast, but for sake of simplicity has been fixed in this example. A more flexible approach would be to implement a user interface screen for setting the expiration date).

The handleProemEvent() method only needs to deal with one type of user interface event and one type of incoming message event.

Upon receiving an IncomingMessageEvent, the peerlet checks to see if there is a match. If there is, it forwards the classifieds to the user interface for display. The user can then respond.

Upon receiving a UserInterfaceEvent, the peerlet checks if it is a positive or a negative response. If it is positive it sends out the user's contact information.

VII.5.4 Summary

The mBazaar application illustrates a method of implementing community agents without using user profiles. Contact information is sent after negotiations between agents and, indirectly, between users has been successful. This approach is preferable in situations in which user are concerned about privacy. In addition, this example illustrates how to use communitycasting and delivery hints.

VII.6 WALID: Opportunistic Task Trading

WALID (Kortuem, Schneider, Suruda, Fickas, and Segall 1999) (see Chapter II.2.1) is our last case study. Historically, it is actually one of the first wearable community applications. It demonstrates how to define and implement a complex community protocol.

VII.6.1 Scenario

The scenario description for the WALID applications is shown in Table 19:

Table 19. WALID Scenario

Name	WALID
Purpose	To strengthen the social ties of a group by identifying opportunities for mutual assistance.
population	Close-knit groups with established trust relationships, for example groups of friends, classmates or colleagues.
Device	Input: user enters and maintains task list Output: display task list, alert user of a agent's proposed trade Sensors: determine user location (for example using GPS Global Positioning System)
Before	User defines tasks
Encounter	Devices exchange task lists, identify if there is a mutually beneficial trade and alert users
After	If users accept a deal proposed by their agents, task lists must be updated to reflect the trade

VII.6.2 User Profile

The WALID application is designed for a group of people who know and trust each other. It is unlikely that people are willing to trade tasks with people they are not familiar with. Therefore, user profiles are simple and only contain the user's name (Table 20).

Table 20. WALID user profile template

Attribute Name	Implementation Type	Description
name	String	The name under which the user is known to friends. May be the first name or an assumed name.

VII.6.3 Community Language

WALID employ ideas from game theory (Rosenschein and Zlotkin 1994) to ensure that results of negotiations are mutually beneficial. The first step in developing a community protocol is to formulize the task trading scenario. We use a variation of the *Postman domain* (Nash 1950) as follows:

> "Agents have to deliver sets of packages to destinations, which are arranged on a graph $G = G(V,E)$. The set V of vertices represents all possible destinations while the set E of edges represents routes along which agents can travel. Agents can exchange packages at no cost during encounters at any vertex.
>
> A *task* is a vertex v and indicates that an agent has to deliver a package to the location represented by v. A *task set* is a set of vertices. If vertex v is in an agent's task set, it means that he has at least one package to deliver to *v*.
>
> The cost of delivering a package from the agent's current location to a destination is defined by a *cost function* c: V x V -> N where $c(v_1,v_2)$ is the length of the minimal path from v_1 to v_2. The cost of a subset of destinations

$X \subseteq V$ is the length of the minimal path that starts at the current vertex and visits all members of X. Thus, the cost of a task set s is defined as the sum of the cost of all tasks contained in s."

The WALID community protocol is based upon Nash's *Product Maximizing Mechanism* (PMM). PMM is a three-step protocol: in the first step the two agents disclose their task sets; in the second step each agent proposes a deal (division of tasks) that is *pareto-optimal*[6]; in the third step the agents select and agree upon a winning deal. The deal that is selected is the one that offers each of the agents the most benefits which is the one with the highest combined cost savings.

Example

Let's assume two agents are negotiating at point C of the graph shown in Figure 29. Landmarks are indicated by circles and paths between landmarks by arrows. For sake of simplicity, distance between neighboring landmarks is assumed to be 1. Agent 1 must deliver packages to points A and D; Agent 2 must deliver a package to point A.

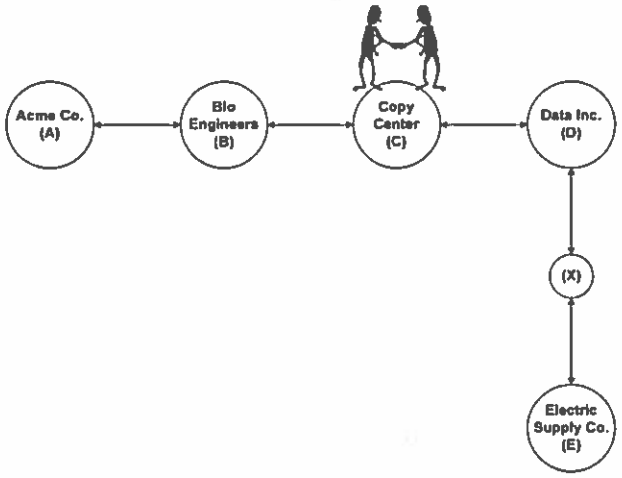


Figure 29. WALID Task Trading Example

[6] Pareto-optimal: No agent could derive more from a different agreement, without some other agent deriving less from that alternate agreement.

Step 1: In the first negotiation step, both agents disclose their respective tasks sets. As result, the agents know their own and their opponents tasks and the costs for each of them. This situation is shown in the following table:

Step 1. Disclosure of Tasks Sets	Agent Location	Task set	Cost = Length of minimal path
Agent 1	C	{A,D}	2 + 2 = 4
Agent 2	C	{A}	2

Step 2: In the second step, both agents propose a deal, where a deal is a new distribution of tasks. Let's assume agent 1 proposes that agent 1 deliver packages to point D and agent 2 deliver packages to point A. Let's further assume agent 2 proposes just the opposite (agent 1 delivers to A and agent 2 delivers to D). The following table shows the costs and utilities for each of these deals from the perspective of each of the two agents. The utility of a deal is defined as the cost saving of the proposed deal, that is as the difference of the costs between of the original situation and the proposed deal:

Step 2. Proposed Deals	Task distribution	Cost for agent 1	Utility for agent 1	Cost for agent 2	Utility for agent 2	Product of utilities
Deal 1	Agent 1: D Agent 2: A	1	3	2	0	0
Deal 2	Agent 1: A Agent 2: D	2	2	1	1	2

The cost of a deal for an agent is the distance the agent has to travel. The utility of a deal for an agent is the cost saving of the proposed task distribution. For example, with deal 1 agent 1 needs to travel from C (its current location) to D. The distance and thus cost is 1. The utility is 3, because without an exchange of task agent 1's cost is 4.

The *Product Maximizing Mechanism* says that the winning deal is the deal with the highest product of the two expected utility. Thus, in this example deal 2 is the winning deal.

Step 3: The third step involves the actual trade of tasks according to the winning deal. Since this scenario is about delivering packages a trade involves the physical exchange of packages. The final result is that agent 1 will deliver two packages to destination A, while agent 2 will deliver one package to destination D. The situation after this trade is shown in the following table:

Step 3. Final task distribution	Agent Location	Task set	Cost = Length of minimal path
Agent 1	C	{A, A} ≡ {A}	2
Agent 2	C	{D}	1

Protocol

The WALID community protocol diagram is shown in Figure 30.

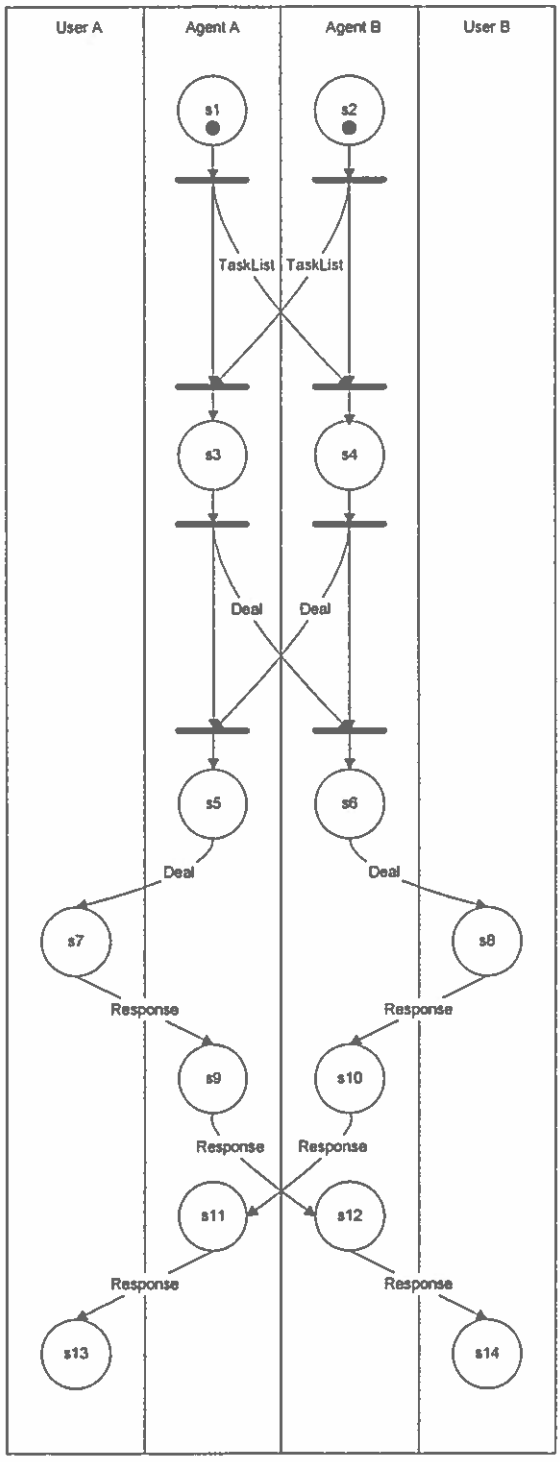


Figure 30. WALID Community Protocol Diagram

Vocabulary

The WALID vocabulary is defined in Table 21.

Table 21. WALID vocabulary

Message Type	Description	Implementation		
		Attribute Name	Implementation Type	Description
Tasklist	A message of this type represents the tasks a user has to do. It contains a list of tasks, each of which consists of a textual description and a location.	Description	String	Textual description of task
		Location_x	Int	Map location where tasks needs to be performed
		Location_y	Int	Map location where tasks needs to be performed
Deal	A deal is represented by two sets of tasks indicating an assignment of tasks to the two agents	UserA	Proem.UserID	ID of first user
		UserB	Proem.UserID	ID of second user
		TasksA	java.util.Set	The task set to be performed by user A
		TasksB	java.util.Set	The task set to be performed by user B
Response	A message of this type indicates whether a user agrees to swap tasks or not.	Accept	Boolean	A value of true indicates that the user accepts a deal, false indicates the use does not accept

VII.6.4 Implementation

The WALID application consists of several Java classes as listed in Program 8:

- WALID_UserProfile implements the user profile
- WALID_Tasklist, WALID_Deal and WALID_Response are classes implementing the vocabulary
- WALID_Peerlet is derived from GenericPeerlet and implements the community agent (Program 9).
- WALID_UI displays the user interface and handles the user interaction (not shown).

In our discussion, we will ignore the implementation of the user interface and limit our attention to the agent implementation.

Program 8. WALID Message and User Profile Classes

```
1.  class Task
2.  {
3.      public int id;
4.      public String description;
5.      public int location_x;
6.      public int location_y;
7.  };
8.
9.  class WALID_Tasklist implements proem.MessageBody
10. {
11.     public Set tasks;
12. };
13.
14. class WALID_Deal implements proem.MessageBody
15. {
16.     public UserID userA;
17.     public UserID userB;
18.     public Set  tasksA;  // tasks for user A
19.     public Set  tasksB;  // tasks for user B
20. };
21.
22. class WALID_Response implements proem.MessageBody
23. {
24.     public boolean response;
25. };
26.
27. class WALID_UserProfile extends proem.GenericUserProfile
28. {
29.     public String name;
30. };
```

Program 9. WALID Peerlet

```
1.  public class WALID_Peerlet implements GenericPeerlet
2.  {
3.      Set tasks;      // this user's task set
4.      HistoryService   hs = Proem.getHistoryService();
5.      MessagingService ms = Proem.getMessagingService();
6.      int state = 0;   // communication state according to CPD
7.
8.      static public PeerletID getPeerletID ()
9.      {
10.       return new PeerletID("proem:peerlet:WALID_01");
11.     }
12.
13.     static public ProtocolID getProtocolID ()
14.     {
15.       return new ProtocolID("proem:protocol:WALID");
16.     }
17.
18.     public void handleProemEvent(Event event)
19.     {
20.       UserID remote_user;
21.       WALID_Deal deal;
22.
23.
24.       if (event instanceof BeginEncounterEvent)
25.       {
26.         Ecounter encounter = ((EcounterEvent)event).getEncounter();
27.         WALID_UserProfile profile = encounter.getUserProfile();
28.         remote_user = profile.getUserID();
29.         if (hs.getSecondsSinceLastEncounter(remote_user)) > 60*60) //1hour
30.         {
31.           Message mes;
32.           mes = ms.newMessage("proem:messagetype:tasks",getProtocolID(),tasks);
33.           ms.send(remote_user, mes);
34.           state = 1;
35.         }
36.       }
37.       else if (event instanceof IncomingMessageEvent)
38.       {
39.         Message in = event.getMessage();
40.
41.         if (in.getMessageType().equals("proem:messagetype:tasks") && state == 1)
42.         {
43.           deal = computeDeal(tasks, in.getBody().tasks);
44.           in.setMessageBody(deal);
45.           ms.send(in.getSender(), in);
46.           state = 3;
47.         }
48.
49.         if (in.getMessageType().equals("proem:messagetype:tasks") && state != 1)
50.         {
51.           Message mes;
52.           mes = ms.newMessage("proem:messagetype:tasks",getProtocolID(),tasks);
53.           remote_user = in.getSender();
54.           ms.send(remote_user, mes);
55.
56.           deal = computeDeal(tasks, in.getBody().tasks);
57.           in.setMessageBody(deal);
58.           ms.send(in.getSender(), in);
59.           state = 3;
60.         }
61.
62.         if (in.getMessageType().equals("proem:messagetype:deal"))
63.         {
64.           deal = pickWinningDeal(deal, in);
65.           in.setMessageBody(deal);
66.           WALID_UI.send(in);
67.         }
68.
```

```
69.          if (in.getMessageType().equals("proem:messagetype:response"))
70.          {
71.            WALID_UI.send(in);
72.          }
73.        }
74.        else if (event instanceof UserInterfaceEvent)
75.        {
76.          if (in.getMessageType().equals("proem:messagetype:response"))
77.          {
78.            ms.send(in.getSender(), in);
79.          }
80.
81.        }
82.        else if (event instanceof EndEncounterEvent)
83.        {
84.          // do nothing
85.        }
86.      }
87. }
```

The code of the WALID peerlet consists of a variable declaration section (lines 3-6) and three methods. The variables are:

- a variable for storing the user's tasks (line 5)
- a variable for holding a reference to the global MessagingService object.
- a variable for holding a reference to the global HistoryService object.
- A state variable for keeping track of the communication state.

The main component of the peerlet is the handleProemEvent() method (line 17-85). It directly reflects the community protocol as defined by the WALID community protocol diagram shown in Figure 30. The method consists of one large if-statement with four cases for handling events indicating (1) the beginning of an encounter (2) an incoming message and (3) a user interface event and (4) the end of an encounter.

Upon receiving a BeginEncounterEvent, the peerlet constructs a message containing the tasks and sends it to the encountered user (line 27).

Upon receiving an IncomingMessageEvent, the peerlet either forwards the message to the user interface or, when receiving a task list, responds by sending a deal back the encountered user's agent (lines 45 and 58).

Upon receiving a UserInterfaceEvent, the peerlet forwards the user's response to the agent of the original sender (line 78).

An EndEncounterEvent is ignored by the peerlet (line 84)

VII.6.5 Summary

The WALID application illustrates how agents implement a reasonable community protocols. In order to build this application, the programmer needs to understand

- how to implement a peerlet by deriving a new class from the class GenericPeerlet
- how to handle events
- how to use the history service
- how to send messages to either the user interface or a remote agent.

VII.7 Summary

This chapter has presented several case studies of applying the development framework. In particular, it has shown two things:

- The use of the design language to capture key aspects of wearable community applications.
- The use of the Peerlet application framework for implementing wearable community applications.

The semi-automatic generation of code templates and the powerful service API dramatically simply the development by reducing the amount of code that programmers need to write.

Chapter VIII

DISCUSSION

This chapter discusses issues related to the utility and novelty of the research presented in this dissertation. First, it described our experiences in using the Proem platform in software engineering education. This will attest to the utility of our work. Next we discuss the differences and similarities of Proem with existing software development platforms. This will attest to the novelty of our work. Finally, we discuss possible enhancements and additions. In particular, we will talk about reputations as mechanisms for trust in wearable communities and peer-to-peer dissemination of wearable community applications.

VIII.1 Experiences of Using Proem in Software Engineering Education

In the previous chapter, we demonstrated how to build wearable community applications with the Proem platform. We were able to demonstrate that the amount of code that needs to be written to create applications is fairly small and that the code structure is straight forward. While we have successfully used Proem over the years as a research test bed, our own experiences are of limited use when it comes to evaluating utility of the WearCoM methodology and the Proem platform. Just as the usability of application software can only be evaluated through empirical tests with real end users, so can the utility of development tools only be evaluated by empirical tests with real developers. Preferably, these developers should be independent developers who were

not involved in the design and implementation of the tool itself. Rather than performing artificial experiments in a lab, we decided to study how Proem performed in practice by using it as development platform in several software engineering courses at the University of Oregon. This approach, while delivering more realistic results than mere lab tests, has one important drawback: we more not able to perform comparative studies of application development with and without Proem. In the following, we will outline our methodology and the result of this study.

VIII.1.1 Methodology

Between spring of 2001 and spring of 2002 three different software engineering courses were taught at the University of Oregon's Computer Science Department using Proem as development platform. These courses were:

- CIS 650: Software Engineering: Peer-to-Peer Computing, Spring 2001
- CIS 610: Mobile Information Systems, Fall 2001
- CIS 422/522: Software Methodologies, Spring 2002

600 courses are graduate level courses and 400/500 courses are mixed undergraduate/graduate level courses. Each course ran ten weeks.

User Population

In total, 35 students participated in this study. Of these, 10 were undergraduates and 25 were graduate students. All but 5 students were computer science majors. All students had medium to advanced Java skills. None of the students had any prior experience in wearable computing, peer-to-peer computing or wireless networking.

Projects

In total, 4 different projects were undertaken aimed at building 4 different wearable community applications:

- mClique: a wearable community application aimed at group awareness. The goal of this application is to determine groups (= cliques) of mutually connected friends.
- mBazaar: a wearable eBay application that enables users to buy, sell and trade goods and services.
- PIRATÉ : a collaborative music guide.
- Infomediation: a distributed application that enables users to access infomediation services provided by other users. Each service enables users to query the database that drives one of the major book sites (Amzon.com, BarnesAndNoble etc.). As result, users were able to perform collaborate price comparisons for books. This application is described in (Segall et al. 2002).

All projects were undertaken by one or more teams of between 3 and 4 students. Both mClique and mBazaar were implemented by 1 team; PIRATÉ was implemented twice (by two different teams) and the infomediation application was implemented by 5 teams. The infomediation application was unique in that it was developed using a distributed development process. All teams specified a common community language and then independently implemented interoperable service components. The end result was a heterogeneous system. In contrast, all other systems were individually designed and implemented by just one team.

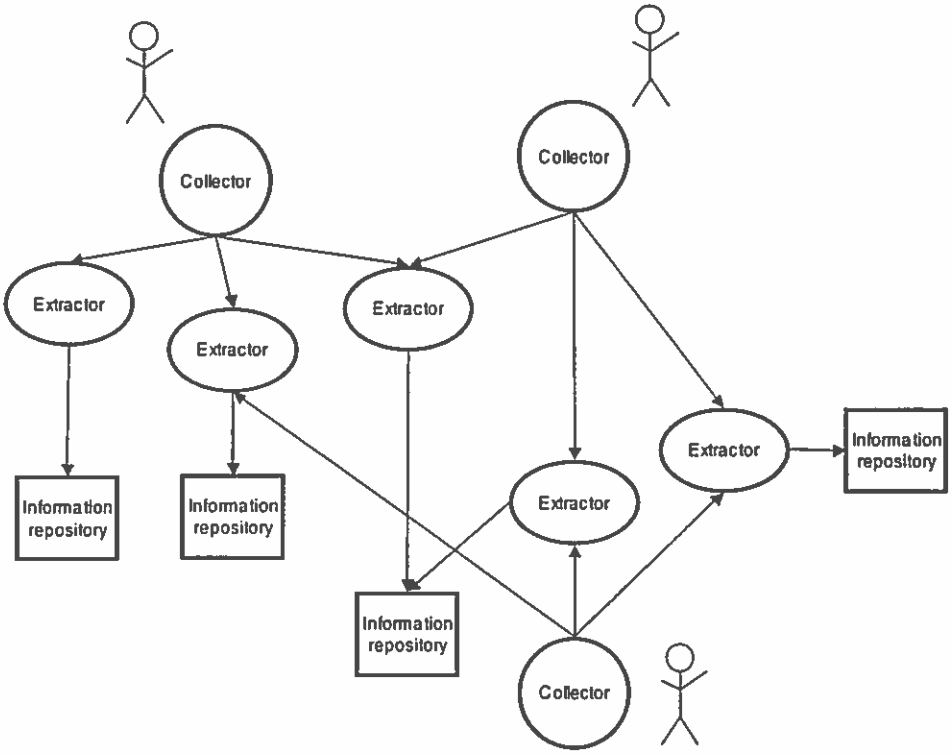


Figure 31. Infomediation Architecture

Process

Each course ran for 10 weeks and was divided into three phases:
Introduction into the application domain (1 week = 2 lectures)
Introduction to the WearCom methodology and the Proem platform (1 lecture)
Project work (design, specification, implementation)

The introduction to the WearCom methodology and the Proem platform was done using one 45 minute lecture. At the same time, students were provided with a Proem User Guide that explains the installation and setup, a Programmer's Manual that teaches how to write peerlets, and a single rather simple example application.

The actual project work was divided into three iterations. At the beginning, each group outlined the tasks they planned to complete, and the features they planned to implement by the end of each iteration. At the end of each iteration, the groups

presented a demonstration of their progress, a summary of the challenges they encountered, and a discussion of unanticipated problems. The end result of each project consisted of design documentation and a running prototype implementation. The first 2 weeks of the project work were reserved for reading, drafting specifications and software design. The actual implementation took between four and five weeks.

At the end of each course, a questionnaire was handed out asking about each student's experience with Proem, problems encountered and suggestions for improvement.

Cooperation between teams was strongly encouraged, and extraordinarily useful, as many teams were solving similar problems. The only rule was "You are cheating if you do not acknowledge the contributions of others." Throughout the project, teams traded code, technical knowledge, and advice on using Proem.

Equipment and Tools

Development was performed in Java SE 1.3 on Sun workstations, PCs and notebooks. Because of logistical problems and lack of hardware, none of the teams had access to actual wearable hardware.

The teams were free in their choice of development tools, but encouraged to utilize version control software, bug tracking software etc.

The Proem version used in all projects was 1.1. It is very similar to the one described in this dissertation but lacked support for user interfaces. Thus, each team had to implement a user interface from scratch using Swing and AWT.

Measured Variables

During the courses, we closely observed the progress of each team and evaluated the final software implementation. We assessed Proem's quality along the following dimensions:

- Learning time: how long does it take each team to implement the very first peerlet? The conceptual simplicity of Proem can be approximated by how long it takes students to learn Proem before they are able to use it independently.
- Implementation time: how long does it take each team to implement their prototype (excluding time spent on design and specification). The effectiveness of the support provided by Proem can be measured by how long it takes students to implement a full-featured application on top of Proem.
- Complexity of Developed Applications: Complexity of an application is hard to measure. We thus relied on two objective measures: application size and number of Proem API functions used (excluding user interface code).
- Support Requirements: how many support incidents (email, personal contact) occurred over the entire project lifetime? In order for students to have timely access to technical support related to the Proem platform, we set up a mailing list. In addition, students had the opportunity to discuss questions with Proem developers during weekly face-to-face meetings. For each project, we tracked the number of support instances over the entire project lifetime.
- Subjective satisfaction: what is the level of satisfaction among students (based on the questionnaire handed out at the end of class)?

VIII.1.2 Results

The results of the study are summarized in Table 22.

In our experiment, in under one week students were able to get the Proem platform up and running, experiment with the example application and write their own very simple tests (participating students carried a full work load with 3 to 4 classes and thus were only able to spend a fraction of their time on this particular project). Data extracted from informal interviews and the questionnaire indicates that it took individual students between 1 and 5 hours to write their first application. Considering that Proem consist of about 80 classes and interfaces this is a comparatively short time.

The implementation time was bounded by the set up of the courses. Each team had about 4 weeks for the three iterations. The time estimates for individual students vary widely and range from 0 hours to 15 hours per week. Some of the teams were highly organized with some students responsible for only design and documentation and others only for implementation. Considering the varying quality of the implementations and the varying scopes of each project it is difficult to judge these numbers. In comparison, two similar projects not based upon WearCoM and Proem took 3 month each with a weekly time estimate of between 10 and 15 hours.

The application size ranged from 22 Java classes (120KB source code) to 39 Java classes (200KB source code). The overall complexity of most applications is high, given the decentralized character and the nature of the interactions. Most of the complexity, however, is hidden in the Proem network and service components.

Table 22. Result Overview

	Learning time	Impl. time	Complexity (source code size excluding user interface)	Complexity (percentage of API used)	Support requirements (# support incidents for 10 week time period)
mClique	1 week	4 week	30 Java classes (~120KB)	~20%	4
mBazaar	1 week	4 week	25 Java classes (~110KB)	~17%	3
PIRATÉ (version 1)	1 week	4 week	22 Java classes (~120KB)	~39%	5
PIRATÉ (version 2)	1 week	4 week	39 Java classes (~200KB)	~21%	1
Infomediation (averaged over all teams)	1 week	4 week	30 Java classes (~150KB)	~20%	3
Average	1 week	4 weeks	~ 30 Java classes (~145KB)	~22%	< 3

Somewhat disappointing is the low percentage of API calls used by the application. In average, applications make use of only 22% of the API. After evaluating the source code, we found that some of the teams reimplemented functionality that is

provided by Proem instead of using the built-in functionality. Informal discussions revealed that some teams were not as familiar with the API as expected and thus they were not aware that they could have used built-in API calls. Up to 50% of the APIs could have been employed by each application, indicating that about half of the required functionality had not been discovered by students. Subsequently, we modified the names and signatures of some of the APIs in order to clarify the concepts and abstractions represented by them. It has yet to be determined if this will increase the clarity and usability of the API.

The support requirements is the area were Proem really shines. The numbers are consistently low and never reach more than 5 support instances per project for the entire 10 week span of the course. A contributing factor to this surprisingly low number is that students had been encouraged to share their understanding of the methodology and Proem across teams. About half of the support incidents were questions related to if and how a particular function is implemented by Proem, while about 25% were bug reports. Overall, we are very satisfied with the stability of the Proem platform.

The evaluation of the subjective satisfaction suffered from a poor response to the questionnaire. Only few students submitted answers to the free-form questions that asked for particular good or bad experiences. However, when students answered the questions, the response was overwhelmingly positive. For example, two students answered the question "What do you particularly like about Proem?" as follows:

- "Its intuitiveness and good abstraction. We don't need to delve into its details and still can do complex job."
- "Working with Proem is just like working with JAVA APIs. Ease of use and understanding. Easy to start and get going."

Informal interviews after the end of each course indicated a very high satisfaction level and about 80% of students were convinced that they could not have done the project without Proem.

VIII.1.3 Key Benefits

Summarizing our experiences during research projects and during the course experiments, we see the main benefits of the *Proem* platform in the following areas:

Reducing the Complexity of Building Applications

Proem simplifies the development task by raising the level of abstraction. This is achieved by three interrelated measures. First, the *Proem* platform provides network transparency and hides specifics of the underlying communication network from programmers. Second, it provides programmers with an application framework and a powerful set of APIs. Third, it realizes a simple programming model with a uniform event-based mechanism for peer-to-peer communication, presence-awareness and user interface management

Enabling Rapid Development

Proem enables rapid prototyping through a combination of several measures: first, the amount of code that needs to be written for simple applications is small. This allows programmers to have a simple application up and running within a very short time frame and add to it gradually. Second, *Proem* supports code reuse through a virtual machine architecture and application plug-in modules (= peerlets). A complex application can be realized as a collection of communicating peerlets that all run on the same machine within the same peerlet engine. Third, because of *Proem's* cross platform capabilities, developers can write and test applications on workstations and afterward deploy them on wearable computers.

Enabling Independent, Distributed Development

Proem represents an open, distributed software infrastructure for wearable community applications. With *Proem*, multiple developers can define a community protocol and create independent yet compatible implementations. This is made possible by two interrelated measures: first, by providing a common communication substrate in the form of the *Proem* Transport Protocol; second, by enabling developers to specify interfaces between devices in the form of peerlet protocols.

Enabling Long term Modification

Because of the virtual machine architecture and high-level APIs, applications are shielded from specifics of the underlying technologies. Modifications necessary due to innovations in communication and device technology can be hidden by the peerlet engine and are transparent to applications.

Interoperability among Heterogeneous Devices

Finally, interoperability among heterogeneous devices is guaranteed by two interrelated measures. First, the cross-platform nature of the peerlet engine makes it possible to install and run existing applications on any devices with a peerlet engine. Second, the network independence of the XML-based Proem Transport Protocol makes it possible for two different Proem implementations to interoperate regardless of the programming language used to implement them.

VIII.2 Related Software Platforms

The Proem software platform has not been developed in a vacuum, but has benefited from ongoing work in many areas including mobile ad hoc networking, peer-to-peer computing, mobile middleware platforms and mobile groupware. This chapter describes software platforms that address similar issues and challenges as Proem. As decentralization and peer-to-peer communication is one of the primary attributes of wearable community systems, we limit out attention to decentralized and peer-to-peer platforms. This excludes, for example, many of the software infrastructures developed for smart environment and interactive workspaces like Stanford's iRoom project (Ponnekanti et al. 2001). Similarly, this survey does not cover intelligent agent platforms or languages such as Jackal (Cost et al. 1999) or KQML (Genesereth and Ketchpel 1994; Finin et al. 1994). Although we make use of the term agent to describe wearable community software artifacts, Proem is not an intelligent agent platform. In particular, it does not employ a formal agent communication language and semantic content representations. Agent languages tend to restrict agent communication. We feel that such a formal definition provides few advantages in a platform mainly concerned with supporting human collaboration.

VIII.2.1 Mobile Middleware Platforms

Proem can be seen as an example of a *mobile middleware platform*. Mobile middleware aims at facilitating communication and coordination of distributed mobile components, concealing difficulties incurred by mobility from application engineers. Until now, very few middleware solutions have been designed for decentralized mobile systems and ad hoc network environments. Moreover, most such platforms focus on data management as the main problem area. Examples include Bayou (Demers et al.

1994; Edwards et al. 1997), TSpace (Wyckoff 1998) LIME (Picco et al. 1999; Murphy 2000; Murphy et al. 2001) and XMIDDLE (Mascolo et al. 2001; Mascolo et al. 2002).

One of the first mobile platforms to address data management in mobile systems was Bayou. The Bayou system was designed to support collaboration among mobile users who are not continuously connected. It employs weak consistency replication techniques to manage replicas of shared user data (for example calendar items, electronic mail messages, documents). Arbitrary read and write operations to any replica are permitted without the need for explicit coordination with other replicas: every computer eventually receives updates from every other, either directly or indirectly, through a chain of pair-wise interactions. Weakly consistent replicated data is not transparent to applications; instead, they are aware that they may read weakly consistent data and that their write operations may conflict with those of other users and applications.

TSpace and LIME are representatives of a tuple-space approach first developed in Linda (Gelernter 1985). TSpace is a middleware system whose goal is to support communication and data management on hand-held devices. Unlike Proem, TSpaces makes a sharp distinction between clients and servers. The client side has a small footprint, as it is designed to run on devices with scarce resources. The server side is supposed to be resource richer and it may run a relational or object-oriented database to achieve persistence of tuple spaces.

LIME uses multiple tuple spaces, called interface tuple space each permanently associated to a mobile unit, and introduces rules for transient sharing of the individual tuple spaces based on connectivity. Each interface tuple space contains tuples that the unit wishes to share with others and it represents the only context accessible to the unit when it is alone. Access to the interface tuple space takes place using conventional Linda primitives, whose semantics is basically unaltered. However, the content of the interface tuple space is dynamically recomputed in such a way that it looks like the result of the merging of the interface tuple space of other mobile units currently

connected. The advantage of this model is that applications only need to worry about the content of their local tuple space, but not about connectivity and data distribution.

XMIDDLE is an XML-based mobile computing middleware designed for ad hoc networks. Relying on peer to peer networking, it allows applications on different hosts to share data, manipulation it remotely or off-line, and reconciles the changes upon reconnection in an application dependent manner. Shared data is represented in a tree data structure. In order to share data, a host needs to explicitly link to another host's exported data branches. The data are cached on the target host, enabling disconnected operations. Hosts may explicitly disconnect from each other or may be separated through physical movement. Upon reconnection, the XMIDDLE platform automatically reconciles any changes to shared data, using application metadata to resolve any conflicts. In addition to simple data sharing, XMIDDLE supports a distributed versioning scheme based on snapshots of shared branches. Version information is stored on each host in a manner that minimizes storage requirements. Hosts reconciling changes can identify common versions and use the latest one as a basis for reconciliation, thus conserving computational power and network bandwidth.

Anhinga (Kaminsky 2001; Kaminsky and Bischof 2002) is another mobile middleware platform specifically designed for ad hoc networks of small mobile wireless devices. In contrast to the above mentioned platforms it not only concentrates on data management issues but also addresses communication aspects of collaborative applications. The unique feature of Anhinga is the Many-to-Many Protocol (M2MP), a network protocol based on broadcast messages that is designed for many-to-many communication among proximal devices. M2MP does not use device addresses. Rather, all M2MP messages are broadcast to all devices at once. The M2MP protocol represents an alternative to traditional message routing in ad hoc networks and shares similarities with the broadcast capability of the Proem Transport Protocol. In contrast to Proem, Anhinga only offers a low-level communication abstraction based on the notion of method calling.

Proem differs from these middleware platforms in its scope. It does not aim at providing a generic solution for a narrow problem domain (data management or communication), but a broad functional and abstraction layer for a particular class of applications, namely applications to support spontaneous collaboration among individuals. Important capabilities provided by Proem but not addressed by the above middleware platforms are identity management, presence awareness, communication models based on human identity and not device identity, and models of cooperation. On the other hand, Proem currently does not address data management issues, mainly because the wearable community applications we investigated did not require data sharing among users. However, the models and algorithms developed as part of Bayou, TSpace, LIME, XMIDDLE and Anhinga could easily be integrated into the Proem platform and made available as yet another service component if called for by future applications.

VIII.2.2 Peer-to-Peer Platforms

The recent interest in peer-to-peer computing has spawned several efforts aimed at creating peer-to-peer development platforms. The most prominent example is Sun's JXTA (Gong 2001). JXTA is a core services framework that developers can adopt to build a wide range of sophisticated peer-to-peer applications. It provides the most basic functionality required by any peer-to-peer application: peer discovery and peer communication. A mobile version of JXTA, called JXME (Arora et al. 2002), which is aimed at the mobile information devices platform contained within the Java 2 Platform Micro Edition is currently under development.

Both Proem and JXTA follow a protocol-centric paradigm in which a standard set of XML-based message formats is designed to let devices discover each other and interact with each other. Since the message formats are programming language neutral, applications can be written in different languages to run on heterogeneous platforms and still collaborate.

The differences between JXTA and *Proem* lie in the following areas:

Proem is a platform for applications to support collaboration among people. User profiles, the presence protocol and people-based addressing all support this goal. In contrast, JXTA only supports interactions between peers, but does not provide abstractions and concepts related to users. Proem is tailored for highly dynamic ad hoc network environments where connectivity and resource availabilities change constantly. The Proem Transport Protocol defines a functional layer on top ad hoc networks that provides the foundation for communication among peers. The Proem messaging service provides developers with a simple communication model and hides details of the underlying network technology. In contrast, JXTA's communication model is geared towards semi-stable environments like the Internet.

Finally, JXTA is a generic peer-to-peer platform that integrates only the most basic functionality required by any peer-to-peer application (namely discovery and communication). In other words, JXTA provides broad but shallow application support. In contrast, Proem's support is narrow, but deep. It is designed for a specific application class and the Proem service layer combines the right abstractions and common functions required for building applications within this class.

In addition to JXTA, a number of collaborative peer-to-peer platforms have been developed:

- Groove (Groove Networks 2002) is an enterprise collaboration platform which can be used to connect peer groups on the fly using its own set of protocols. Both the client and server pieces of the Groove application sit on the user PC, but Groove peer groups can include network-connected mobile devices.
- Magi (Bolcer 2000) is a peer-to-peer collaboration platform which relies on Internet-standard protocols such as http for data serving and SSL for security. Magi supports the WebDAV standard and allows users to read and write to remote documents stored on other users' computers. Magi can run within a PocketPC operating system and thus supports the latest generation of handheld devices.

- Pocket DreamTeam (Roth 2002) is a mobile version of the groupware platform DreamTeam (Roth and Unger 1998). DreamTeam is an environment for developing synchronous shared applications. It makes collaboration aspects transparent to applications, thus allowing developers to built collaborative multi-user applications just as easy as traditional single user applications.

These platforms support traditional collaborative activities like file sharing, instant messaging and collaborative document editing. Although they support mobile peers, they are more geared towards work-related activities within teams. They do not support the spontaneous, presence-aware activities typical for wearable community applications and supported by Proem.

VIII.2.3 Other Platforms

Hive (Minar et al. 1999) is a distributed agent platform developed at MIT that serves as software infrastructure for MIT's "Things That Think" project. Although billed as agent platform, HIVE is actually much closer to a distributed object system.

Hive is based on the idea of an "ecology of distributed agents". A HIVE application is created out of the interaction of multiple agents across a network. A Hive agent is a small, autonomous, self- describing program. Each agent is located in a particular place (in Hive, called a cell), and uses various local resources (shadows). Agents communicate with each other to share information and access to resources. An application is made from the communications and actions of agents. Hive provides ad hoc agent interaction, ontologies of agent capabilities, mobile agents, and a graphical interface to the distributed system.

Similarly to JXTA, Hive is a generic platform that with shallow support for a large class of applications. Hive is mainly concerned with providing a decentralized communication infrastructure and is more like a distributed object systems than a middleware platform.

The main differences between Hive and Proem can be summarized as follows:

- Hive is deeply rooted in the Java programming language and uses Java types to encode agent capabilities. This makes it very hard to implement Hive in any other programming language or to build a heterogeneous Hive application that consists of agents implemented in different programming languages. Proem, in contrast, has been designed to enable interoperability between heterogeneous implementations. This goal has been achieved by defining a programming language neutral communication standard in form of the Proem Transport Protocol. Proem follows a protocol-centric paradigm in which a standard set of message formats is designed to let devices discover each other, exchange data and events, and otherwise interact with each other. Since the message formats are programming language neutral, applications can be written in different languages to run on heterogeneous platforms and still collaborate.
- Hive does not provide a standard way for agents to dynamically detect who else is on the Hive network. Contrary to the idea of a decentralized system, Hive relies on a registry to maintain membership in the global Hive network. In order to find out who else is on the network Hive cells need to contact the registry. Hive does not allow agents to advertise their presence throughout a network and thus does not adequately support the ad hoc formation of federations of Hive agents. This makes it very difficult to deploy Hive in dynamic ad hoc network environment. In Proem, discovery is completely decentralized and ad hoc.
- Most importantly, Hive is agnostic about users. While Proem supports user profiles, people-based addressing and user relationships, Hive is purely agent (or object) centric and does not provide any high level APIs for applications to support human collaboration.

VIII.2.4 Summary

Proem provides a set of features that in this combination is not provided by any other platform. This includes

- peer-to-peer communication
- decentralized discovery
- presence-awareness
- human-centered communication
- platform independence
- interoperability
- simple programming model
- extensive set of APIs for applications supporting spontaneous human collaboration (identity management, presence-awareness, relationships, interaction histories etc)

Related platforms either focus on a narrow problem domain (for example, data management) or provide generic support for a very large class of applications. More important, however, is the fact is that the related platforms lack an associated methodology for developing applications. Without a methodology, there is no support for conceptual design and specification of applications supported by these platforms. Furthermore, without a methodology, it is very difficult to reuse design and software artifacts across applications. By integrating design, specification and implementation support, the WearCoM methodology and the Proem platform are uniquely equipped to facilitate rapid, yet systematic development of wearable community applications.

VIII.3 Enhancements

The WearCom methodology and the platform are work in progress. There are two important shortcomings: first, although trust among individuals is an important aspect of wearable communities, Proem does not provide any trust mechanism; second, although peer-to-peer dissemination of wearable community applications is part of the WearCoM methodology, it is not implemented in Proem. In the following, we will discuss possible solutions to these shortcomings.

VIII.3.1 Trust in Wearable Communities

Social interactions are built around trust and at any given time, the stability of a community depends on the right balance of trust and distrust. We trust our friends to honor their word, we trust a doctor to give us medical advice and we often trust complete strangers to help us when we are lost or need assistance. Even more so, members of a wearable community need to be able to reason about trust, to facilitate their social interactions. This leads to the question of how to evaluate the trustworthiness of wearable community member. In this chapter, we propose a trust framework for wearable communities based on reputations. The framework is grounded in real-world social trust characteristics and mimics the word-of-mouth flow of personal recommendations in the real world. The trust framework is based on a decentralized model and exclusively relies on direct peer-to-peer interactions.

The proposed trust framework is not integrated into the Proem platform but can easily be added due to its peer-to-peer nature. We will sketch a Trust API as extension to the Peerlet Framework.

Approaches for Managing Trust

Although a small number of trust models have been proposed for the virtual medium, we find that they are largely impractical for dynamic wearable communities. The traditional solution to the issue of trust is that of a centralized reputation server. An example of a widely used centralized reputation server is the one used by e-bay.com to provide trust information (reputations) about potential buyers or sellers in their open market.

The centralized approach has many shortcomings. The most significant is its vulnerability to falsified information. A user with a large number of positive reports and a small number of negative reports looks like a legitimate trader unless the positive reports have all been falsified. Another problem with centralized reputation servers is the ability of anyone to know their exact reputation. This lets the bad trader know when to "flush" their old identity and to start again with a clean slate. Finally, there is the trustworthiness issue of a centralized reputation server. A centralized trust server places a great deal of unsupported trust in the security of the centralized server. Can an on-line auction-house's trust server trust information be trusted when they themselves are the buyers or sellers? What about the reputations provided regarding their advertisers? Can the security of their trust server be trusted? How they will use the personal information, contained in trust/distrust statements you make about other users.

Evidence of the ineffectiveness of a centralize trust server can be seen by the almost 10,000 cases of fraud relating to on-line auctions reported in 1999 to the National Consumer League (NCL 1998). The National Consumer League also posts the following warning regarding online auction sites: "Look at the auction site's feedback section for comments about the seller. Be aware that glowing reports could be "planted" by the seller".

Another approach to handling trust is the decentralized model used by PGP (Zimmermann 1995). The PGP system introduces does not use a centralized reputation server but rather it allows users to vouch for one another. An example of this is having

user A vouch for user B, and user B vouches for user C, and thus someone who trusts user A will also trust user C based on the chain of recommendations. This is the basic transitive trust property that decentralized trust servers are based upon.

Disseminating Reputations in Wearable Communities

We have developed a decentralized reputation framework (Schneider et al. 2000) that uses opportunistic physical encounters to propagate trust data. Each member in the wearable community stores database information containing the reputation information on everyone they have personally negotiated with or have had information regarding others negotiations distributed to them. This reputation database has the following structure:

- userid: unique identifier of the user the reputation entry is about.
- personal opinion value: personal opinion about user userid based on personal encounters.
- number of personal encounters with that person
- community reputation value: summarized opinions of other users about user userid.
- number of users who contributed to the community reputation value.

The opinion values range from –2.0 to 2.0 with 0.0 corresponding to no information.

Whenever members of a wearable community complete a negotiation and can evaluate how well their negotiating partner fulfilled their part of an agreement, they record their opinion of their negotiation partner in their local reputation database. This information is propagated through a mutual exchange of local reputation databases whenever a social encounter in the wearable community occurs.

This exchange allows both parties to update their databases after reasoning about the information provided. Providing reputation has the effect of telling every person you encounter "here are my feelings about the people I've traded with and here's

what I've been told about them." This information may then be passed on during subsequent encounters and will eventually spread throughout the entire wearable community.

Reasoning about and applying reputation information involves several steps:

3. If there has been a recent exchange with the same user, then we assume that the information will only reinforce old views and should be discarded. This prevents any specific user from too much weight.
4. If another user's opinion differs too greatly from the personal opinion value then the information provided is suspect and will be disregarded.
5. Otherwise, the values from the other users' personal opinion and community opinion will be averaged in to the local user's community information values. It is important not to base any reasoning on the number of recommenders or the number of encounters, as this could allow users to have a stronger "voice" in recommendations by artificially inflating these values.

This system solves the key problem of the centralized reputation server as each person can only provide a single positive or negative feedback about any other individual in a category. To get 10,000 positive comments about an individual would require 10,000 different people to provide positive feedback. This is as opposed to the centralized server where one person can say positive things 10,000 times.

The problem of an individual knowing their exact status in the eyes of the community also vanishes under this system. It is replaced with only a general knowledge of ones reputation. The distributed reputation server also solves the need for trust in a foreign server. Everyone maintains their own information and if part of the databases suffers due to negligence or intentional corruption it will be corrected by the valid information in the rest of the community.

By using social interactions to trade entire trust set data we gain the advantage of a rapid propagation of information from the numerous encounters in an environment, yet we maintain the advantages of a system that mirrors the way real world reputations

spread – by word of mouth during daily encounters. In addition, a decentralized system does not suffer from the issues that occur with traditional reputation servers.

VIII.3.2 Peer-to-Peer Application Dissemination

The sixth phase of the WearCoM methodology entails the dissemination of wearable community applications through a peer-to-peer distribution mechanism. The current implementation of Proem does not support such a mechanism; however it provides most of the features necessary to implement it. Most importantly, Proem features a virtual-machine architecture and supports the explicit loading (and unloading) of application components (= peerlets).

The missing features are related to a safe execution environment for peerlets and a transport mechanism for peerlets.

VIII.3.3 Safe Peerlet Execution Environment

In order to prevent harmful code from being spread throughout a community and to ensure safe execution of peerlets, a secure runtime environment with the following properties is required:

- Verification of application code - to unsure integrity of application code
- Safe execution of application code (sand boxing) - to prevent malicious code from causing harm

Clearly, the Java programming language provides most of the building blocks for a safe peerlet execution environment. It is similar to an execution environment for applets.

VIII.3.4 Peerlet Transport Mechanism

A peerlet transport mechanisms can be realized through a combination of a new built-in Proem protocol and the implementation of a transport manager. The new protocol, called Proem Distribution Protocol (PDP), can easily be built on top of the Proem Transport Protocol. Because peers exchange metainformation during an encounter including a list of the peerlets each of the carries, the PDP protocol is a simple request-reply style protocol: in the first step, one peer (the client) request a particular peerlet from another peer (the server). In the second step, the server either sends the peerlet or sends a negative response.

The protocol logic is implemented on both sides by a transport manager. This manager has a user interface that enables users to initiate a request and approve or disapprove a transmission. As a matter of fact, it would not be difficult to implement the transport manager as a peerlet. This would only require minimal changes to the underlying Proem system.

VIII.4 Design Principles for Wearable Communities

The Proem platform provides a software infrastructure for wearable community applications and the WearCoM methodology guides the design and development process of such applications. Yet these both elements do not guarantee that the applications we build will actually contribute to the formation of successful wearable communities. In Chapter II.3 we asked: "When does a collection of individuals using wearable computers become a wearable community?" After having developed an infrastructure for wearable communities and having developed a number of wearable community applications, we can provide some preliminary answers.

Our projects have provided us with insight into what works and does not work when building wearable communities. We have codified our experiences in form of design principles for wearable communities. Design principles are guidelines that may be applied to evaluate existing designs, guide the design process and educate designers about the characteristics of successful systems. In Sociology, there exists an extensive body of empirical research on the success and failures of online communities (Godwin 1994; Kollock and Smith 1996; Kollock 1998; Preece 2000) and face-to-face communities (Ostrom 1990). While we have not done any empirical studies on dynamics of wearable communities and the behavior of their members, we have identified six preliminary design principles that we believe contribute to successful wearable communities.

Principle 1: *Make users aware of hidden benefits of random encounters.* The WALID community application has been designed according to this principle. When two individuals meet, they are usually not aware of each others tasks and if they happen not to talk about their respective errands will never find out that they are able to benefit from a trade. The proactive personal applications employed by WALID, however, are designed to look out for possible trading partners and to make their respective users aware if there is a mutually beneficial opportunity. Determining the value of a trade for each individual requires slightly complex but straight-forward calculations based on the current location, the distance to each destination and the type of the tasks. This type of calculation can easily be performed by software, but is difficult to do by humans. Once they have been made aware of an opportunity, users have the liberty to act upon or ignore the advise of their applications.

Principle 2: *Reward users for social interactions.* We followed this strategy when we designed the Genome game: the game is constructed in a way that the person who interacts the most frequently with other people and with the largest number of people has the greatest chance of winning.

Principle 3: *Enable individuals to recognize each other.* Without identity and mutual recognition, a group of people will always remain strangers. In order to promote altruistic behavior and cooperation, a social feedback mechanism is required that allows individuals to evaluate and keep track of the behavior of others. Such a feedback mechanism requires identity and mutual recognition.

Principle 4: *Support Expressiveness.* Live Action Role Playing games have gained widespread international popularity. Such games are social events that involve gatherings of people who have taken on imaginary identities based on historical or mythical characters. The social interactions taking place during such a gathering are the key feature contributing to the players' enjoyment. Similarly, wearable communities enable members to invent digital identities to augment their true identities. Wearable communities should promote such creativity and artistic expressiveness to the largest extent possible.

Principle 5: *Use knowledge about past encounters to enrich present interactions.* One of the most important aspect of social encounters is our ability to recognize other people and to remember if, when, where and under which circumstances we have met the person before. Wearable computers can not only be made aware of the user's current context (defined by the presence of other individuals), but can easily keep track of an individual's *interaction history* (historical context). Bringing this knowledge to the table can augment encounters in a significant way as highlighted by our FriendFinder application.

Principle 6: *Include social feedback loops.* Any community has members who show disruptive or unruly behavior. In newsgroups, these are people who consistently and inappropriately flame other members. On EBay, these are members who cheat by not paying for items they purchased from other members or by artificially inflating bid

prices of items they sell. Successful communities are successful in part they have found a way of policing the behavior of their members, but not by giving some members extraordinary powers, but by way of social feedback mechanism. For example EBay, uses reputations (= aggregated signed information from EBay users on a user's past transaction history) for giving members a way of rating and judging other members' trustworthiness. The community as a whole can thus choose to "exclude" members from their community.

These principles are preliminary as they are the result of informal observations by developers and not of systematic empirical studies. The necessary studies are outside of the scope of this dissertation. For a brief discussion of future research in this direction, see Chapter IX.

VIII.5 Summary

This chapter discussed the utility and novelty of the research presented in this dissertation. First of all, we presented an evaluation of the WearCoM methodology and Proem platform. Through a set of experiments in which we used Proem as development platform in software engineering courses, we were able to determine its key benefits. These are:

- A reduction in the complexity of building applications achieved by a combination of network transparency, a high-level application framework and a simple, unified programming model.
- The facilitation of rapid development achieved mainly by the very small amount of code that developers need to write even for moderately complex applications
- The facilitation of independent, distributed development achieved by the support for peerlet protocols which define a clean interface between components on different machines.

Two additional benefits are:

- The facilitation of long-term modification of wearable community applications through a combination of a virtual machine architecture and high-level APIs.
- Interoperability among heterogeneous devices through the use of the network and programming language independent Proem Transport Protocol.

Second, this chapter discussed the differences of Proem to existing software development platforms. We concluded that such platforms either focus on a narrow problem domain (for example, data management) or provide generic support for a very large class of applications and that none provided an associated methodology for developing applications. Without a methodology, there is no support for conceptual design and specification of applications supported by these platforms. By integrating design, specification and implementation support, the WearCoM methodology and the Proem platform are uniquely equipped to facilitate rapid, yet systematic development of wearable community applications.

Third, this chapter explored limitations of the current Proem platform and presented two areas for future enhancements: trust management based on reputations and peer-to-peer application distribution.

Finally, we presented a set of preliminary design principles that attempt to capture aspects of successful wearable communities.

Chapter IX

CONCLUSION AND FUTURE WORK

In this final chapter, we summarize our research and discuss future research directions.

IX.1 Research Summary

The use of mobile communication and computation technologies in our society has reached the critical mass necessary to induce large-scale modifications of our social behavior, norms and conventions. With the move from mobile to wearable computers - devices that are constant, aware, communicative and proactive - we can expect to see changes that are even more dramatic. The general availability and wide-spread use of wearable technology will create new opportunities for computer-mediated communities. William Gibson once famously described cyberspace as "A consensual hallucination experienced daily by billions of legitimate operators, in every nation ... " (Gibson 1984, 51). Wearable communities represent an alternative model of human communication that instead of on shared imaginations relies on embodied real-world encounters and first-hand experiences.

The key challenge for wearable communities is that social and technical issues interact and co-evolve in such intimate ways that they often merge. The success or failure of communities hinges on its ability to promote lasting personal relationships. Yet without prior experience we cannot know how to design technology that leads to

the emergence of successful communities. While past research on online communities provides valuable insight into the dynamics of traditional computer-mediated communities, it is not clear if and in as much the results are applicable to wearable communities. Since there is no sound theoretical foundation for building wearable communities, we advocated an exploratory design process based on rapid prototyping and successive incremental refinement. An exploratory approach starts with an initial technology prototype that is refined over time in an incremental process based on the success or failure of emerging communities.

The engineering of wearable community systems - the hardware, system software and application software to support wearable communities - represents a non-trivial challenge. A wearable community system is a loosely coupled, dynamic, decentralized system composed of potentially large numbers of wearable devices. In order to support the formation of wearable communities anywhere at anytime, such systems must be independent of external communication and computing infrastructures, relying solely on the capabilities of devices carried by individuals. To tackle the complexity of such systems, we introduced the distinction between wearable community infrastructure and wearable community applications. Wearable community infrastructure comprises system-level software components that are useful and necessary for the support of a large variety of wearable communities. This infrastructure is generic in a sense that it embodies fundamental aspects of the wearable community domain. A set of requirements for wearable community infrastructure was discussed at the end of Chapter III. Wearable community applications are community-specific software applications that are built on top of and make use of functionality provided by wearable community infrastructure. Developing wearable community applications is a difficult task that requires highly specialized knowledge in a variety of fields. It ranges from understanding of human-factors issues related to new collaboration paradigms based on opportunistic, proximity-based interactions to technical issues related to expertise in ad hoc networking and context-awareness. The fundamental problem in the development of wearable community applications is the semantic gap between the

application domain and system layer. As of today, there is little or no direct support for the variety of features that wearable community applications require. Likewise, there are no appropriate programming and building abstractions for developers. This results in a lack of generality, requiring each new application to be built from the ground up. This situation is fundamentally incompatible with an exploratory design process based on rapid prototyping.

IX.2 Contributions

To address this problem, this dissertation made the following contributions:

1. The first contribution is the WearCoM wearable community methodology. Its purpose is to guide the design and development of wearable communities and wearable community applications. It consists of three components: (1) a conceptual model that defines terminology and an abstract architecture; (2) a design language that addresses the specification of important analysis and design decisions and enables developers to specify key aspects of the application design; and (3) a development process that outlines a sequence of development steps that result in the creation of specific artifacts.
   - The central concepts of the methodology are community agent, user profile, and community language. A community agent is a personal, proactive, presence-aware and communicative software application that functions as intermediary between a user and a community. A user profile is a typed data item that defines the identity of a user within a community. It contains information a user willingly discloses to other community members and may include any information that is defined useful or necessary to identify or describe a person. A community language defines interactions that can take place between agents and users during an encounter. The description of a community language consists of two components: (1) the community

vocabulary is a collection of messages types that actors can exchange (2) The community protocol defines message sequences.

- The wearable community design language is a semi-formal notation for specifying the key aspects of the design of wearable community applications. It provides the modeling language for: scenarios, user profile templates, community vocabularies and community protocols
- The purpose of the wearable community process is to guide the activities the lead to the implementation of a wearable community application and its deployment on user devices. The design process guides design and development activities ranging from the initial design to deployment of application software on a user's device. The process is an iterative, cyclic process divided into seven phases: assessment, conceptual design, specification, implementation, seeding, dissemination and review. Each phase defines specific activities to be performed by users, designers or developers; the outcome of the phase is a specific set of design artifacts.

2. The second contribution is the Proem peer-to-peer platform. Proem is designed to tightly integrate with the WearCoM methodology and provides infrastructure and development support required for wearable community applications. The main focus of Proem is the information needs of applications and the provision of high-level programming abstractions. The three main platform components are:

   - The Peerlet application framework, a collection of libraries and APIs for the rapid development of wearable community applications.
   - The Proem Runtime System, a software environment for hosting and executing applications built with the Proem framework.
   - The Proem protocols, a set of peer-to-peer protocols that define the way in which Proem peers communicate and cooperate over the network.

3. The final contribution of this research is the evaluation of the WearCoM methodology and Proem platform. Through several case studies involving design and implementation of sample wearable community applications and through a set of experiments in which we used Proem as development platform in software engineering courses, we were able to determine the key benefits of the Proem platform as follows:

   - Reducing the complexity of building applications: Proem simplifies the development task by raising the level of abstraction. This is achieved by three interrelated measures. First, the Proem platform provides network transparency and hides specifics of the underlying communication network from programmers. Second, it provides programmers with an application framework and a powerful set of APIs. Third, it realizes a simple programming model with a uniform event-based mechanism for peer-to-peer communication, presence-awareness and user interface management.
   - Enabling rapid development: Proem enables rapid prototyping through a combination of several measures. First, the amount of code that needs to be written for simple applications is small. This allows programmers to have a simple application up and running within a very short time frame and add to it gradually. Second, Proem supports code reuse through a virtual machine architecture and application plug-in modules (= peerlets). A complex application can be realized as a collection of communicating peerlets that run on the same machine within the same peerlet engine. Third, because of Proem's cross platform capabilities, developers can write and test applications on workstations and afterward deploy them on wearable computers.
   - Enabling independent, distributed development: Proem represents an open, distributed software infrastructure for wearable community applications. With Proem, multiple developers can define a community protocol and

create independent yet compatible implementations. This is made possible by two interrelated measures: first, by providing a common communication substrate in the form of the Proem Transport Protocol; second, by enabling developers to specify interfaces between devices in the form of peerlet protocols.

- Enabling long-term modification: Because of the virtual machine architecture and high-level APIs, applications are shielded from specifics of the underlying technologies. Modifications necessary due to innovations in communication and device technology can be hidden by the peerlet engine and are transparent to applications.
- Interoperability among heterogeneous device platforms: Finally, interoperability among heterogeneous devices is guaranteed by two interrelated measures. First, the cross-platform nature of the peerlet engine makes it possible to install and run existing applications on any devices with a peerlet engine. Second, the network independence of the XML-based Proem Transport Protocol makes it possible for two different Proem implementations to interoperate regardless of the programming language used to implement them.

By providing a common ground for the development and execution of wearable community applications, the Proem platform serves as a catalyst for wearable communities. The quality of the student projects – both in conceptual and technical terms – validates the overall effectiveness of the WearCoM methodology and the utility of the Proem platform.

IX.3 Future Research Directions

Future research falls into two categories:

- technical improvements and additions of the Proem peer-to-peer platform
- empirical research into social aspects of wearable communities

IX.3.1 Prototyping Environment

In Chapter VIII we discussed two shortcomings of the current Proem implementations: the lack of a trust framework and the inability to copy peerlets, i.e. community agents, from peer to peer.

Another area for future work is an even tighter integration of methodology and platform. In particular, we envision a unified graphical prototyping environment for wearable community applications. Such an environment would provide built-in support for the design language and development process and allow designers and developers to interactively specify an application design. The major new aspect of the environment would be the automatic generation of a community agent implementation from high-level specifications. With the current design language it would not be possible to generate a complete implementation because the language lacks the capability to express specific details about the internal behavior of community agents. Thus, we could either generate a partial implementation or leave it up to developers to fill in the missing parts or we could enrich the design language. The current language is purposefully simple, because it was designed to mainly address the aspects that are unique to wearable community applications. A prototyping environment, however, could easily make use of existing specification methods for more complex software behaviors. Two areas seem especially interesting: a tool for analyzing community protocols that is able to verify important protocol properties (such as liveliness); a tool

for specifying user interfaces for proactive application. Since the community language includes interactions between users and agents, these tools could as well be combined.

IX.3.2 Wearable Community Trial

The idea of wearable communities is a vision that in its totality has not yet been realized. The two primary reasons are: (1) there are not enough wearable computer users to reach the critical mass necessary for community building (2) there is a lack of development support for wearable community applications.

In this dissertation, we have addressed the second problem; the first one will become less of an issue as a technology progresses. As result of the current situation, wearable communities are confined to small groups of experimentally minded individuals at University or industry research labs. Although these technical and social circumstances might not perfect, we nevertheless need to address the question we raised early one: "What are the success criteria for wearable communities?" This question concerns the social behavior of individuals and can only by answered through controlled empirical evaluation of wearable communities. The challenges of such an undertaking are tremendous:

- The high mobility of individuals makes it impossible to observe social interactions in a restricted lab space.
- Participants of a study might be concerned about their privacy and modify their behavior while being observed.
- There are technical difficulties of automatically collecting data from a highly dispersed group of individuals.

These are challenges which must and are faced by any researcher trying to study communities, whether computer-mediated or not.

The result of such a study would provide insights into several crucial questions including:

- In which way are wearable communities different from or similar to online communities?
- In which way are wearable communities different from or similar to unmediated face-to-face communities?

The results of a wearable community trial could be condensed to a set of design principles. These principles may be similar to the ones discussed in Chapter VIII, yet they would be proven by empiry and would not rely on preliminary observations by developers. At the end, the results could also be used to enrich the WearCoM methodology with a set of built-in design rules. In any case, large scale deployment of wearable community systems and empirical evaluation of wearable communities are necessary prerequisites for the successful creation of wearable communities.

IX.4 Conclusion

This dissertation has described a research framework for development support of wearable communities consisting of a methodology and associated software platform. The methodology addresses the social and technical design of wearable communities and community applications, while the platform realizes a foundation for rapid implementation of applications. Together, the methodology and platform function as catalyst for the development of wearable communities.

BIBLIOGRAPHY

Abdul-Rahman, Alfarez and Stephen Hailes. 2000. In *Proceedings Hawaii International Conference on System Sciences 33*, Maui, Hawaii, 4-7 January 2000.

Acharya, A. and B.R. Badrinath. 1996. A Framework for Delivering Multicast Messages in Networks with Mobile Hosts. *ACM/Baltzer Mobile Networks and Applications* 1 (2):199-219.

Arora, A., Carl Haywood, Kuldip Singh Pabla. 2002. *JXTA for J2ME – Extending the Reach of Wireless with JXTA Technology*. Whitepaper, Sun Microsystems, Inc.

Bauer, Martin, Gerd Kortuem, and Zary Segall. 1999. "Where Are You Pointing At?" A Study of Remote Collaboration in a Wearable Video-Conference System. *Proceedings Third International Symposium on Wearable Computers* (ISWC99), 18-19 October, 1999, San Francisco, California.

Bauer, Martin, Timo Heiber, Gerd Kortuem, and Zary Segall. 1998. A Collaborative Wearable System with Remote Sensing. *Proceedings Second International Symposium on Wearable Computers* (ISWC98), Oct 19-20, 1998, Pittsburgh, PA.

Belotti, V. and S. Bly. 1996. Walking Away from the Desktop Computer: Distributed Collaboration and Mobility in a Product Design Team. *Proceedings ACM 1996 Conference on Computer Supported Cooperative Work* (CSCW'96), ACM Press.

Bergqvist, J., P. Dahlberg, F. Ljungberg and S. Kristoffersen. 1999. Walking Away from the Meeting Room: Exploring Mobile Meetings. *Proceedings European Conference on Computer Supported Cooperative Work* (ECSCW'99), Copenhagen, Denmark.

Billinghurst, M., J. Bowskill, M. Jessop and J. Morphett. 1998. A Wearable Spatial Conferencing Space. *Proceedings Second International Symposium on Wearable Computers* (ISWC'98), Oct 19-20, 1998, Pittsburgh, PA.

Billinghurst, M., S. Weghorst, and T. Furness. 1997. Wearable Computers for Three Dimensional CSCW. *Proceedings First International Symposium on Wearable Computers (ISWC'97), Cambridge, MA, October 13-14, 1997*. IEEE Press, Los Alamitos: 39-46.

Björk, S., J. Falk, R. Hansson and P. Ljungstrand. 2001. Pirates! - Using the Physical World as a Game Board. *Proceedings Interact 2001*, IFIP TC.13 Conference on Human-Computer Interaction, July 9-13, 2001, Tokyo, Japan.

Bluetooth SIG Special Interest Group. 2002. *The Bluetooth Specification*. Available at http://www.bluetooth.com/dev/specifications.asp, accessed 10 October 2002.

Bolcer, G. 2000. Magi: An Architecture for Mobile and Disconnected Workflow. *IEEE Internet Computing* 4, no 2. (May/June): 46-54.

Bolcer, G., Michael Gorlick, Arthur S. Hitomi, Peter Kammer, Brian Morrow, Peyman Oreizy, Richard N. Taylor. 2000. Peer-to-Peer Architecture and the Magi Open Source Infrastructure. Endeavours Technologies White Paper. Available at http://www.endeavors.com/pdfs/ETI%20P2P%20white%20paper.pdf.

Borovoy, R., B. Silverman, T. Gorton, J. Klann, M. Notowidigdo, B. Knep and M. Resnick. 2001. Folk Computing: Revisiting Oral Tradition as a Scaffold for Co-Present Communities. *Proceedings of the CHI 2001 Conference on Human Factors in Computing Systems*, New York, ACM Press.

Borovoy, R., F. Martin, S. Vemuri, M. Resnick, B. Silverman, and C. Hancock. 1998. Meme Tags and Community Mirrors: Moving from Conferences to Collaboration. *Proceedings 1998 ACM Conference on Computer Supported Cooperative Work* (CSCW'98), 1998, ACM Press.

Borovoy, R., M. McDonald, F. Martin, and M. Resnick. 1996. Things That Blink: Computationally Augmented Name Tags. IBM Systems Journal 35 (3&4).

Breese, J.S., D. Heckerman and C. Kadie. 1998. Empirical Analysis of Predictive Algorithms for Collaborative Filtering. *Proceedings 14th Conference on Uncertainty in Artificial Intelligence*, Madison WI, Morgan Kauffman.

Charas, Philippe. 2001. Peer-to-Peer Mobile Network Architecture. *Proceedings First International Conference on Peer-to-Peer Computing* (P2P2001), Aug 27-29, 2001, Linköpings, Sweden.

Che, T.-W. and M. Gerla. 1998. Global State Routing: A New Routing Scheme for Ad-hoc Wireless Networks. *Proceedings of the IEEE International Conference on Communications* (ICC), Atlanta, GA, June 1998, 171-175.

Clarke, I., O. Sandberg, B. Wiley, and T.W. Hong, Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Proceedings Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, 2001, LNCS 2009, Springer, New York.

CLIP2. 2002. *The Gnutella Protocol Specification, version 0.4*. Available at www9.limewire.com/developer/gnutella_protocol_0.4.pdf, accessed 10 October 2002.

CNN. 1998. Japan's lonely hearts find each other with 'Lovegety'. CNN online, availavle at http://www6.cnn.com/WORLD/asiapcf/9806/07/fringe/japan.lovegety/, accessed 10 October 2002.

Corson, M.S. and A. Ephremides. 1995. A Distributed Routing Algorithm for Mobile Wireless Networks. *ACM/Baltzer Wireless Networks 1* (1): 61-81.

Corson, M.S. and S.G. Batsell. 1995. A Reservation-Based Multicast (RBM) Routing Protocol for Mobile Networks: Initial Route Construction Phase. *ACM/Baltzer Wireless Networks 1 (4):*427-450.

Cost, R., T. Finin, Y. Labrou , X. Luan, Y. Peng, I. Soboroff , J. Mayfield and A. Boughannam. 1999. An Agent-based Infrastructure for Enterprise Integration. *Proceedings 1st International Symposium on Agent Systems and Applications* (ASA'99), ACM Press.

Dahlberg, P., H. Fagrell, and J. Redström. 1999. The News Pilot - Exploring Proximity Awareness. In *Proceedings Information Systems Research Seminar in Scandinavia* (IRIS 22), 7-10 August, Keuruu, Finland.

Dahlberg, P., J. Redström, and H. Fagrell. 1999a. People, Places and the NewsPilot. Proceedings of CHI *Conference on Human Factors in Computing Systems* (CHI'99), 1999, ACM Press.

Dahlberg, Per, Fredrik Ljungberg and Johan Sanneblad. 2001. Proxy Lady: Mobile Support for Opportunistic Interaction. *Scandinavian Journal of Information Systems* 13(1).

Danesh, A., K. Inkpen, F. Lau, K. Shu, and K. Booth. 2001. Geney: Designing a Collaborative Activity for the Palm Handheld Computer. *Proceedings of CHI, Conference on Human Factors in Computing Systems* (CHI 2001), Seattle, USA, April 2001, ACM Press.

Deering, S.E. and D.R. Cheriton. 1990. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems* 8(2):85-110.

Demers, A., K. Petersen, M. Spreitzer, D. Terry, M.M. Theimer, and B. Welch. 1996. The Bayou Architecture: Support for Data Sharing among Mobile Users. In *Proceedings Workshop on Mobile Computing Systems and Applications*, Santa Cruz, California, December 1994, IEEE Press.

Dourish, P. and S. Bly. 1992. Portholes: Supporting Awareness in a Distributed Work Group. *Proceedings of CHI, Conference on Human Factors in Computing Systems* (CHI 1992), Monterey, CA, 1992, ACM Press.

Edward T Hall. 1966. The Hidden Dimension. Garden City, N.Y.: Doubleday.

Edwards, W. Keith, Elizabeth Mynatt, Karin Petersen, Mike Spreitzer, Douglas Terry, and Marvin Theimer. 1997. Designing and Implementing Asynchronous Collaborative Applications with Bayou. *Proceedings 10th ACM Symposium on User Interface Software and Technology*, Banff, Alberta, Canada, October 1997.

Engelbart, Douglas. 1962. *Augmenting Human Intellect: A Conceptual Framework.* Research Report AFOSR-3223, Stanford Research Institute, Menlo Park, California.

European Telecommunications Standards Institute (ETSI). 2002. HiperLAN Specification. Available at http://www.etsi.org/, accessed 10 October 2002.

Feiner, S., B. MacIntyre, T. Höllerer, and T. Webster. 1997. A Touring Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment. *Proceedings First International Symposium on Wearable Computers* (ISWC '97), October 13-14, 1997, Cambridge, MA.

Finin, T. , R. Fritzson, D. McKay, and R. McEntire. 1994. KQML as an Agent Communication Language. Proceedings *3rd International Conference on Information and Knowledge Management* (CIKM94), December 1994, ACM Press.

Fish, R., R. Kraut, and B. Chalfonte, B. 1990. The VideoWindow System in Informal Communications. *Proceedings of ACM 1990 Conference on Computer-Supported Cooperative Work* (CSCW '90), Los Angeles, CA, 1990, 1-11.

Fish, R., R. Kraut, R. Root, and R. Rice. 1993. Video as a Technology for Informal Communication. *Communications of the ACM* 36(1):48-61, ACM Press.

Fitzpatrick, G., S. Kaplan, and S. Parsowith. 1998. Experience in Building a Cooperative Distributed Organization: Lessons for Cooperative Buildings. *Proceedings of First International Workshop on Cooperative Buildings* (CoBuild'98), 25.-26. February, 1998, GMD, Darmstadt, Germany, 66-79.

Fox, Kate. 2002. *Evolution, Alienation and Gossip - The Role of Mobile Telecommunications in the 21st Century*. Online report of the Social Issue Research Centre, 2002. Available at http://www.sirc.org/publik/gossip.shtml.

Garcia-Luna-Aceves, J.J. and M. Spohn. 1998. Scalable Link-State Internet Routing. *Proceedings of the IEEE International Conference on Network Protocols* (ICNP), Austin, TX, October 1998, 52-61.

Garcia-Luna-Aceves, J.J. and M. Spohn. 1999. Source-Tree Routing in Wireless Networks. *Proceedings of the IEEE International Conference on Network Protocols* (ICNP), Toronto, Canada, October, 1999, 273-282.

Gelernter, D. 1985. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems* 7(1): 80-112.

Genesereth, Michael R. and Steven P. Ketchpel. 1994. Software Agents. *Communications of the ACM* 37(7):48–53.

Gibson, William. 1984. *Neuromancer*. Ace Books.

Godwin, Mike. 1994. Nine Principles for Making Virtual Communities Work. *Wired Magazine* 2(6): 72-73.

Gold, R, and C. Mascolo. 2001. Use of Context-awareness in Mobile Peer-to-Peer Networks. *Proceedings 8th IEEE Workshop on Future Trends of Distributed Computing Systems* (FTDCS'01), Bologna, Italy, October/November 2001.

Gong, L. 2001. JXTA: A network programming environment. *IEEE Internet Computing* 5: 88-95.

Groove Networks. 2002. Groove collaboration software, available at http://www.groove.net/, accesses 10 October 2002.

Groten, D. and J.R. Schmidt. 2001. Bluetooth-based Mobile Ad Hoc Networks: Opportunities and Challenges for a Telecommunications Operator. In *Proceedings IEEE VTS 53rd Vehicular Technology Conference* (VTC 2001), 1134-1138.

Groundspeak Inc. 2002. Geocaching web site, available at http://www.geocaching.com/, accessed 10 October 2002.

Hall, Edward T. 1959. *The Silent Language*. New York, Doubleday.

Hall, Edward T. 1962. *Proxemics: The Study of Man's Spatial Relations*. Connecticut, International Universities Press.

Healey, J. and R. W. Picard. 1998. StartleCam: A Cybernetic Wearable Camera. *Proceedings Second International Symposium on Wearable Computers* (ISWC 98), Pittsburgh, PA, October 1998.

Höllerer, T., S. Feiner, and J. Pavlik. 1999. Situated Documentaries: Embedding Multimedia Presentations in the Real World. *Proceedings Third International Symposium on Wearable Computers* (ISWC'99), San Francisco, CA, October 18-19, 1999, 79-86.

Höllerer, T., S. Feiner, T. Terauchi, G. Rashid, and D. Hallaway. 1999. Exploring MARS: Developing Indoor and Outdoor User Interfaces to a Mobile Augmented Reality System. *Computers and Graphics* 23(6): 779-785.

Holmquist, Lars Erik, Jennica Falk, and Joakim Wigström. 1999. Supporting Group Collaboration with Inter-Personal Awareness Devices. *Personal Technologies* 3(1):13-21.

ID3. 2002. *id3v2 Specification*. Available at http://www.id3.org/, accessed 10 October 2002.

IEEE. 1999. *IEEE 802.11 Wireless Specification*, ISO/IEC 8802-11: 1999. Available at http://standards.ieee.org/getieee802/802.11.html, access 10 October 2002.

IEEE. 2002. *IEEE 802.15 Working Group for WPAN*, available at http://www.ieee802.org/15/, accessed 10 October 2002.

IETF. 2002. *IETF Mobile Ad-hoc Networks Working Group*. Home page available at http://www.ietf.org/html.charters/manet-charter.html, accessed 10 October 2002.

Ishida, T. 1998. *Community Computing: Collaboration over Global Information Networks*, John Wiley and Sons.

Ishida, T. 1998. Towards Communityware. *New Generation Computing* 16(1):5–21.

Johnson, D.B. and D.A. Maltz. 1996. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Mobile Computing*, ed. Tomasz Imielinski and Hank Korth, Chapter 5, Kluwer Academic Publishers.

Johnson-Lenz, Peter and Trudy Johnson-Lenz. 1981. Consider the Groupware: Design and Group Process Impacts on Communication in the Electronic Medium. In Hiltz, S. and Kerr, E. (eds.). *Studies of Computer-Mediated Communications Systems: A Synthesis of the Findings*, Computerized Conferencing and Communications Center, New Jersey Institute of Technology, Newark, New Jersey.

Johnson-Lenz, Peter and Trudy Johnson-Lenz. 1982. Groupware: The Process and Impacts of Design Choices. In Kerr, E. and Hiltz, S., *Computer-Mediated Communication Systems*, Academic Press, New York.

Jubin, J. and J. Tornow. 1987. The DARPA Packet Radio Network Protocols. In *Proceedings of the IEEE*, 75(1):21—32.

Kaminsky, Alan and Hans-Peter Bischof. 2002. Many-to-Many Invocation: A new object oriented paradigm for ad hoc collaborative systems. In *Proceedings 17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications* (OOPSLA 2002), November 2002.

Kaminsky, Alan. 2001. *Infrastructure for Distributed Applications in Ad Hoc Networks of Small Mobile Wireless Devices*. Technical Report, Rochester Institute of Technology, IT Lab.

Kan, Gene. 2001. Gnutella. In Andy Oram (ed.) *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, O'Reilly.

Kistler, J. J. and M. Satyanarayanan. 1992. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems* 10(1):3.

Kollock, Peter, and Marc Smith.1996. Managing Virtual Communities: Cooperation and Conflict in Computer Communities. In *Computer-Mediated Communication: Linguistic, Social, and Cross-Cultural Perspectives*, edited by Susan Herring. Amsterdam: John Benjamins, 109-128.

Kollock, Peter. 1998. Design Principles for Online Communities. *PC Update* 15(5): 58-60.

Korpela, Eric, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. 2001. SETI@home: Massively Distributed Computing for SETI. *Computing in Science and Engineering Magazine*, January 2001: 78-83.

Kortuem, Gerd and Segall, Zary. 2003. Building Wearable Communities. *IEEE Pervasive Computing* (to appear).

Kortuem, Gerd, Jay Schneider, Dustin Preuitt, Thaddeus G. Cowan Thompson, Stephen Fickas, Zary Segall. 2001. When Peer-to-Peer comes Face-to-Face: Collaborative Peer-to-Peer Computing in Mobile Ad-hoc Networks. In *Proceedings First International Conference on Peer-to-Peer Computing* (P2P2001), Aug 27-29, 2001, Linköpings, Sweden.

Kortuem, Gerd, Jay Schneider, Jim Suruda, Steve Fickas, and Zary Segall. 1999. When Cyborgs Meet: Building Communities of Cooperating Wearable Applications. *Proceedings Third International Symposium on Wearable Computers*, 18-19 October, 1999, San Francisco, California.

Kortuem, Gerd, Martin Bauer, Timo Heiber, and Zary Segall. 1999. Netman: The Design of a Collaborative Wearable Computer System. *ACM/Baltzer Journal on Mobile Networks and Applications* 4 (1).

Kortuem, Gerd. 2002. Proem: A Middleware Platform for Mobile Peer-to-Peer Computing. *ACM SIGMOBILE Mobile Computing and Communications Review* (MC2R), Special Feature on Middleware for Mobile Computing (to appear).

Kraut, R.E., R. Fish, R. Root, and B. Chalfonte. 1990. Informal Communication in Organizations: Form, Function, and Technology. In S. Oskamp and S. Scacapan (Eds.) *Human Reactions to Technology: Claremont Symposium on Applied Social Psychology*. Beverly Hills, CA, Sage Publications.

Kraut, R.E., V. Lundark, S. Kiesler, T. Mukopadhyay, and W. Scherlis. 1998. Internet Paradox: A Social Technology that Reduces Social Involvement and Psychological Well-being? *American Psychologist* 53:1017-1031.

Lee, Seungjoon and Chongkwon Kim. 2000. Neighbor Supporting Ad Hoc Multicast Routing Protocol. In *Proceedings MobiHoc 2000*, August 11, 2000, Boston, Massachusetts, USA.

Leiner, B. and D. Nielson. 1987. Issues in Packet Radio Network Design. In *Proceedings of the IEEE,* Special Issue on Packet Radio Networks, 75(1):6-20.

Licklider, Joseph Carl Robnett. 1960. Man-Computer Symbiosis. *IRE Transactions on Human Factors in Electronics*, Volume HFE-1: 4–11.

Licklider, Joseph Carl Robnett. 1968. The Computer as a Communication Device. *Science and Technology*, April.

Luff, Paul and Christian Heath. 1998. Mobility in Collaboration. In *Proceedings 1998 ACM Conference on Computer Supported Cooperative Work* (CSCW'98), 305-314, ACM Press.

Maltz, D.A. , J. Broch, J. Jetcheva, and D.B. Johnson. 1999. The Effects of On-Demand Behavior in Routing Protocols for Multihop Wireless Ad Hoc Networks. *IEEE Journal on Selected Areas in Communications*, Special Issue on Wireless Ad Hoc Networks, 17(8):1439-1453.

Mann, Steve. 1996. Smart Clothing: Wearable Multimedia Computing and Personal Imaging. In *Proceedings of ACM Conference on Multimedia*, November 1996, 163-174.

Mann, Steve. 1997. Wearable Computing: A First Step Toward Personal Imaging. *IEEE Computing* 30(2).

Mann, Steve. 2001a. Humanistic Intelligence/Humanistic Computing: 'Wearcomp' as a New Framework for Intelligent Signal Processing. *IEEE Intelligent Systems and Their Applications* 16 (3), Special Issue on Wearable Computing and Humanistic Intelligence.

Mann, Steve. 2001b. *Intelligent Image Processing*. John Wiley and Sons.

Martin D., A. Cheyer, and D. Moran. 1999. The Open Agent Architecture: A Framework for Building Distributed Software Systems. *Applied Artificial Intelligence: An International Journal*. 13(1-2):91-128.

Mascolo, Cecilia, Licia Capra and Wolfgang Emmerich. 2001. An XML-based Middleware for Peer-to-Peer Computing. In *Proceedings of the First International Conference on Peer-to-Peer Computing* (P2P2001). Linkopings, Sweden, Aug 2001.

Mascolo, Cecilia, Licia. Capra, S. Zachariadis, and Wolfgang Emmerich. 2002. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. In *Personal and Wireless Communications Journal 21 (1)*, Kluwer.

Meetup, LCC. 2002. Meetup web site, available at http://www.meetup.com, accessed 10 October 2002.

Minar, Nelson, Matthew Gray, Oliver Roup, Raffi Krikorian, and Pattie Maes 1999. Hive: Distributed Agents for Networking Things. In *Proceedings of the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents* (ASA/MA'99).

Misztal, B. 1996. *Trust in Modern Societies*. Polity Press, Cambridge MA.

Moats, R. 2001. URN Syntax. RFC2141. Available at http://www.ietf.org/rfc/rfc2141.txt, accessed 10 October 10.

Murphy, Amy L. 2000. *Enabling the Rapid Development of Dependable Applications in the Mobile Environment*. Washington University Technical Report WUCS-00-15, Sever Institute of Washington University.

Murphy, Amy L., Gian Pietro Picco and Gruia-Catalin Roman. 2001. Lime: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems* (ICDCS-21), Phoenix, AZ, USA, 16-19 April 2001, 524-233.

Murthy, S. and J.J. Garcia-Luna-Aceves. 1996. An Efficient Routing Protocol for Wireless Networks. *ACM/Baltzer Mobile Networks and Applications* 1(2):183-197.

Myers, B.A., H. Stiel, and R. Gargiulo. 1998. Collaboration using Multiple PDAs Connected to a PC. In *Proceedings 1998 ACM Conference on Computer Supported Cooperative Work* (CSCW'98), ACM Press, 285-294.

Mynatt, E. D., A. Adler, M. Ito, and V.L. O'Day. 1997. Design for Network Communities. In Proceedings of *CHI Conference on Human Factors in Computing Systems* (CHI'97), 22-27 March 1997, Atlanta, GA, ACM Press.

Nakanishi, H., C. Yoshida, T. Nishimura, and T. Ishida. 1996. FreeWalk: Supporting Casual Meetings in a Network. In *Proceedings 1996 ACM Conference on Computer Supported Cooperative Work* (CSCW'96), ACM Press.

Nash, John F. 1950. The Bargaining Problem. *Econometricia* 28:155-162.

Nasipuri, A. and S.R. Das. 1999. On-Demand Multipath Routing for Mobile Ad Hoc Networks. In *Proceedings of the IEEE International Conference on Computer Communications and Networks* (ICCCN), Boston, MA, October 1999, 64-70.

NCL's National Fraud Information Center and Internet Fraud Watch, 1998. *Internet Fraud Watch Online Auction Tip*. Available at http://www.fraud.org/news/1998/nov98/111698.htm, accessed 10 October 2002.

Nichols D.M. 1998. Implicit Rating and Filtering. In *Proceedings Fifth DELOS Workshop on Filtering and Collaborative Filtering*, Budapest, Hungary, 10-12 November 1997, 31-36.

Nidd, Michael. 2000. Timeliness of Service Discovery in DEAPspace. In *Proceedings of the 2000 International Workshop on Parallel Processing*, August 21 - 24, 2000, Toronto, Canada, 73-81.

Object Management Group (OMG). 2001. *Unified Modeling Language Specification*, Version 1.4. September 2001.

Olson, Mancur. 1965. *The Logic of Collective Action: Public Goods and the Theory of Groups*. Cambridge, MA, Harvard University Press.

Ostrom, Elinor. 1990. *Governing the Commons: The Evolution of Institutions for Collective Action*. New York, Cambridge University Press.

Pei, G., M. Gerla, and T.-W. Chen. 2000. Fisheye State Routing: A Routing Scheme for Ad Hoc Wireless Networks. In *Proceedings of the IEEE International Conference on Communications* (ICC), New Orleans, LA, June 2000, 70-74.

Pei, G., M. Gerla, and X. Hong. 2000. LANMAR: Landmark Routing for Large Scale Wireless Ad Hoc Networks with Group Mobility. In *Proceedings of the ACM/IEEE Workshop on Mobile Ad Hoc Networking and Computing* (MOBIHOC), Boston, MA, August 2000, 11-18.

Pering, T. and C. Pering. 2001. Mercantile: Social Interaction Using a Mobile Computing Platform. In *Proceedings Ubicomp 2001 Workshop on Designing Ubiquitous Computing Games*, Atlanta, GA, USA, 30 September 2001.

Perkins, C.E. and E.M. Royer. 1999. Ad-Hoc On Demand Distance Vector Routing. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications* (WMCSA), New Orleans, LA, February 1999, 90-100.

Perkins, C.E. and P. Bhagwat. 1994. Highly Dynamic Destination-Sequenced Distance Vector Routing (DSDV) for Mobile Computers. In *Proceedings of the ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications*, London, UK, September 1994, 234-244.

Picard, R. W. and J. Healey. 1997. Affective Wearable. *Personal Technologies* 1(4): 231-240.

Picco, Gian Pietro, Amy L. Murphy, and Gruia-Catalin Roman. 1999. Lime: Linda Meets Mobility. In *Proceedings of the 21stInternational Conference on Software Engineering* (ICSE'99), Los Angeles, May 1999.

Pocit Labs. 2001. *BlueTalk*, available at www.pocitlabs.com, accessed 10 June 2001.

Ponnekanti, Shankar R., Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. 2001. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *Proceedings Ubiquitous Computing Conference* (UBICOMP 2001), Atlanta, Georgia.

Prakash, Ravi. 1999. Unidirectional Links Prove Costly in Wireless Ad Hoc Networks, In *Proceedings 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications* (DIAL-M), August 1999, Seattle.

Preece, J. 1999. What happens after you get online? *Information Impacts Magazine*, December.

Preece, J. 2000. *Online Communities: Designing Usability, Supporting Sociability*. Chichester, UK, John Wiley & Sons.

Pursley, M.B. and H.B. Russell. 1993. Routing in Frequency-Hop Packet Radio Networks with Partial-Band Jamming. *IEEE Transactions on Communications* 41(7):1117-1124.

Randell, C. and H. Muller. 2000. The shopping Jacket: Wearable Computing for the Consumer. In *Proceedings Second International Symposium on Handheld and Ubiquitous Computing*, Bristol, UK, September 2000.

Rekimoto, J., Y. Ayatsuka and K. Hayashi. 1998. Augment-able reality: Situated Communications through Physical and Digital Spaces. In *Proceedings of the 2nd International Symposium on Wearable Computers* (ISWC'98), Pittsburgh, PA, October 1998, 68-75.

Resnick P., N. Iacovou, M. Suchak, A. Bergstrom, and J. Riedl. 1994. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *Proceedings of ACM 1994 Conference on Computer Supported Cooperative Work* (CSCW'94), Chapel Hill, NC, ACM Press, 175-186.

Rheingold, Howard. 1993. *The Virtual Community: Homesteading on the Electronic Frontier*. Perseus Books.

Rhodes, Bradley. 1997. The Wearable Remembrance Agent: A System for Augmented Memory. *Personal Technologies* 2(1):218-224.

Rosenschein, Jeffrey S and Giliad Zlotkin. 1994. *Rules of Encounter, Designing Conventions for Automated Negotiation among Computers.* MIT Press, Cambridge Massachusetts.

Roth, Jörg (2002). Mobility Support for Replicated Real-time Applications. In *Proceedings Innovative Internet Computing Systems* (I2CS), Kühlungsborn, June 2002, LNCS 2346, Springer,

Roth, Jörg and Claus Unger. 1998. DreamTeam - A Platform for Synchronous Collaborative Applications. In *Proceedings Groupware und Organisatorische Innovation* (D-CSCW'98), B. G. Teubner Stuttgart, Leipzig, 153-165.

Salonidis, Theodoros, Pravin Bhagwat, Leandros Tassiulas, and Richard LaMaire. 2001. Distributed Topology Construction of Bluetooth Personal Area Networks. In

Proceedings *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies* (INFOCOM 2001) 22-24 April 2001, Anchorage, Alaska.

Salonidis, Theodoros, Pravin Bhagwat, Leandros Tassiulas, and Richard LaMaire. 2001. *Proximity Awareness and Ad Hoc Network Establishment in Bluetooth.* Technical Report CSHCN 2001, Center for Satellite and Hybrid Communication Networks, University of Maryland.

Schneider, Jay and Gerd Kortuem. 2001. How to Host a Pervasive Game: Supporting Face-to-Face Interactions in Live-Action Roleplaying. In *Proceedings Ubicomp 2001 Workshop on Designing Ubiquitous Computing Games*, 30 Sept. – 2 Oct. 2001, Atlanta, Georgia.

Schneider, Jay, Gerd Kortuem, Joe Jager, Steve Fickas, Zary Segall. 2000. Disseminating Trust Information in Wearable Communities. In *Proceedings Second International Symposium on Handheld and Ubiquitous Computing* (HUC2K), 25-27 Sept 2000, Bristol, England.

Segall, Zary, Andrew Fortier, Gerd Kortuem, Jay Schneider, Sean Workman. 2002. Multishelf: An Experiment in Peer-to-Peer Infomediation. In *Proceedings First International Conference on Peer-to-Peer Computing* (P2P2001), August 27-29, 2001, Linköpings, Sweden.

Shardanand, U. and Pattie Maes. 1995. Social Information Filtering: Algorithms for Automating "Word of Mouth". In *Proceedings of the CHI 1995 Conference on Human Factors in Computing Systems*, ACM Press.

Sharman Networks. 2002. *The KaZaA Media Desktop*. Available at http://www.kazaa.com/en/index.php. Accessed on 10 October 2002.

Siewiorek, D., A. Smailagic, L. Bass, J. Siegel, R. Martin, and B. Bennington. 1998. Adtranz: A Mobile Computing System for Maintenance and Collaboration. In *Proceedings Second International Symposium on Wearable Computers*, Pittsburgh, PA, October 1998, IEEE Computer Society Press.

Siewiorek, D.P. and A. Smailagic. 1994. The CMU Mobile Computers as Maintenance Assistants. In *Proceedings of the NSF Workshop on Mobile and Wireless Information Systems*, Rutgers University, N.J., October 1994.

Smailagic, A. and D.P. Siewiorek. 1994. The CMU Mobile Computers: A New Generation of Computer Systems. In Proceedings of COMPCON '94, San Francisco, 28 August – 3 September 3, 1994, IEEE Computer Society Press.

Smailagic, A. and R. Martin. 1997. Metronaut: A Wearable Computer with Sensing and Global Communication Capabilities. In *Proceedings First International Symposium on Wearable Computers*, Boston, MA, October 1997, IEEE Computer Society Press.

Smailagic, A., D. Siewiorek, R. Martin, and J. Stivoric. 1998. Very Rapid Prototyping of Wearable Computers: A Case Study of VuMan 3 Custom versus Off-the-Shelf Design Methodologies. *Journal on Design Automation for Embedded Systems* 3(2/3): 217-230.

Starner, Thad, Joshua Weaver, and Alex Pentland. 1997. A Wearable Computing Based American Sign Language Recognizer. In *Proceedings First International Symposium on Wearable Computers* (ISWC '97), 13-14 October 13-14 1997.

Starner, Thad, Steve Mann, Bradley Rhodes, Jeffrey Levine, Jennifer Healey, Dana Kirsch, Rosalind W Picard and Alex Pentland. 1997. *Augmented Reality Through Wearable Computing*. Technical Report No. 397, M.I.T Media Laboratory, Massachusetts Institute of Technology, Cambridge, MA,

Starner, Thad. 1999. *Wearable Computing and Contextual Awareness*. Ph.D. Dissertation, Massachusetts Institute of Technology.

Starner, Thad. 2001. The Challenges of Wearable Computing. *IEEE Micro* 21(4): 44-52, 21(5): 54-67.

Stewart, J., B. Bederson, and A. Druin. 1999. Single Display Groupware: A Model for Co-present Collaboration. In *Proceedings 1999 Conference on Human Factors in Computing Systems* (CHI 99), ACM Press.

Sumi, Y. and K. Mase. 2001. AgentSalon: Facilitating Face-to-Face Knowledge Exchange through Conversations Among Personal Agents. In *Proceedings of Agents'01*, Montreal, CA, 393-400.

Tanenbaum, A. 1996. *Computer Networks* (3rd Edition). Prentice Hall, Upper Saddle River, N.J.

Terry, Michael, Elizabeth D. Mynatt, Kathy Ryall, Darren Leigh. 2002. Social Net: Using Patterns of Physical Proximity Over Time to Infer Shared Interests. In *Proceedings 2002 Conference on Human Factors in Computing Systems* (CHI 2002), ACM Press.

Toh, C.-K. 1997. Associativity-Based Routing for Ad-Hoc Mobile Networks. *Wireless Personal Communications Journal* 4(2):103-139.

UWBWG. 2002. Ultra-Wideband Working Group web site, available at http://www.uwb.org, accessed 10 October 2002.

Wenger, E. 1998. *Communities of Practice: Learning, Meaning, and Identity.* Cambridge, UK, Cambridge University Press.

Whittaker, S., D. Frohlich, and O. Daly-Jones. 1994. Informal Workplace Communication: What is it Like and How Might We Support it? In *Proceedings 1994 Conference on Human Factors in Computing Systems* (CHI 1994), ACM Press.

Whitten, I. H., H.W. Thimbleby, H. G. Coulouris, S. Greenberg. 1991. Liveware - A new Approach to Social Networks. In S. Greenberg (eds.), *Computer-supported Cooperative Work and Groupware*, University Press, Cambridge, UK, 211-222.

Wu C.W. and Y.C. Tay. 1999. AMRIS: A Multicast Protocol for Ad hoc Wireless Networks. In *Proceedings of the IEEE Military Communications Conference* (MILCOM), Atlantic City, NJ, November 1999, 25-29.

Wyckoff, P. 1998. TSpace. *IBM Systems Journal* 37(3).

Záruba, Gergely V., Stefano Basagni, and Imrich Chlamtac. 2001. Bluetrees - Scatternet Formation to Enable Bluetooth-Based Ad Hoc Networks. In *Proceedings ICCS*, 28-30 May 2001, San Francisco, CA, USA.

Zimmermann, P. 1995. *The Official PGP User's Guide*. MIT Press.