UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING AND APPLIED SCIENCE

School of Electronics and Computer Science

Mobile-Agent based Middleware for Mobile Users

by

Norliza Mohamad Zaini

A thesis submitted for the degree of Doctor of Philosophy

May 2005

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE
SCHOOL OF ELECTRONICS & COMPUTER SCIENCE

Doctor of Philosophy

MOBILE-AGENT BASED MIDDLEWARE FOR MOBILE USERS

by Norliza Mohamad Zaini

Mobile terminals such as cellular phones and PDAs have rapidly come into widespread use in recent years. Consequently, there is a growing requirement for increased flexibility in applications, especially in overcoming limitations of wireless environments. However, developing distributed applications that make effective use of networked resources from a mobile terminal is difficult for a number of reasons. We focus on two main problems: namely, mobile terminals do not have a permanent connection, and have intermittent connectivity to the fixed network; and the nature of a mobile terminal is the limiting factor when supporting computation-intensive applications. Such characteristics prevent the large-scale deployment of advanced services to a mobile terminal. Here, we consider an application offering collaborative editing support for mobile users. The coordination model that drives the application keeps track of and coordinates activities performed by mobile collaborators. The application's architecture is designed so as to allow documents hosted on mobile terminals, to be shared and edited in parallel by mobile collaborators. The collaborative editing application is a complex application requiring communication, memory and computing resources, and does not lend itself to a port to mobile terminals with limited resources and intermittent connectivity. For this reason, we decided to offload the computationally-intensive part of the application to the infrastructure, and to introduce the idea of an intermediary located in the network infrastructure, interacting with applications on behalf of the mobile terminal, thereby hiding away evidence of the intermittent connectivity. Our vision is that of a mobile agent, called a shadow that is always in close vicinity of the mobile terminal. We show that multiple shadows may co-exist, and we propose a protocol capable of coordinating them. We present a middleware application called Mobile-Agent based Middleware for Mobile Users (MAMiMoU), which hides away communication and coordination details, and offers a substrate for building the distributed collaborative editing application across mobile terminals and fixed infrastructures. Implementation details of our application are also presented. We undertake a systematic evaluation of our architecture by using a testing environment that simulates different collaboration patterns. Our conclusion is that when a system is supported by MAMiMoU in mobile users' environments, it performs significantly better than such a system without MaMiMoU.

Contents

List of figures

Acknowledgements

I would like to thank Prof. Luc Moreau for his excellent supervision and support in this research. A sincere gratitude to the QinetiQ and EPSRC for funding the project. Thanks to Dr. Paul Lewis for his guidance and advice during the evaluation phase of the research. Thank you also to Dr Bruce Hunt and Richard Lawley who have helped proof-read this thesis. I would like to mention all the people I have worked with over the course of my PhD, including Jing Zhou, Victor Tan and Wei Yan Zheng. Special thanks go to my family, especially to my parents, my grandparents, my brothers and Fuad, and to my friends, especially to Nor Aniza, Zuaini, Nadia, Shima, Rosmila, Jing, Aniza and Arouna. Many thanks also to all the IAM members that have helped and supported me over the last four years.

Above all, I would like to thank God who makes all things possible.

I dedicate this thesis to my parents, my brothers, my husband-to-be and in loving memory of my grandparents.

Chapter 1

Introduction

In the past few years, we have seen unprecedented growth in the number of mobile users, applications and wireless network access technologies. More and more users travel with their laptops or portable devices, accessing the Internet at a variety of places including their homes, workplaces and public places, e.g. in shopping malls, cafes, airports or on public transportation such as trains. This shows how communications and work activities can now take place anytime, anywhere. This "online anytime and anywhere" trend is supported by technology that is becoming more advanced. Wireless and portable communication devices continue to provide newer and better ways for mobile users to communicate with other people and access online services, while at the same time fixed online services are also being expanded to support mobile and wireless users. With the parallel incremental use of mobile terminals and advances in supporting technologies, a new type of computing, called mobile computing, has emerged [50].

Mobile computing focuses on supporting computing on the move and has the goal of providing users with access to online services and communication supports in a mobile, sometimes wireless, environment. Briefly, there are three different types of wireless data networks, namely wireless personal area networks (WPAN), wireless local area networks (WLAN), and wireless and wide area networks (WWAN). Wireless Personal Area Networks are networks that follow a short-range protocol (10 meters) and operate at 1 Mbps [44]. Bluetooth technology is an example of wireless personal area networking (WPAN) technology that has gained significant industry support and coexists with most wireless LANs. A wireless LAN (WLAN) is an on-premises data communication system that reduces the need for wired connections and makes new applications possible, thereby adding new flexibility to networking [127]. WWANs are based on several existing technologies, namely cellular, Satellites, Wireless Local Loops (WLL), Wireless Asynchronous Transfer Mode (ATM) and Mobile Internet Protocol (IP). WWAN technologies

include the so-called 2.5 and 3G technologies, e.g. General Packet Radio Service (GPRS) and Universal Mobile Telecommunications System (UMTS), which give rise to new mobile devices providing fast and immediate (i.e. "always connected") access to the Internet [66]. In the near future there is the interesting prospect of technologies that promise broader coverage and higher speed of connection to the network being continuously deployed.

Even though high-speed wireless technologies are widely used and newer technologies are emerging, there are still some limiting factors that cannot be avoided. These factors include i) poorer quality of connections to the fixed network due to lower bandwidth and more intermittent connectivity; ii) users' mobility, i.e. frequent changes to the point of attachment to the network that may result in interruptions in the connection; and iii) the limited processing power, memory and battery power of terminals. These limiting factors typically result in slower download times and more frequent failures being raised by applications hosted on mobile terminals when such applications try to communicate with the fixed network. Taken as a whole, such limiting factors may prevent or hinder large-scale deployment of advanced applications in mobile users' environments, as these tend to be communication and computation intensive.

1.1 The scenario

To focus the discussion, we adopt a scenario by making use of such a wireless environment. The fixed infrastructure provides a vast number of high-level services for users, such as information databases, location maps, and interfaces to intelligent devices, e.g. sensors, displays, brokering services and collaboration supports. The users of these services include mobile users who may move from one location to another. One such scenario is described as follows:

Two researchers are attending a project meeting that takes place at another university. During the meeting, information related to the meeting agenda is shared between the project members in an ad-hoc manner, possibly by accessing shared documents from a local repository. The shared documents can be cached onto mobile terminals by the members for future reference or use. In some cases, such shared documents will be changed by some members, after which the updated documents are republished to all. Such collaborative editing can be continued even after the meeting, in which case the two researchers may continue the collaboration remotely using resources at their own university, possibly exploiting local and remote repositories. This type of

scenario is described in [71] as the "meeting room scenario" and will be our motivation in designing a collaborative editing application for mobile users (cf. Chapter 6).

In supporting such a scenario, we present the challenges involved, followed by the approaches we take in addressing the challenges.

1.2 Challenges

Even though high-speed wireless connections such as WLAN and 3G technologies are more prevalent nowadays, lower quality cellular wireless technologies such as GSM are still used by many mobile users [95][62]. Although when using fast wireless connections users may sometimes need to operate in a lower bandwidth mode, i.e. when they move further away from a base station and the signal has dropped, mobile users with such low bandwidth connectivity commonly experience higher error rates than others. Low bandwidth connectivity prevents communication-intensive applications from working properly, in which case the slow downloading time of data sometimes leads to the failure of an application to continue serving the user's requests. Furthermore, with unreliable wireless service coverage, a mobile terminal may have to tolerate variable bandwidths ranging from several Mbps to a fraction of kbps.

Some wireless communication services are unreliable in terms of availability, where mobile users may not be within the service coverage area of a network, thus making the network unavailable to them. Service coverage mainly depends on the output power of the transmitter, its location and the frequency used to transmit data. For each particular application, throughput decreases as distance from the transmitter or access point increases. Based on these conditions mobile users will experience intermittent connectivity to the network. In such a situation, messages sent by a mobile terminal would fail, which could lead to a timeout in the mobile terminal. Besides this, the synchronous invocation method, which is a common interaction model in the distributed environment, is inadequate for mobile systems since the client needs to wait for a reply after each request, which is difficult in an environment where high latency and disconnected operations occur more often [3]. A challenging task in such a situation is to maintain interactions, possibly asynchronous, which are established between applications hosted on the mobile terminal and network-based applications.

Looking from another perspective, a mobile user, whether accessing the network either by wired or wireless means, will no longer be stationary. Mobile users are commonly connected to the network from different locations and this raises complexity in maintaining interactions between mobile terminals and network-based applications. Not only are such interactions interrupted when users lose their connections, but once reconnected from a new location, the network-based applications will have problems in recognizing the mobile terminal's applications. From this perspective, the challenge is to maintain such associations and interactions between mobile terminal and network-based applications while users roam across the network. Supporting this would require routing and keeping track of users' current locations, which may involve a mobile device generating a location update when it detects that it has been re-connected to a network. Users' mobility can cause network instability [55], which may result in a mobile terminal operating in disconnected mode. In other cases, a user may voluntarily decide to connect to the network only periodically.

Internet-enabled cell phones, Personal Digital Assistants (PDAs) and smartphones have emerged as the newest products that can connect to the Internet across a digital wireless network. Such small devices most likely have limited processing and power capabilities compared to desktop computers. High utilization of devices' battery power prevents excessive use of computation- and communication-intensive tasks. The small display areas of these devices is also a limiting factor, preventing the display of complex and friendlier user interfaces. Security is also a greater concern in wireless systems than in wired systems, since information may be traveling in free space. Based on these limitations, the majority of current Internet content is not optimized for these devices; at present, only email, stock quotes, news, messages, and simple transaction-oriented services are commonly used. Besides that, in terms of security issues, most wireless devices are not capable of handling strong encryption applications [44].

Despite the aforementioned limitations, an important subset of emerging activities is collaborative, i.e. they involve mobile users who share and collaboratively edit documents. In practice, mobile users would upload documents on their mobile terminals in order to be able to work on them while they are disconnected; thus, such documents, which we refer to as *mobile documents*, benefit from the physical mobility of mobile terminals, but fail to remain accessible by other users due to the frequent disconnections of mobile terminals from the network. Our aim is to be able to support such collaborative editing practices between mobile users by providing solutions that reduce the impact of the aforementioned limitations in mobile users' environments.

Mobile collaborators may not all be connected at the same time, due to temporary disconnection from the Internet. Thus, the challenge is to support asynchronous collaboration between users.

1.3 Research Aims and Approaches

In this section, we outline our research aims and approaches to two main categories; the first category deals with limitations in mobile users' environments, while the second category deals with challenges in supporting collaborative editing of mobile documents between mobile users.

1.3.1 Addressing limitations in mobile users' environments

Our aim is to address the aforementioned limitations and challenges in mobile users' environments and to support mobile document sharing and collaborative editing between mobile users. We now break down this aim into smaller targets and approaches as shown below.

Firstly, we want to address the low bandwidth problem and our approach is to minimize the number of message exchange via wireless medium between mobile terminal applications and network-based applications. This is done by offloading the communication-intensive parts of the mobile terminal's applications onto the fixed network. Messages can then be exchanged by such offloaded application parts with network-based applications via a wired communication medium that can reliably support their interactions.

Secondly, we want to address problems due to users' frequent disconnections from the fixed network. To do this, we envision an application with autonomous behaviour being hosted on the local network, which can be used to maintain associations and interactions between mobile terminal applications and network-based applications. Additionally, this application can act on behalf of a mobile user while the user is disconnected from the network. Synchronous interactions are not suitable when the mobile terminal experiences frequent disconnections. In this case, support for asynchronous interactions is a solution, where a mobile terminal issues a request and continues operating, then collects the result at an appropriate time [3]. Such a model decouples the act of communication from the actual time a service provider on the network requires to produce and deliver results to the mobile terminal, allowing the service provider to

make progress when the mobile terminal is disconnected. Such a technique therefore has the potential to mask some of the network failures experienced by mobile terminals [27]. To adopt such asynchronous interactions, network-based applications can first send messages to the autonomous application, which will forward the messages to the mobile terminal once the mobile terminal reconnects. The autonomous application can be regarded as a proxy or a representative of a mobile user on the fixed network, which has a store-and-forward mechanism to support message exchange between mobile terminal applications and network-based applications.

Thirdly, we want to address problems raised by users' mobility, which we approach by hiding it from mobile terminal and network-based applications. To do this, we can provide an abstraction layer that transparently maintains interactions between mobile terminal and network-based applications. The abstraction layer is mainly hosted on the fixed network, which is able to locate and establish a new connection with the mobile terminal once the mobile terminal moves and is reattached from a new location. This way, network-based applications that are interacting with applications on the mobile terminal are not required to keep track of the mobile terminal's new location in order to continue their interactions with the mobile terminal's applications. Being supported by the abstraction layer, i.e. using the store and forward mechanism; such network-based applications need only to maintain their interactions with the abstraction layer, which is acting as a representation of the mobile terminal's applications on the fixed network. In parallel, mobile terminal applications are offered the same service, in which case they can interact with the infrastructure, and find and exploit services [122] to fulfill the user's needs for network-based applications without having to be aware of the connection status of the mobile terminal. Supporting this, one part of the abstraction layer is hosted on the mobile terminal, corresponding to the rest of the abstraction layer on the fixed network once the mobile terminal is connected.

In recent years, Mobile IP [86] has been standardized by the IETF to provide users with the freedom to roam beyond their home subnet whilst consistently maintaining their home IP address. Mobile IP supports tunneling of messages for a mobile terminal while connected to the fixed network, but it does not provide a store and forward mechanism for messages. We propose that this would be a useful solution for supporting disconnected mobile terminals. Our work does not intend to replace such a protocol, but to further assist in allowing the seamless movement of mobile terminals, where we aim to allow mobile terminals to join and leave networks and sub-networks without explicit actions on the part of the users, specifically in re-establishing interactions between mobile terminal and network-based applications. Our work also does not

involve providing network-level support allowing mobile terminals to roam across several different wireless and mobile networks, such as a service presented in [116], but our work can be supported by such a service.

Fourthly, to address mobile terminal limitations, our approach is to allow the computation- and communication-intensive components of mobile terminal applications to be offloaded onto the fixed network, leaving only the light-loaded components to be hosted on the mobile terminal. Offloaded components have the advantages of accessing high capability resources, e.g. high computation power and large memory, while having only the light-loaded components of the application left on the device will save the mobile terminal's resources. Such a design decision by application developers helps to overcome the inherent imbalances of *i)* processing power between mobile terminal and network-based processors and *ii)* uplink and downlink bandwidth due to the transmission power available from the wireless mobile terminal [57], in which case as bandwidth increases, power consumption increases. Wireless uplink refers to the bandwidth used to transfer data from the mobile terminal to the fixed network, while downlink is the opposite of this.

Given these aims, this thesis combines the approaches needed to address the problems by realizing them in a middleware application for mobile users. With all these approaches translated into appropriate functionalities, the middleware, once implemented, can be used to support both mobile terminal and network-based applications. This middleware will act as an abstraction layer that hides the complexity in the interactions between mobile terminal and network-based applications. In the mobile environment, the user's locality, i.e. a particular operating environment or network neighbourhood to which the user's mobile terminal is attached, which offers resources and services locally; changes as the user moves. Thus, if a middleware application serving a user is local and stationary, over time it may cease to be locally available to the user, due to the users migrating elsewhere. The communication path can grow disproportionately to actual movement since it may traverse more intermediaries, e.g. a small movement can result in a much longer path when crossing network administrative boundaries [27]. Such a long network path may result in longer latency and greater risk of disconnection, and consumes more network capacity, even though the bandwidth between the mobile terminal and the middleware may not degrade. To avoid these disadvantages, we allow the middleware to dynamically serve users locally, i.e. to operate in their vicinity, for example being connected to the same local network.

To realize this, we employ mobile agents as the main component of the middleware, which are defined as running programs that autonomously decide to change location in order to continue their execution in an environment with better resources [73]. Mobile agents have emerged as a powerful paradigm to deal with intermittent connectivity [59]. They can make decisions at runtime and migrate to locations with better resources to autonomously fulfill their tasks. In our middleware, mobile agents will perform tasks on behalf of their users, e.g. maintaining associations and interactions between mobile terminal and network-based applications. The middleware will be presented further in Section 1.4.1. Having addressed common limitations in mobile users' environments, our focus can now be shifted to our next aim, which is to address problems in supporting collaborative editing between mobile users.

1.3.2 Support for collaborative editing of mobile documents

In terms of addressing the challenges of supporting collaborative editing of mobile documents among mobile users, we first want to be able to provide accessibility to a mobile document to a group of mobile users. One way of doing this is by having a copy of the document hosted on the fixed network. This way, other mobile users can obtain a copy of the document from the network and cache it on their mobile terminals, allowing them to access the document while being disconnected. In terms of the document's consistency, copies of a document, either hosted on the fixed network or on mobile terminals, can be updated with new changes once the original document is changed. Furthermore, access to such copies can be restricted to specified users.

Secondly, in relation to supporting collaboration between mobile users, we are aware that such collaborating mobile users are frequently not online at the same time, which prevents them from having synchronous or real-time collaboration. In this respect, we want to support asynchronous collaboration, and our way of doing this is by having a component to support asynchronous messaging between mobile collaborators, which also in parallel keeps track and coordinates the activities performed by the collaborators. Such an application will be hosted on the fixed network to provide continuous access to all collaborators.

Merging these two approaches, we have designed a collaborative editing application, of which a major part is a collaboration protocol capable of supporting both collaborative editing and the sharing of mobile documents. This application will be further explained in Section 1.4.2.

1.4 Research Contributions

The work described in this thesis makes a number of important contributions to the state of the art in the area of distributed information management in mobile users' environments. The main aim of the thesis is to prove that our proposed mobile-agent based middleware for mobile users is a practical solution to support applications in mobile users' environments by improving their performance in such environments. In achieving this, this thesis first elaborates on the middleware layer, which is supported by a coordination algorithm incorporating approaches and mechanisms for addressing problems in mobile users' environments. Secondly, the thesis introduces a collaboration protocol that supports a collaborative editing application, which is tailored to the characteristics of mobile users and the features of their environments. Thirdly, we present an evaluation that proves that the middleware improves the application's performance in mobile users' environments. Each of these contributions consists of several novelties, which are introduced next.

1.4.1 Mobile-Agent based Middleware for Mobile Users (MAMiMoU)

Our proposed middleware is called the *Mobile-Agent based Middleware for Mobile Users (MAMiMoU)*. MAMiMoU acts as an abstraction layer that conceals the complexity in the interactions between mobile terminal and network-based applications by providing a set of APIs to the applications. The mobile agent, which is MAMiMoU's main component, is hosted on the fixed network, performing various functions on behalf of a mobile user on the infrastructure by taking advantage of the resources in its environment, while in parallel the mobile agent also attempts to migrate closer to the mobile terminal. Due to this behaviour, we call this mobile agent a *shadow*. The shadow incorporates a store-and-forward mechanism, which allows asynchronous messages to be exchanged between mobile terminal and network-based applications. The shadow can exchange messages with the mobile terminal whenever the mobile terminal is online, i.e. attached to the network, whereas, when the mobile terminal is

disconnected, the shadow can receive messages from network-based applications on behalf of the mobile terminal's applications, which will be forwarded to the mobile terminal once the mobile terminal is online. In doing this, the shadow maintains associations between interacting applications. Besides this, during a mobile terminal's disconnection, the shadow can continue with other user tasks, e.g. creating or removing applications on the fixed network on behalf of the user.

On the failure of a shadow to migrate, for example due to the disconnection of the user's current local network from the rest of the Internet, the middleware dynamically creates a new shadow for the user to provide an immediate support at the user's new location. Adopting this approach may result in a situation where a user is associated with multiple shadows on the fixed network. To address this, the middleware is supported by a coordination algorithm, which handles reconciliation between existing shadows and coordinates interactions between the components of the middleware. The reconciliation between existing shadows includes the accumulation of tasks, i.e. in the form of states, between the shadows, and these tasks will be handed over to and later performed by a single main shadow, in which case other shadows are then allowed to terminate.

The novel features of the middleware include:

- The middleware layer architecture consists of one or more mobile agents that carry the states of tasks requested by a user.
- The middleware is driven by a coordination algorithm that coordinates interactions and activities between its components and coordinates the reconciliation of tasks between multiple shadows.
- The middleware provides an abstraction layer allowing users to create and remove applications on the fixed network, and also allowing network-based applications and mobile terminal applications to transparently interact with each other.

The practicality of the middleware is first demonstrated by having it support an application that supports information sharing between mobile users [134], and secondly by supporting a collaborative editing application, which is described in Chapter 6.

In describing the middleware, we prove that the middleware is able to offer advantages listed below:

1. Complex functions can be executed on the fixed network, relieving capability-limited mobile terminals.
2. Less communication is required and thus less bandwidth is consumed.
3. Online operations while mobile terminals are disconnected are accounted for.
4. A mobile terminal's mobility is hidden from the network-based applications by having a mobile agent as its representative on the fixed network.

1.4.2 Collaboration Protocol for Mobile Users

We introduce a collaborative editing application architecture that supports mobile document sharing and collaborative editing between mobile users. A major part of the application is its collaboration protocol that keeps track of the collaborative activities performed by mobile users, and details of the shared mobile documents. The application's architecture includes repositories, which are hosted on the fixed network to store copies of the shared mobile documents, making the copies readily accessible to other mobile users. In supporting such shared mobile documents, the collaboration protocol incorporates a mechanism that maintains consistency between a document and its copies hosted across repositories on the fixed network and other users' mobile terminals.

The design of the collaboration protocol is tailored to the characteristics of an environment for mobile users such that it:

- Supports asynchronous interactions between collaborating mobile users
- Maintains consistency between copies of the shared documents, which are replicated across repositories on the fixed network, and cached on users' mobile terminals
- Restricts access to a document and its copies using the users' roles, in which case one user may have more control over the original mobile document than the others
- Adopts the editing token metaphor to support the delegation of control over a shared document. These combined features are described by the collaboration protocol, which is another novelty presented in this thesis.

1.4.3 Evaluation of the middleware

We perform an evaluation in order to prove that MAMiMoU is a practical solution to address the intermittent connectivity experienced by mobile users, and limitations caused by their mobility. Specifically, this is done by showing that MAMiMoU can improve the performance of the collaborative editing application in mobile users' environments. The evaluation involves testing the real implementation of the system in simulated environments of mobile users, in which the collaborative editing application is used by a group of collaborating mobile users to accomplish a collaborative editing task. In these environments, we compare the effect of MAMiMoU supporting the application with MAMiMoU not supporting it, by recording and analysing the performed collaborative editing tasks.

The evaluation is another novel item presented in this thesis, which has the features listed below:

- Evaluation set-up: the evaluation is set up by simulating mobile users' environments in four dimensions, three of which reflect the potential limitations in mobile users' environments.
- Simulation methodology: A group of mobile users is simulated as performing collaborative editing tasks adopting three different patterns of collaboration.
- Measurements are taken in a time metric, each representing the time needed for a group of mobile users to complete a collaborative editing task. Hypothesis testing is performed on the measurements to prove their statistical significance.
- Analyses are given based on different aspects of limitations in mobile users' environments.

1.5 Thesis Structure

In Chapter 2 and 3, we survey related literature, using this to clarify the problem domain and provide reasons for the work. We first discuss the mobile computing environment in Chapter 2 by presenting some distributed information management applications that have been developed for mobile users. In Chapter 3, we present the background of mobile agent technology and some related mobile-agent based applications. In Chapter 4, we present our proposed Mobile-Agent based Middleware for Mobile Users (MAMiMoU), where we describe its architecture and

coordination model. This is followed by the detailed presentation of the coordination algorithm employed by the middleware in Chapter 5. In Chapter 6, we present in detail our application, which is designed to support mobile document sharing and collaborative editing between mobile users. In Chapter 7, we discuss the evaluation we perform to prove that MAMiMoU is a practical solution to support mobile users' applications by addressing limitations in their environments. Finally in Chapter 8, we present our conclusion and future work.

Chapter 2

Applications for Mobile Users

Mobile computing has become more prevalent in our lives, influencing the design of applications in many areas of computing. In comparison with traditional computing, the design of mobile computing applications requires more effort, since their environments demonstrate a high level of dynamicity with ever-changing populations of accessible resources. Adding to this dynamicity are users, who are frequently mobile and may experience intermittent connectivity and variability in the quality of their connections. All these aspects are experienced as challenges in the design and development of applications for mobile users.

2.1 Overview

The advantages offered by wireless technology motivate its common use among Internet users. Wireless technology allows users to be connected to the Internet while being mobile or "always connected" [66]. Many mobile applications have been developed to support personal needs, e.g. to communicate to other people; business, e.g. m-commerce [109][117][64]; security e.g. battlefield [39]; and community service, e.g. health service [76]. Our interest is to explore the area in which mobile applications are developed to provide adapted information to mobile users [113]. Thus in the rest of this chapter our discussion will be focused on issues and applications in this area.

Providing applications in a wireless or mobile environment is a challenging task, which requires the application designer to consider users' disconnection, application context and failure modes. In considering the variability of connections, application designers may adopt one of three ways

based on the currently available connection: i) the mobile applications assume high bandwidth connections and operate only while plugged in; ii) the mobile applications assume low bandwidth connections and intermittent connectivity and do not take advantage of higher bandwidth when it is available; or iii) the mobile applications adapt to the currently available resources, providing the user with a variable level of detail or quality [27]. Our interest is in looking at applications that are developed according to the third approach. Accordingly, we survey the related issues, which are presented in the next section.

2.2 Mobile Applications Issues

Applications that are developed for mobile users are known as mobile applications [29], or as nomadic computing applications [1]. They are programs that allow the user to leverage network connectivity provided by either a wired or a wireless infrastructure, to productively access and utilise information of interest to the users from any location, on any platform, and at any time [1]. The mobile environment is more dynamic than the traditional wired environment. For this reason, we choose to break our study into three main issues, namely i) limitations in traditional communication models, ii) dynamic reconfiguration, and iii) adaptation.

2.2.1 Limitations in existing communication models

The characteristics of a wireless network, especially cellular wireless, fluctuate variably in comparison to those of a fixed network that change slowly over time. Besides this, communication in a wireless environment is characterised by frequent disconnections. Therefore, communication model such as Remote Procedure Call (RPC) suffers a variety of limitations in a mobile environment [41]. These traditional communication models have the drawback of tight coupling, which means that a sender has to know the exact identity and address of a receiver. The sender also has to wait for the receiver to be ready for exchanging information (synchronization paradigm). In addition, the receiver stores long-term state information related to particular sender-receiver associations [31]. Such requirements fit poorly in a dynamic environment like mobile computing, where a decoupled and opportunistic style of computing is thus required. Application requirements such as dynamic reconfiguration, context-awareness, and adaptation also need to be supported [29].

Another requirement in the mobile environment is disconnected operation, which is defined as the ability of an application to operate as close to normal as possible when disconnected from network resources [3]. For example, in the case of a client-server application, the client should be able to operate for long periods of time after either voluntary or involuntary disconnection from its server. When the client reconnects, it should be able to reintegrate changes onto the server [3].

Some other requirements in developing newer and more active applications that have caught our interest are presented in [41], namely i) support for dynamic routing so that messages that fail to be delivered to the mobile terminal can independently change routing within the network; ii) message cloning, creation and persistence in order to support increased robustness and enable dynamic multicasting; and iii) active (code-carrying) messages capable of autonomous operation when the mobile terminal is disconnected. With all the aforementioned requirements, the question remaining is "With what technology can such an active application be built?"

2.2.2 Dynamic Reconfiguration

Being mobile, users' terminals are exposed to different types of environments, for example, environments with different types of resources provided for the locally connected users. In facing such changing environments, mobile terminals need to support dynamic reconfiguration in order to be able to appropriately adapt to the environment. Besides, considering a different aspect of users' mobility, there is a need to support users' personal mobility, which addresses the issue of personalization [112]. Personalization means the dynamic reconfiguration of a user's operating environment regardless of the terminal or network the user is using. Another area in which dynamic reconfiguration is needed is where context-aware applications are employed. Such applications rely on the current context of a mobile user at any one time, e.g. location-dependent information.

Traditional, stationary terminals are provided with information that is configured statically, such as the local name server and available printers, whereas a challenge for mobile computing is to factor out this information intelligently and provide mechanisms to obtain configuration data appropriate to the present environment of the mobile terminal [27]. The mobile user may not be aware of such changes in configuration, since such configuration can be made explicit only to the

mobile terminal's intermediary or to mobile applications. This information may include details on local resources that are available on the fixed network to support the operation of mobile applications.

One most important aspect in dynamic reconfiguration is the discovery of the available resources in a mobile terminal's current operating environment; this is known as services discovery [29][66]. Since mobile terminals often roam around various areas, services or resources that were present before disconnecting from the network may not exist after reconnecting. To support dynamic service discovery, there are systems like Jini [122], Ninja Service Discovery Service (SDS) [17], Universal Plug and Play (UPnP) from Microsoft, E-speak from Hewlett-Packard, and the Service Location Protocol (SLP) [66].

To illustrate the function of a service discovery system, we now describe Jini. Jini [122][66] enables any terminal that runs a Java Virtual Machine to interoperate with others by offering and using services. A service is defined as an entity that can be used by a person, a program, or another service. Printing a document is an example of a service that can be advertised by Jini. When a service is in use, it is registered as being leased by another service. To join the federation, a device or a service uses a standard mechanism to register. Firstly, a lookup service has to be discovered by polling the local network. Once the lookup service is found, the device or service registers itself. The lookup service stores pointers, code or proxies representing the service, as well as defining service characteristics and attributes.

2.2.3 Adaptation

In the dynamic mobile environment, mobile applications are required to be able to adapt to the prevailing operating conditions. This makes adaptation an important mechanism for supporting mobile applications. Adaptation is used to conceal complexity and variability in the mobile environment from users, mainly by using local resources to reduce communication and to cope with uncertainty. Three different approaches to adaptation are suggested in [100], namely a laissez faire approach, application-transparent adaptation, and application-aware adaptation.

In a laissez-faire approach, adaptation is entirely the responsibility of individual applications. This avoids the need for system support, and there is no need for a central arbitrator to resolve the

incompatible resource demands of different applications. Such an approach makes applications more difficult to write, and fails to amortize the development cost of support for adaptation. By adopting an application-transparent approach, adaptation is placed entirely on the system. Such adaptation is backward-compatible with existing applications, where the system provides the focal point for resource arbitration and control. But in some cases, it may also be inadequate or counterproductive. Finally the application-aware adaptation supports a collaborative partnership between applications and the system, which permits applications to determine how best to adapt [100].

The adaptation support for a mobile terminal's applications can be provided by an abstraction layer that provides an API to the applications. To be able to serve many different types of applications, the abstraction layer can adopt the application-aware support, since this would give more advantages to the applications; e.g. by providing them with the right type of adaptation, and at the same time reducing the adaptation burden on the abstraction layer.

2.2.4 Discussion

The aforementioned requirements may involve a full or partial re-implementation of an existing application in order to fit the mobile operating environment. Another alternative is to have an intermediary that adopts an appropriate communication model, while in parallel hosting dynamic reconfiguration and adaptation mechanisms for mobile applications. The diversity and heterogeneity of mobile terminal systems also raises the need for intermediaries. For example, a low-end mobile terminal may be incapable of hosting a complex application. Therefore, the mobile terminal requires support from an intermediary on a more powerful terminal, such as a stationary terminal connected to the fixed infrastructure. Then the intermediary translates and forwards external communication requests to the low-end mobile. Such intermediaries can also be used to transform ordinary information streams to enhance the information quality [31].

Middleware systems can provide such intermediaries, which deal with the increased complexity that comes with a dynamically changing population of applications, types of connections and hardware capabilities in mobile environments. Such middleware systems will support not just one specific mobile application but also all that are hosted on a mobile terminal. Some work has already been done in this area and some examples will be presented in the next section.

2.3 Middleware for Mobile Users

In the context of mobile computing, a middleware for mobile computing seeks primarily to hide the underlying networked environment's complexity by insulating applications from explicit protocol handling, distributed memories, data replication, network faults, and parallelism [31]. In addition, some middleware systems are designed to mask the underlying heterogeneity of the hardware and software platforms of mobile terminals, while others are built to address the mobility issue of the mobile users. In this section, we present some existing middleware that have been designed and built to support such functionalities.

2.3.1 Issues in Middleware design

Conventional middleware technologies, e.g. CORBA, Web Services, DCOM and Java RMI, have focused on masking out the problems of heterogeneity to facilitate the development of distributed systems. They have proved their suitability for standard client-server applications, but they are not prepared to address the dynamic aspects of mobile systems [29]. This is due to their nature, which allows them to deal best with static execution platforms where the host location is fixed, the network bandwidth does not fluctuate, and services are well defined. Besides this, such middleware platforms are too heavy to run on devices with limited resources.

Moving in a different direction, asynchronous and reactive systems are now targeted in order to address the dynamic aspects in the mobile environment. To be suitable to serve such asynchronous and reactive systems requires a more abstract communication paradigm, i.e. one based on events. Events are in the form of simple asynchronous messages, which have the ability to represent an intuitive way of modelling that something has occurred, in which case such an occurrence or event is potentially of interest to some other objects. Middleware systems that are based on this paradigm are often called "publish and subscribe middleware" [66]. The publish and subscribe model is defined as a mechanism for sharing data between applications [26]. Applications can "publish" their data to a publisher, which will automatically notify all applications that have "subscribed" to that type of data. Anonymous and loosely coupled communication between publisher and subscriber, along with the inherently asynchronous nature

of these systems, help them adapt quickly to changing environments, making them a good choice for mobile cellular networks.

In Web Services technology, a Web Service is allowed to pass a reference to itself or to another Web Service, permitting the latter to call the former. This callback capability lets services asynchronously push events or notifications to each other, and forms the basis of the WS-Eventing and WS-Notification specification [42]. The WS-Eventing specification allows Web Services to provide asynchronous notifications to interested parties, in which it defines the simplest level of Web Services interfaces for notification producers and notification consumers. At the same time, the WS-Notification specification defines a standard Web services approach to notification using a topic-based publish/subscribe pattern.

Unpredictable disconnections are one of the issues that need to be addressed in mobile environment [29]. Data replication and caching is one approach taken to address this issue [49][111]. This approach maximises the availability of data to users, where users are allowed to have access to replicas and to continue their work while being disconnected. Different mechanisms can then be used to maintain consistency among the replicas, e.g. discovery of inconsistent data and data reconciliation. Since conventional pessimistic locking and concurrency control mechanisms restrict mobile systems too tightly, an optimistic approach is more favourable in such settings, where modifications to replicated data are allowed during disconnection, followed by data reconciliation upon reconnection [31].

Another solution proposed to handle the problem of disconnected operations is by having a Tuple Space middleware, especially to support asynchronous communication [29]. Tuple Space is a logically shared memory, which provides the appearance of shared memory but does not require an underlying physical shared memory. Any process can reference any Tuple[1] regardless of where that Tuple is stored. Tuple Space solutions, like LIME [88], TSpaces [130], and JavaSpaces [123] provide tuples, which are the basic elements that can be read or written by many participants. Tuple-space systems are suitable to operate in the dynamic nature of mobile environments since they exploit the decoupled nature of tuple spaces for supporting disconnected operations. The asynchronous interaction paradigm that they offer is an appropriate solution for dealing with the intermittent connection of mobile devices, as is often the case when a server is not in reach or a mobile client requires to voluntarily disconnect to save battery and bandwidth.

Some middleware systems are specifically designed to support context-aware operation. Such systems provide mobile applications with the necessary knowledge about the execution context in order to allow applications to adapt to dynamic changes in mobile host and network conditions [29]. The most commonly used context is location-based information, while other types of context include mobile device characteristics, network condition, and user activity. Such context information is typically presented in a suitable format, which can easily be used by the middleware to apply a certain adaptation policy.

Another issue in middleware design is security. Cryptographic techniques are commonly employed in distributed systems to secure messages from one terminal to another and to safeguard private information stored in networked systems [124]. These techniques can be adapted accordingly, e.g. by ensuring private data does not become public. To protect privacy, schemes based on "digital pseudonyms" could be used to eliminate the need to give out items of personal information that are routinely entrusted to the wires today, such as credit card numbers, social security numbers and addresses.

2.3.2 Middleware Systems

In this section, we review some existing middleware systems that have been designed for mobile users. Their functionalities vary according to different dynamic aspects of the mobile environment.

2.3.2.1 iMASH

The iMASH research project [1] aims to facilitate network interconnectivity between an application server (AS) and users working on heterogeneous mobile and fixed clients. In the iMASH architecture, a Middleware Servers (MWS) layer is interpositioned between the clients and the AS. Several servers are employed to act as an aggregate Middleware Service (MWService) to provide the scalable functionality intended by the iMASH environment. The iMASH middleware is designed primarily to ease the burden on the AS by performing user

[1] A tuple is a fixed fixed-length list containing elements that need not have the same type.

authentication and profiling, presentation transcoding, location tracking and network quality-of-service adaptation, computational off-loading from client to middleware, and application session handoff.

This Application Session Handoff (ASH) facility allows the user to have a seamless, uninterrupted computing experience across client machines, where users are allowed to move the current session state of an application running on one client to another (likely heterogeneous) client. Supported by such mobile session states, the user is presented with a familiar set of data that is adapted, or transcoded, to fit the prevailing operating environment of the device being used. This way the user is allowed to access his data regardless of current platform or physical location. The runtime environment that supports this model transparently tracks the user's location and transfers the application's session state across devices. Hiding such complexity is a tier of MWS that seamlessly provides the service to the clients by performing content transcoding and intelligent data updates so that the user's view of data is consistent across all devices.

In iMASH, the middleware servers (MWS) are acting as intermediaries between mobile terminals and application servers. Multiple servers are employed to address scalability, in which case a local available server will serve a mobile terminal. Even though the system supports the handoff of application sessions between different types of mobile terminal, it does not address the intermittent connectivity and mobility of mobile terminals. No description is provided on what the system will do when a mobile terminal is disconnected or relocates to a new environment. It is not clear whether the server stores the application session while the mobile terminal is disconnected and whether the system allows the mobile terminal to resume this session upon reconnection.

2.3.2.2 Integrated Personal Mobility Architecture (IPMoA)

User mobility has been divided into two areas, namely terminal mobility and personal mobility [112]. Terminal mobility focuses on the movement of the terminal and work in this area includes the standardisation of Mobile IP [86]. Personal mobility, on the other hand, addresses the issues of users' contactability and personalisation. IPMoA (Integrated Personal Mobility Architecture) [112] is a middleware that is specially designed to address personal mobility. IPMoA operates

between the network and the application layer, which is organised similarly to the way home agents and foreign agents are organised in Mobile IP.

A user who wishes to use the IPMoA services subscribes to an IPMoA network, which is known as the user's "home" IPMoA network. On arrival at a new location the user has to log on to an IPMoA network, which is known as the "visiting" IPMoA network. The visiting IPMoA network first authenticates the user with her "home" IPMoA network, after which access to the personal mobility support functions is given. The IPMoA network consists of three primary components (cf. Figure 2.1), namely, i) four manager entities, ii) an adaptation engine, and iii) an execution environment. Each of the four managers has different functionality, e.g. the Application Manager assists a user in executing the user's applications remotely while being away from the home network. The execution environment provides an environment for the user's helpers, namely, agents to execute tasks and assist the user, while the adaptation engine is responsible for performing the adaptation. Three agents are used to provide assistance to the user: a Personal Application Assistant (PAA); a Personal File Assistant (PFA); and a Personal Communication Assistant (PCA). These agents are realized as mobile agents, allowing them to operate in close proximity to the user in order to increase flexibility by facilitating operation over poor communications links.

Once a user reconnects to a visiting network, the PAA, PFA and PCA migrate to the user's current local environment. The PCA allows the user to make a call to another user, the PFA allows the user to get needed files from their home network, while the PAA allows applications to be executed at the user's requests on their home network. In performing their tasks, these agents may migrate and clone themselves. One problem with the approach taken by IPMoA is theat the system depends too much on the home network of the user. All the three agents that serve the user at his / her new location originate from the home network. They will be unable to provide their services if the visiting network is not connected to the rest of the Internet, or if the home network is disconnected from the Internet. A mechanism to handle such failures needs to be considered, e.g. by having the visiting network dynamically create new agents for the visiting user. In the case where the visiting network already has an application that can serve the user's requests, the locally-available application should be manipulated by the PAA instead of having to execute the application remotely on the user's home network.

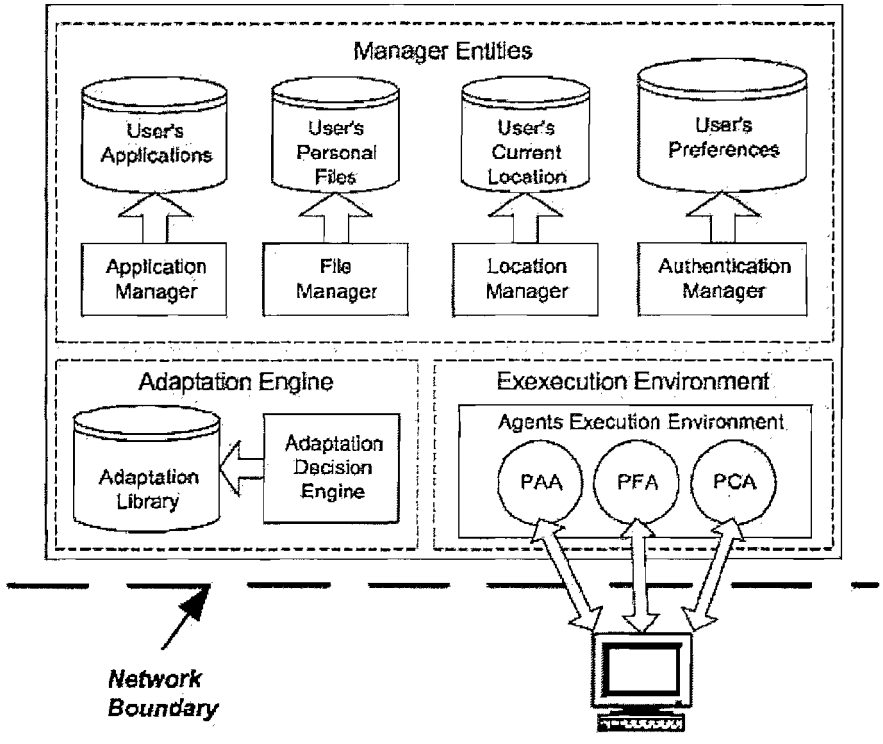


Figure 2.1: The components in an IPMoA network.

2.3.2.3 Mobile Application Support Environment (MASE)

The Mobile Application Support Environment (MASE) [3] is a middleware system that acts as a mediator between applications and their users on the one hand and the mobile terminals and the underlying networks on the other hand. MASE's goal is to hide the complexity of the network from the application, by attempting to make different wireless networks appear to the user and applications as a seamless and homogeneous communication medium. The services of MASE are accessible through a mobile API that includes location, session, and power management, support for disconnected operation, and system adaptivity management to support quality of service in a mobile environment. These services allow the applications to adapt, for example to changes in quality of service, connectivity, and user location.

One important feature of MASE is its adaptation mechanism, for which it employs the UMTS Adaptation Layer (UAL). UAL encapsulates low-level mobile communication functionality, allowing it to establish, maintain, and terminate network communication, alerting its users to

incoming calls, field strength, fading, etc. Quality-of-service requests are made to the UAL as well as requests for charging and location information. Thus, the UAL shields its users from the complexity of mobile networks, offering instead a simpler communication abstraction.

Other components of MASE include the session manager, system adaptivity manager, location manager and accounting manager. The MASE session manager offers services to establish, close and exchange data in a communication session. It selects and requests the appropriate UAL service based on QoS (e.g., quality, cost, security, priority), user/terminal profiles, and available connectivity. The system adaptivity manager (SAM) is responsible for controlling system resources and their usage. SAM tunes the mobile system for best performance, given the current operating environment and the user's preferences. The location manager deals with mobile location information by locating the mobile terminal through the appropriate means available from MASE, e.g. the network, the application and user inputs. Finally, the accounting manager is responsible for charging and billing tasks.

MASE supports disconnected operation, where the mobile terminal is allowed to operate as close to normal as possible when disconnected from network resources. Upon reconnection, the mobile terminal is allowed to re-integrate the modifications performed during the disconnection from the server. While providing such support, MASE does not enable operations to be resumed on the network on behalf of the user while the user is disconnected. This ability is also useful in mobile users' environments, especially when user is involuntarily disconnected from the network. This way, when the user reconnects they may obtain the results of tasks performed on the network while he / she was disconnected.

2.3.2.4 Java messaging middleware (JMS)

Java messaging middleware (JMS) [62] operates using message queues, which are hosted on both mobile devices and back-end servers. Fully bi-directional, one-to-one, or one-to-many communications are supported. Outgoing messages are added to a queue on the mobile device and are forwarded to a server when a network connection can be established. This allows the client and server to communicate despite sporadic network disconnections. JMS was not explicitly designed for mobile devices but it provides an ideal abstraction layer for developing mobile applications. JMS allows many mobile devices to simultaneously send messages to a

server. On arrival at the server, these messages are added to an inbound queue and will be dealt with when resources are available, or can be forwarded to other servers for load sharing.

In comparison with CORBA, JMS is better suited for running applications over wireless networks as it provides an asynchronous, message-based transport [62]. Using message queues hosted on the client and the server side, JMS applications can operate in disconnected mode. A mobile terminal being served by a JMS system may not be able to access the server that queue messages intended for it due to the mobile terminal's new location not being connected to the rest of the Internet. JMS can introduce more flexibility by having messages queued for a mobile terminal be transferred to a server local to the mobile terminal. This way, the mobile terminal is able to use a local service to send and receive the messages.

2.3.2.5 Discussion

By surveying existing middleware systems for mobile users, we can conclude that the following are the main aspects that need to be considered when designing a middleware for mobile users.

Hide the underlying complexity in the mobile environment

The mobile computing environment is more dynamic than the traditional computing environment. Not only has it involved limitations due to wireless communication and low-end devices, but it is also populated with heterogeneous software and hardware platforms and ever-changing resources, services and clients. Looking from a mobile user's perspective, while roaming the user's mobile terminal is exposed to different types of environments. In order to continue their operations, mobile applications that are hosted on the mobile terminal need to be able to adapt to the current environment of the fixed infrastructure. For example, they should be able to maintain interactions with network-based applications that are providing them with services. This is one aspect that needs to be addressed by a middleware system, i.e. an abstraction layer that hides the underlying interaction complexity from mobile applications, allowing transparent interactions between a mobile terminal's applications and network-based applications.

Asynchronous communication support

Mobile terminals have a tendency to be intermittently connected to the fixed infrastructure. Since such behaviour will lead commonly used synchronous communication to fail, the perfect solution

would be to have an asynchronous form of communication that did not require both sender and receiver to be connected at the same time. The simplest approach in supporting such asynchronous communication is by having a store and forward mechanism, which allows messages to be queued and forwarded to the recipients.

Mobile terminal's location updates

In order to serve user's requests, mobile applications may need to interact with network-based applications. Due to the user's mobility or disconnection, such interaction may be interrupted and, upon reconnection, be impossible to be re-established since the network-based application has no way of recognising the mobile application as its location has changed. To address this issue, a mechanism hosted on the fixed network is needed to maintain associations between the interacting mobile terminal's applications and network-based applications. This mechanism must also have the ability to know the current location of the mobile terminal, so that messages from the network-based applications can be delivered to the applications hosted on the mobile terminal.

Scalability

Mobile applications must not solely rely on a single middleware server / intermediary to serve its request. A single and centralised server can be a single point of failure when it is not accessible by the client or for whatever reason fails to respond. Thus to prevent such failure, multiple servers / intermediaries should be employed by the middleware.

Having reviewed the existing middleware systems for mobile users, we have identified some of their limitations, which we describe below:

i) The use of a centralized server for mobile users does not address scalability, in which case a high number of requests received from the mobile terminals may introduce delay and a bottleneck. A centralized system is also a potential single point of failure, in which case the services offered would become unavailable to all of the mobile terminals.

ii) A stationary proxy of a mobile user is lacks flexibility, and may be located far away from the mobile terminal's current point of attachment to the fixed network. A local proxy is always preferred in mobile users' environments, since being local the proxy can facilitate operation over poor communication links. The mobile terminal can communicate with the proxy using a specialized protocol.

iii) Services offered by proxies need to incorporate disconnected operation, not only by allowing mobile terminals to operate normally while they are disconnected, but also by allowing user's session to be updated on the fixed network while their mobile terminal is disconnected, having the results forwarded to the mobile terminal upon reconnection.

In the next section we shift our focus to a study of existing collaborative applications designed for mobile users. A collaboration normally involves shared resources between collaborators, and our interest is to find out how such resources are maintained and how interactions between collaborators are supported.

2.4 Collaboration Support for Mobile Users

Employees who travel or work in remote and changing locations should be able to attend to business tasks regardless of their physical location. Furthermore, while roaming to different network areas, an employee has to be allowed to seamlessly continue a collaborative session. It is widely recognized that supporting nomadic workers in such settings is a key business requirement for large enterprises [101]. Hence, there is a growing need for a comprehensive software infrastructure, which enables employees to work together by locating information and sharing it with their collaborators [13].

In such a collaboration context, the concept of supporting collaboration-aware applications refers to those that are especially designed to support a group, i.e. they contain special code for group functions, while collaboration-transparent applications are original single-user applications, which, with help of a specialised application, e.g. a toolkit, can be used by many users simultaneously [13]. Another form of applications that support collaboration is groupware applications that provide a means for geographically dispersed users to show, manipulate, and emphasize ideas using a software architecture that disseminates multimedia information in real-time to other participating users [1]. We will first survey some research work from the Computer Supported Cooperative Work (CSCW) field, while in the rest of the section we describe some applications that are specially designed to support collaboration between mobile users. This survey is performed as a background study for our work (cf. Chapter 6).

2.4.1 Computer Supported Cooperative Work

Research papers in the field of Computer Supported Cooperative Work (CSCW) study a variety of issues like concurrency control, user interface and the transmission of awareness information [23][51][20][35]. In this section we present some related work from this field.

The Artefact framework [23] is a tool for building collaborative applications on the web. In Artefact, online presence of a user is provided by a software object called the Artefact User Agent (AUA). The AUA can be contacted by the browser through a single URL while AUA interacts with different applications. To build a collaborative application, objects can be declared and added on-the-fly by users. Every object supports access-control lists to allow fine-grained control over security. Artefact also provides a complete object-oriented SDK for developing customized application objects.

Iris [51] is a collaborative system based on a replicated architecture, designed to support geographically-distributed co-authors in the writing of multimedia documents. To address maintaining consistency of replicated documents in this environment, Iris supports optimistic concurrency control mechanisms. Iris allows the documents to be accessed and modified by different user applications, including standard text editors. Replica managers are hosted on every machine participating in the editing process, which store the document objects and provide access and notification broadcasting. Specifically, they handle read and write operations, and notify the users of events happening in the group.

Groove Workspace [35] provides support for online and offline access to integrated messaging and collaboration (e.g. group calendaring/scheduling, project management, file transfer and access, instant messaging, and online meetings). The Groove platform uses binary differentials to optimize network traffic as content flows throughout the lifecycle of a project. Binary differentials technique of comparing the original document and the revised document at the binary level, determining which bits have changed, and sending only the changes across the network to the other members of the shared space.

Artefact, Iris and Groove are examples of collaborative applications built for stationary users. Even though they may have many interesting features, they are not specifically tailored to the characteristics of mobile users. Below we present an example of a collaborative editing application specifically built for mobile users.

CoFi [20] is an architecture that supports document editing and collaborative work over bandwidth-limited clients. It enables bandwidth-limited clients to edit documents that are only partially present at client. Such conditions are subject to adaptation systems that adopt techniques to lower download latencies. These systems reduce network traffic by the use of subsetting and versioning. In subsetting adaptations, only a subset of the components of the original document, for example the first page is transferred. In versioning adaptations, some of the components are transcoded into lower-fidelity representations, for example low-resolution images. CoFi is a collaboration system that is more specific to documents being manipulated by the adaptations systems. This makes CoFi suitably attached to adaptations systems. Our focus in designing the collaboration application does not include such types of documents. In principle, CoFi does not assume a predetermined relationship between system nodes, while our work includes the concern of intermittent connectivity with mobile users.

2.4.2 QuickStep

QuickStep [94] is a platform that supports the development of mobility and collaboration aware applications. It provides the developers with communication and collaboration primitives, through which they can concentrate on application-specific details. This platform has been especially designed for handheld devices, and employs a stationary server to offer information that can be used by handheld applications. The server also acts as a communication relay between handhelds, relieving them from heavy tasks and storing data during disconnection. In terms of collaboration support, QuickStep introduces the idea of *relaxed synchronous* collaboration, which is defined as collaboration for users who collaborate synchronously, but may infrequently be disconnected from the network for short periods of time. Applications built on this platform allow information sharing between mobile users, e.g. a collaborative calendar tool.

In QuickStep, when a mobile terminal is disconnected, the mobile terminal's session is stopped and the corresponding session's records are immediately removed from both the server and the

mobile terminal's caches. For an involuntary disconnection, such an action may not be suitable since the stopped session may include an incomplete task requested by user. A better approach is to allow such sessions to proceed until the requested tasks are completed, after which the results can be forwarded to the mobile terminal upon reconnection.

2.4.3 Pocket DreamTeam

Pocket DreamTeam [95] is another platform that supports collaborative applications on mobile devices. It is an application framework that was especially designed for mobile users in synchronous sessions. Using Pocket DreamTeam, a developer can create prototypes for mobile collaborative applications by using a high amount of reusable source code and a runtime system, which executes demanding services in the background. Pocket DreamTeam is the extension of a decentralised groupware platform called DreamTeam, with the addition of a stationary proxy computer, which executes the computationally expensive operations of the applications.

DreamTeam is based on a completely decentralised architecture. The DreamTeam environment allows the developer to develop co-operative applications like single user applications. It comprises a development environment, a runtime environment and a simulation environment. Applications under DreamTeam are developed according to the DreamTeam Resource Model (DRM). Applications consist of an application frame, a set of resources and a user interface. With this interface, the runtime system can initialise, start and stop applications. All data is available locally, thus an application is still runnable in case of network problems. The original DreamTeam uses pessimistic concurrency control procedures, which involve pessimistic locking protocols enforcing that data is locked before it is written. This is one of the mechanisms that promote a strict consistency model, in which participants have exactly the same views and data all the times. Such mechanisms are suitable for optimally-connected users in stationary networks, but among mobile and weakly-connected users, such mechanisms are unsuitable since mobile users are normally intermittently connected to the network. As a solution, Pocket DreamTeam offers a combination of pessimistic concurrency control for the mobile segment.

In providing its services to mobile users, Pocket DreamTeam uses additional computers called proxies to execute computationally-expensive operations on behalf of the mobile terminals. Once a mobile terminal is disconnected, the state of the interrupted session is updated by the proxy.

Since Pocket DreamTeam uses multiple proxies to serve mobile terminals, the failure of one of the proxies does not mean that the mobile terminals' sessions handled by the proxy are terminated. The system provides automatic recovery mechanisms that switch a mobile terminal to another proxy without interruption.

While addressing mobile terminal's disconnections, Pocket DreamTeam does not specifically address a mobile terminal's mobility. During a collaboration session, a mobile terminal may roam or relocate to a new location. Throughout this time, the collaboration session state continues to be updated by the mobile terminal's proxy. However upon reconnection, the mobile terminal still needs reconnect to the same proxy, regardless of its new location. More flexibility has to be introduced in such a situation. Since Pocket DreamTeam employs multiple proxies located across the network to serve mobile users, it can allow the mobile terminal to always be served by a local proxy, allowing a local collaboration. At the same time, the mobile terminal can still continue its collaboration session by communicating with a remote proxy. This way, the mobile terminal can maintain both collaboration sessions. One advantage of having a local collaboration is that the mobile terminal is able to manipulate local resources.

2.4.4 YACO

Some efforts in this area focus more specifically on collaboration in the corporate domain. Such an effort is realised as a framework called YACO [8]. YACO offers services to collaborating users in a corporate domain and exploits the capabilities of SIENA [11], a publish/subscribe system, to support collaborative work. In this system, a sender publishes messages and receivers subscribe for messages that are of interest to them, while the system is responsible for delivering published messages to matching subscribers.

Additionally MOBIKIT [9], a support service for mobile publish/subscribe applications, is used to manage the mobility of its client. Using the SIENA architecture, servers provide clients with access points that offer the publish/subscribe interface. These servers are interconnected as a distributed network. The YACO architecture is designed in such a way that MOBIKIT can be used to support the movement of clients between the access points of the SIENA system. YACO uses MOBIKIT to handle the mobility issue of users, whereby each user has a stationary proxy handling their subscriptions and storing incoming notifications. Every time the user changes

location, another locally available proxy becomes the user's new proxy on the user's arrival at the new location. This new proxy downloads all subscriptions and messages for users from the user's previous proxy. This approach requires such a proxy to exist on all the sites ever visited by the user. We believe a more flexible approach is to have a mobile proxy for each user, so that the proxy is able to follow the user by migrating to their current vicinity. This way there is no strict requirement for such a proxy to exist at every site.

YACO allows users to share artefacts, and every user in YACO uses a local repository for storing artefacts. YACO uses publish/subscribe application to support the sharing of those artefacts, where the artefact's owner publishes the events related to the artefact, such as when the artefact becomes available or has just been modified. This way, interested users are notified when these events have occurred. By using publish/subscribe method of communication, only the publisher has the privilege to modify the artefact, while subscribers may only download and view the different versions of the artefact. This may be a good solution for publishing documents, but it may not be suitable in collaborative editing work, where collaborators are allowed to edit a shared document in parallel.

2.4.5 PCCE

PCCE [87] is specific to the facilitation of scientific collaboration within widely distributed environments. PCCE provides a persistent space in which collaborators can locate each other, exchange messages and archive conversations. The architecture includes a centralized server to provide persistence and a modified public domain IRC server to enable text-based synchronous chat. The system is referred to as LBNLSecureMessaging and deployed as the presence and messaging component of the PCCE. Their vision is to extend this system to support file-sharing and collaborative editing. PCCE designers consider integrating both asynchronous and synchronous collaboration into the system. The proposed sharing and collaborative editing of documents will be an extension of the current PCCE system, which consists of a secured messaging system. This requires the collaboration to be tightly coupled to the messaging tool.

Like some other described systems, PCCE employs a centralized server to provide services to mobile users. In this case, the PCCE system is not concerned with the location of the mobile terminals. In supporting collaboration between mobile users, the location of the server providing

the service is an important consideration when the collaborators are stationary users. Centralized systems introduce problems such as latency and bottlenecks, since the processing of requests from the clients are performed sequentially and these problems occur when the server receives too many requests from clients. By having only one server, the probability of a mobile terminal not being able to access the server at a particular time is high, since the mobile terminal may connect to a local network that is disconnected from the rest of the Internet.

2.4.6 Bayou

Emphasizing the support of application-specific conflict detection and resolution and the provision of application-controlled inconsistency, the Bayou system is presented in [111]. Bayou is a platform of replicated, highly available, variable consistency mobile databases on which to build collaborative applications. The focus of the Bayou project is on exploring mechanisms that let mobile clients actively read and write shared data. Bayou adopts a flexible client-server architecture, where the server stores one or more databases and provides the service for read and write operations to the clients. Clients are terminals accessing data from the server. A terminal can be a server for databases and a client for others. Bayou permits "lightweight" servers to reside on portable machines. Bayou chooses a read-any / write-any replication scheme to maximize a user's ability to read and write data. A user is able to read from and write to any copy of the database. Servers propagate writes among copies of the database using an "anti-entropy" protocol, which is commonly known as a "reconciliation" process when used to synchronise file systems. Anti-entropy ensures all copies of a database are converging towards the same state and will eventually converge to identical states if there are no new updates.

The Bayou system detects update conflicts in an application-specific manner. A Bayou write operation includes an application-specific procedure called a *mergeproc* that is invoked when a write conflict is detected. This program reads the database copy residing at the executing server and resolves the conflict by producing an alternate set of updates that are appropriate for the current database contents. The Bayou design includes the notion of explicitly "committing" a write. Once a write is committed, no other non-committed writes will be ordered before it, and thus the outcome will be stable. A write that has not yet been committed is called "tentative". Each Bayou database has one distinguished server, the "primary", which is responsible for committing writes. The other "secondary" servers tentatively accept writes and propagate them

toward the primary using anti-entropy. As secondary servers contact the primary, their tentative writes are converted to committed writes, and a stable commit order is chosen for those writes by the primary server.

Bayou does not address the possibility of having different types of writes. It treats writes from all users the same by having them as "tentative" on secondary servers first before they are committed by the primary servers. In a collaboration between a group of mobile users, the writes made by a particular user, e.g. the document owner, may be valued as more significant than the writes made by the others. Thus writes made by such users take precedence over the other writes, but this does not mean that writes made by other users will be rejected. Parallel editing of the document between users is still enabled but the modification made by different users may have different effects on the document.

2.4.7 Coda File System

Coda [49] supports the mobile environment by providing infrastructures such as disconnected operation, data hoarding[2] and weak connection adaptation. To deal with disconnection, Coda supports data caching on clients to improve performance and to enhance availability. In the mobile environment, data caching allows critical work to be continued on the clients when the server is inaccessible. Another approach adopted by Coda to achieve high availability is server replication. This allows files to be replicated on more than one server. The set of replication sites for a file volume is known as a volume storage group (VSG), while the subset of a VSG that is currently accessible is a client's accessible VSG (AVSG).

Coda distinguishes between replicas on servers, and cache copies on clients. The first are known as first-class replicas, while the latter are known as second-class replicas. First-class replicas are more persistent, widely known, secure, available, complete and accurate. Second-class replicas, in contrast, can only be useful by periodic revalidation with respect to a first-class replica. Coda adopts optimistic replica control, which provides much higher availability by permitting reads and writes on any replica. Coda deals with conflicts by detecting and resolving them after their occurrence. Disconnected operation starts when the AVSG becomes empty, i.e. no server is

[2] This is the process of hoarding or storing useful data on the mobile terminal in anticipation of disconnection.

accessible to a client. In such a case, while disconnected, the Venus services file system, which is the file system hosted on a mobile terminal, relies solely on the contents of its cache. During this time, cache misses appear as failures to application programs and users. Once the mobile terminal re-connects, Venus propagates modifications and reverts to the server replication.

Coda copes with all disconnections either involuntary or voluntary in the same manner. The only differences are in terms of user expectations and the extent of user cooperation. The disconnected operation supported by Coda is similar to a manual caching system with write-back upon reconnection performed by a user, where the user identifies files of interest and downloads them from the shared file system for use while disconnected. When the user reconnects, modified files are copied back into the shared file system. In its operation, Coda introduces three states, namely hoarding, emulation and re-integration states. In the hoarding state, Venus hoards useful data in anticipation of disconnection. In the emulation state, Venus starts to function as a pseudo-server by performing many actions normally handled by servers. Finally in the reintegration state, Venus propagates changes made during emulation, and updates its cache to reflect current server state.

In supporting file sharing in a mobile environment, Coda highlights the following requirements:

i. File replications and caching are needed to enhance data availability;
ii. Optimistic replica control is suitable in mobile environment since clients are expected to be frequently disconnected from the servers;
iii. Reintegration is needed upon a reconnection to revalidate the modified files.

Coda distinguishes second-class replicas hosted on the clients, e.g. mobile terminals, from the first-class replicas stored on the server. The first-class replicas are considered more accurate and complete, while the second-class replicas are considered useful only with periodic revalidation with the servers. In a real collaboration, such categorisation can be the other way round. For example, a copy of a document hosted on the owner's mobile terminal is the latest, most accurate and complete version of the document, while its other replicas hosted on the fixed network may be outdated.

2.4.8 Discussion

From our survey of existing collaborative applications, we have come to the conclusion that in order to support collaboration between mobile users, the aspects described below need to exist in a collaborative editing system for mobile users. The described systems may include some of these features, but lack the others.

Asynchronous interactions

Since mobile collaborators may not be online at the same time, and due to the frequent disconnections experienced by mobile collaborators, asynchronous interactions between them must be supported. Such interactions can be supported by having a store and forward mechanism hosted on the fixed network, for example queued messaging, shared memory or a publish / subscribe system.

Disconnected operation

During disconnection, to some extent users need to be allowed to proceed with the collaborative activity, e.g. working on a copy of the shared document in isolation. One way of allowing this is by supporting document caching on mobile terminals. Upon reconnection, the work performed by the user on the document has to be re-integrated with the original copy of the shared document hosted on the fixed infrastructure. Other approaches can also be adopted to improve disconnected operation, e.g. caching.

Data replication

Replication of shared data is one way to maximise the availability of data to users. The shared data is replicated across multiple servers and normally a mechanism is needed to maintain the consistency between replicas. One way of doing this is by propagating new updates between servers.

Consistency maintenance

As we choose optimistic over the pessimistic concurrency control, we need to think about the inconsistency that is likely to happen to replicas of a shared document. Optimistic concurrency control is suitable for a mobile collaboration since collaborators are allowed to modify the shared document in parallel. As opposed to pessimistic concurrency control, this approach does not impose a locking mechanism on a shared document. Thus, no problem will arise when a

collaborator is disconnected for a long time. First of all, we need to think of a conflict detection mechanism and secondly we have to decide on how the conflicts that occur will be resolved.

Access control

Another aspect of collaboration that we may need to think of is regarding the collaborators' roles. Some of these roles may have higher permission and more control over resources shared between the collaborators. This includes the type of permission allowed for a collaborator to access and manipulate a shared document.

2.5 Conclusion

We have described the limitations of the traditional communication models in supporting mobile computing applications, and we have presented some newer and more suitable approaches. Also in this chapter we have presented our survey of mobile applications of interest. Two classes of applications, namely middleware systems and collaborative applications for mobile users are surveyed. From this study we have extracted their common features and requirements, which are used as the foundation of our work. While the systems discussed may have some of the features discussed in Section 2.4.8, they lack others (cf. Figure 2.2).

One feature of proxies serving mobile users is their location. We think that being locally available to mobile users offers more advantages, e.g. communication between the proxy and the mobile terminal can use a specialized protocol. One approach to realizing this is to have mobile proxies or intermediaries for mobile users, implemented using mobile agent technology. We describe mobile agents in more detail in the next chapter and survey some existing mobile applications based on mobile agent technology.

Features	**Asynchronous interactions**	**Disconnected operation**	**Data replication**	**Consistency maintenance**	**Access control**
Artefact	No	No	No	No	Yes
Iris	Yes	No	Yes	Yes	Yes
Groove Workspace	No	No	Yes	Yes	Yes
CoFi	No	Yes	Yes	Yes	Yes
QuickStep	Yes	No	No	No	Yes
Pocket DreamTeam	No	Yes	Yes	Yes	Yes
YACO	Yes	Yes	Yes	No	Yes
PCCE	Yes	No	No	No	Yes
Bayou	No	Yes	Yes	Yes	Yes
Coda File System	No	Yes	Yes	Yes	Yes

Yes: supported feature
No: not supported feature

Figure 2.2: Comparison between systems

Chapter 3

Mobile Agents

In the previous chapter, we described the current trend in designing applications and middleware for mobile users. We now shift our study to mobile agent technology, which is a potential solution for developing suitable middleware for mobile users. We will start by introducing the concept of a mobile agent, followed by a review of existing mobile-agent- based systems, and finally a discussion.

3.1 Introduction to Mobile Agents

Before we describe what a mobile agent is, we present some previous definitions of software agents. According to [58][16], an agent is defined as a program or software that acts on people's or organizations' behalf, which allows these people or members of the organization to delegate tasks to their agents. Being more specific, [47][129] define an agent as a computer system, situated in some environment that is capable of flexible autonomous action in order to meet its design objectives. Being autonomous, the system is able to act independently without the direct intervention of users or other agents, and it has control over its own actions and internal state. Being flexible, the system is expected to be responsive, pro-active and social. Responsive means that agents are able to perceive and respond to their local environment. By being pro-active, agents are able to exhibit opportunistic, goal-oriented behaviour and take the initiative where appropriate. Social means that agents are able to interact with other agents in order to solve their problems or to help other agents. These four attributes are believed to be the basics of agenthood and to be necessary to provide the power of the agent paradigm that distinguishes agent systems from other software paradigms [47][129]. The FIPA standard fully embraces the agent paradigm, specifically providing the definition of a reference model of an agent platform and a set of services [46].

There is actually a distinction in concern between multi-agent systems / intelligent agents community and mobile agents community [21]. The definitions presented above are from the intelligent agent community, while for the mobile agent community, the concept of the mobile agent is believed to have grown out of three earlier technologies [128], namely process migration [91], remote evaluation [102], and mobile objects [48]. According to [28], the fundamental idea in implementing mobile agents comes from the basic concept of code mobility, which is defined as the capability to dynamically change the bindings between code fragments and the locations where they are executed [10]. The mobile agent community concentrates more on the subsystems for code shipping, while the intelligent agent and multi-agent community tends to focus on application-specific problems. From the mobile agent community point of view, the term "mobile agent" was first introduced by Telescript [21].

Based on the definitions given by both communities, mobile agents can be defined simply as running programs that autonomously decide to change location in order to continue their execution in an environment with better resources [73]. Having migration autonomy allows a mobile agent to travel among the hosts on the network, which is an additional advantage in comparison with more passive forms of mobile code, i.e. code on demand and remote evaluation [28]. As a result, mobile agents are considered as the most powerful form of code mobility [119]. Besides flexibility, mobile agents offer other advantages, which will be presented in the next section.

3.1.1 Advantages of Mobile Agent Applications

Before we describe the advantages offered by mobile agents per se, we take a step backward by identifying the advantages offered by code mobility.

3.1.1.1 Advantages of code mobility

We refer to [28] for a list of advantages offered by code mobility, which are summarized next. First of all, code mobility enables service customization, in which case the server is allowed to request the remote execution of code. This helps to increase server flexibility without

permanently affecting the size and complexity of the server. Secondly, mobile code is used in the deployment and maintenance phases of the software development process by providing an automated installation process. To some extent a mobile code can be autonomous, which allows it to independently execute on a server without involving communication over a physical link. This ability is especially useful in a disconnected environment such as wireless / mobile users' environments. Such autonomy can also improve fault tolerance, e.g. a mobile code migrates to a server to accomplish its task, allowing occurring faults to be handled locally by the server.

Finally, code mobility offers flexibility in data management and protocol encapsulation. The components of a distributed application may exchange data between themselves, where each component owns the protocol to interpret data. It is impractical to hard-wire such code into application components as it may frequently change. In this case, code mobility enables more efficient and flexible solutions. For example, only the specific code that implements a particular protocol is requested to migrate to the local host by an application. All of these advantages also apply to mobile agents in their lower level form, being mobile codes. Next we discuss specific advantages offered by mobile agents.

3.1.1.2 Advantages of Mobile Agents

First of all, the application of a mobile agent can reduce network load in distributed environments [58]. In completing a particular task, applications in a distributed environment normally rely on some communication protocols that require multiple interactions to be carried out between them. One way to minimize the network load in such a situation is by packaging the conversation and moving it closer to the remote application. This allows the conversation to be performed locally and thus removes the need for distanced interactions. Besides this, such an approach is also applicable when an application requires the processing of large volumes of data that are stored remotely. By migrating a mobile agent close to the data, the entire processing is able to be performed remotely and close to the data, preventing large amounts of data being exchanged across the network. In both cases, the use of a mobile agent allows the computation to be moved closer to the needed resources, and thus reduces the network load involved.

Secondly, mobile agents can also overcome network latency [58]. Network latency is a major limitation in critical real-time systems such as robots in manufacturing processes that need to

respond to changes in their environments in real time. Thirdly, in terms of the user's privacy, the capability of mobile agents to keep user-profile data within themselves and prevent other agents to directly access the data [97]. Fourthly, from the security point of view, mobile agents can operate in secured environment where they can provide high levels of security and reliability without requiring additional code to be written [68]. And finally, mobile agents can also greatly help to structure distributed applications [53], since a software architecture that includes mobile agents as one of its components can be represented by many different architectural styles or design paradigms. Such flexibility allows the designer to identify the most reasonable way to use the mobile agent to best represent the architecture.

3.1.1.3 Advantages of Mobile Agents in the Mobile Computing Environment

Mobile agents can run autonomously and asynchronously and this ability is most useful in the wireless environment, where users rely on expensive connections to the network [58][60][106][68][89][37][98]. Having an autonomous mobile agent running on the network on behalf of a mobile user removes the necessity for the user to have continuous connection with the network. The user can always reconnect later to get the results for accomplished tasks. In comparison with other types of software agents, the agents' ability to adapt to their environments is more useful to mobile agents. This is because mobile agents are more exposed to different environments in comparison with stationary agents, as they may visit different hosts throughout their lifetime [58]. The combination of mobility and adaptability possessed by mobile agents offers another advantage in that they can be used to continuously maintain the optimal configuration for solving a particular problem in different types of environments. Based on this behaviour, a mobile agent can be a representative of a mobile terminal that is able to adapt and reconfigure itself to each network being visited by the mobile terminal. This way, by communicating through the mobile agent, the mobile terminal does not have to reconfigure its setting every time it is attached to a new location. From a different point of view, mobile agents can also benefit small terminals such as PDAs and cellular phones [106] by adapting themselves to different types of platform capability.

3.1.2 Mobile Agent Systems

A mobile agent system offers agent life-cycle management, security enforcement, resource access control, agent transport, communication and persistence [6][115][106][105]. Such a system normally consists of an agent platform, which is a runtime environment that is able to perform the tasks of supporting the agents' creation, execution, localisation, migration, communication and security control [33]. Creation and transfer of mobile agents are normally defined by the communication API in the communication layer of the system. A mobile agent platform accepts incoming agents and performs authentication and authorization on them, and also launches new agents and provides mobility services for them [77]. Each agent is identified by a unique name and provides a set of services, allowing agents to interact with each other. In addition, some mobile agent systems support other services like the remote cloning of agents [125], remote creation of agents [58][6] or extensions of services to support operations on small devices [106][46][4].

SoFAR, the Southampton Framework for Agent Research [74][72][73], is a system that supports mobile agents. Since our implementation is mostly based on SoFAR, in the next sub-section we present its overview and its support for agent mobility. This is followed by a sub-section on mobile agent security issues.

3.1.2.1 SoFAR

SoFAR agents communicate based on the speech-act based communications, i.e. the communication mechanism abstracts away from the communication details. A communication between two agents is based on a "virtual link" defined by a startpoint and an endpoint. An endpoint identifies an agent's ability to receive messages using a specific communication protocol; it extracts messages from the communication link and passes them on to the agent. A startpoint is the other end of the communication link, from which messages get sent to an endpoint. Agents' communications are also declarative, where a minimal set of performatives has been defined, representing the most common communication patterns, such as querying, informing, registering, subscribing or requesting. Currently, SoFAR implementation relies on shared memory communications, Java RMI [104], and SOAP [36].

Page 45 missing

executing: i.e., the one executing the method migrate. In a case where an agent is currently processing other performatives, the migrate method will block until all the other threads terminate their execution.

3.1.2.2 Mobile Agent Security Issues

One issue that is normally dealt with by mobile agent systems is security. In mobile-agent-based applications, two major security issues are identified, namely i) protecting host systems and networks from malicious agents, and ii) protecting agents from malicious hosts [53][106][54][121]. With regard to the first issue, malicious mobile agents may access and modify data to which they should not have access and interfere with the execution of other agents [54]. In addressing this problem, host security is a reasonably well-researched issue, and many viable mechanisms have been developed to address it. Examples of such mechanisms include sandbox security (used in Java to provide access control) [32], safe-typed languages [120] and proof-carrying code [78]. Another approach to protecting a host is presented in [54], where unauthenticated agents are rejected from the host or can be executed as an anonymous agent within a very restricted environment. Furthermore, authorization is adopted to determine the mobile agent's access permissions for the host's resources.

Referring to the second security issue, malicious hosts may cheat mobile agents migrating to them and therefore interfere with the successful execution of the mobile agents. In addressing this problem, some of the more well-known mechanisms used to overcome this problem include code obfuscation [40], encrypted functions [99], execution tracing [118] and tamper-resistant hardware [126][131]. These solutions must at least include techniques to protect the code and data integrity of an agent from a malicious host [121]. In [107], mobile agent code and state information is partitioned into self-contained components, which will then be encrypted using symmetric keys that will be made available to platforms hosting the mobile agent. In supporting the distribution of keys to these platforms, a specific type of mobile code called a keylet is employed. Specifically, the keylet is used to control the propagation of keys in a system by directing the distribution of keys that encrypt and decrypt components. Another approach taken to protect mobile code is presented in [108], which introduces the use of a verification server, i.e. a trusted third-party, to verify the execution traces on behalf of the platform launching the mobile agent. This approach ensures the capability of a particular host platform to undertake the correct execution of a mobile

agent by having the server, which assumes the role of a Certificate Authority (CA) in a PKI, testify to this by means of constructing a certificate.

3.2 Mobile-Agent based Systems for Mobile Users

Mobile agent technology has been used in many areas of application due to the advantages it offers. One common advantage is the degree of scalability promoted by mobile agents, which allows the distributed applications to be widely deployed across the network. Mobile-agent-based applications range from many areas such as the advanced telecommunication field [28], electronic commerce [58][80], information retrieval [28], personalization [60][97], adaptations [79] and active networks [132][5]. Since our interest is in studying the application of mobile agents in mobile computing environments, in the rest of this section we will present our study on the existing mobile-agent-based systems for mobile users. Our main focus is on understanding the interactions between mobile-terminal-based applications and services hosted on the fixed network and how exactly mobile agents should fit into such architecture.

3.2.1 A Mobile Agent Framework for M-Commerce

An m-commerce framework based on mobile agents, presented in [67], aims to overcome the limitations of mobile devices, e.g. bandwidth limitations, low resources, etc. The framework provides three types of agents, specifically device agents, service agents and courier agents. A device agent is a stationary agent that resides on a user's device, which is able to locate and access various wireless services on behalf of the user. The device agent is also responsible for handling specific tasks, such as negotiating or paying for a service. A service agent is a heavyweight agent hosted on the fixed network, which is used by a service provider to handle service requests generated by users. Finally, courier agents are lightweight agents, which are used by service agents to communicate with users. These agents contain only limited functionality and the data applicable to the current interaction between a service agent and a user. Once a courier agent has been transferred from a service agent to a device agent, it communicates back with the service agent via an XML-based communication protocol. Once migrated to a mobile terminal, a courier agent cannot travel back to the service agent, to another device, or to another network location.

A courier agent is comprised of two parts, which are the information to be displayed to a user and the business logic that determines how that information is manipulated. Courier agents typically have a short life span; once they have displayed their information, they may be destroyed or cached depending upon the application. A user who disconnects from the network will still be able to interact with the cached courier agent. Once reconnected, the courier agent may resume its session with the service agent. But the acceptable length of inactivity, i.e. the disconnection of the user's device from the wired network, will differ from application to application.

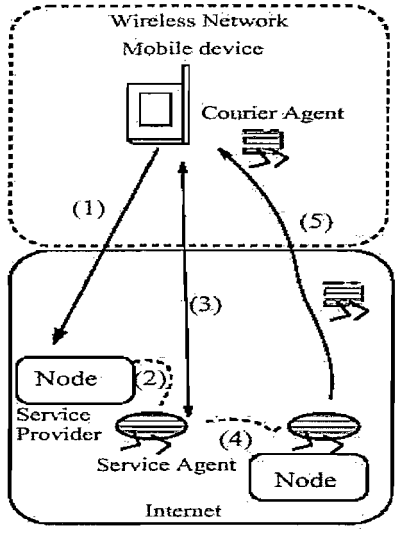


Figure 3.1: Overview of M-Commerce Framework

Figure 3.1 presents a sample scenario of using the framework; the steps include:

1. A user locates and issues a request to a service provider.
2. A service provider creates a service agent to handle the user's service request.
3. The service agent and the device agent begin negotiations for the service.
4. The service agent moves to another network location to fulfil the user's request.
5. The service agent communicates with the user by sending courier agents.

A service agent is able to communicate with the device agent that has initiated the service and with the courier agents it has generated. The same principle applies for courier agents, in which

case they are allowed to communicate with the service agent they originated from and with other courier agents from the same service agent.

In this framework, the service agent is a mobile agent, which is able to move around the wired network in order to gather information and exploit the available resources. Having local access to the needed resources has the advantage of allowing fast and reliable communication between the service agent and the resource. As mobile devices tend to lose their connection frequently, having application logic cached on the mobile device is beneficial, as it allows the mobile user to manipulate the cached information while being disconnected and waiting for the connection to be re-established.

The framework requires an agent framework to be installed on the mobile device, allowing it to run agents. For low-end handheld devices this seems to be too much load since they have limited resource capability, such as limited memory and processing power to run heavyweight applications. Thus, having an agent system running on a mobile device requires either a very lightweight agent system, or that the mobile device is powerful enough to run an agent system. Some operating systems on mobile devices such as PalmOS on PDAs do not support multiple application threads [45]. Although it is possible to install a lightweight agent system, it would allow only one application or agent to be active at a time. The requirement of the framework of having two agents, a device agent and a courier agent running at once prevents such operating systems using the service.

Having courier agents migrate to mobile devices in order to maintain interactions with service agents raises several issues. The process of an agent migrating usually involves the ability to move the agent's state and code, which includes the serialisation and de-serialisation of the transferred data. Besides, security and access control becomes an important aspect when dealing with migrating agents. Again, the fact that some mobile devices do not have a platform capable of running these tasks and handling security issues prevents them from using the service. Having an agent migrating to a mobile device does give a great advantage if the task can be done by an agent residing on the mobile device. For instance, the device agent can handle additional tasks, i.e. the courier agent's tasks, including establishing and maintaining interactions with service agents and storing and manipulating application logic while being disconnected.

Another question raised is how a device agent hosted on a mobile device can discover a service provider on the fixed network. The service provider can be on a remote platform located at a distance away from the network to which the mobile device is connected, in which case the service agent may wait for the mobile device to reconnect since the previous connection was lost. Upon a request from the mobile device, the service agent needs to communicate remotely with the mobile device to serve the device's requests. A better alternative is to have a local service agent serve the user's requests, to allow easier and faster access and more reliable communication [15][112]. This will be an approach we adopt in our middleware (cf. Chapter 4).

3.2.2 Personal Agent System

The Personal Agent System [15] is a mobile agent system that provides a mobile user with a personalized information retrieval service. Using mobile agent technology developed at Dartmouth College, the system gathers information from the Internet and uses context-aware mechanisms to manage the information according to a mobile user's needs and preferences. The user's schedule and location are the context indicators in this system. The Personal Agent System consists of agents, information servers, information-viewing applications and context-related sensors (cf. Figure 3.2). The agents form the central part of the Personal Agent System. The actual personal agent resides on a stationary server and communicates with agents residing on the mobile device. The agents in the system handle the gathering and personalization of information and communicate with one another extensively to provide an integrated service to the user. The information servers supply the personal agent with the information that the agent requests. These information servers can reside on any machine and typically would be owned and maintained by third parties that wish to offer information services to users. For each form of information that the personal agent gathers, there is a corresponding user application that provides an interface for displaying and managing the information. Finally, the context-related sensors monitor the user's location and schedule and notify the appropriate agent when the user's context changes.

The system involves migration of the personal agent to other stationary servers so that it follows the user around in the wired network while the user and her mobile device move around in the wireless network. In other words, the personal agent always moves to a location that is close to the user. Having the personal agent and the agents on the mobile device in close proximity to each other, i.e. ideally in the same local network, allows communication between them to be fast

and reliable. The system is aware of both the user's location and schedule. The general location of a user can be determined from the subnet in which he is located. The user's schedule can be determined from a calendar application that is on the mobile device and is updated by the user. When combined with user preferences, these two forms of information can be actively used to filter and convert incoming information as well as to determine the timing of notifications so that the user is presented with a minimally intrusive user interface.

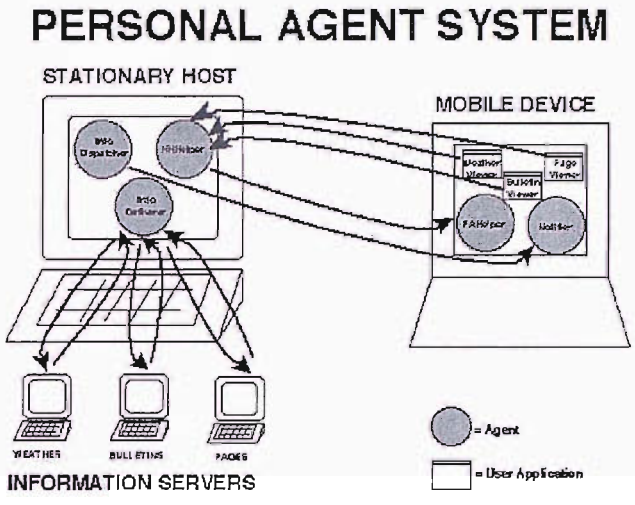


Figure 3.2: Personal Agent System Overview

In this architecture, the personal agent moves around the network trying to stay as close as possible to the mobile user (cf. Figure 3.3). This pattern allows local communication between the mobile device and the personal agent, and ensures fast and reliable communication between them. This is actually a good solution to a situation where the mobile user moves to a new location which is a long distance from its last point of connection to the wired network, where the personal agent is residing. Instead of having to communicate through a very long communication medium if the personal agent were to stay on the same platform, it would make more sense for the personal agent to move closer to the mobile device and communicate locally.

The service of delivering the right information to the mobile user based on the user's profile and context is a suitable approach, especially when considering the limited resource capability on the mobile device. For example, the limited screen area means the device is not suitable for displaying all of the information from a normal html document. The low bandwidth of wireless connection is too expensive to be wasted on delivering unnecessary information.

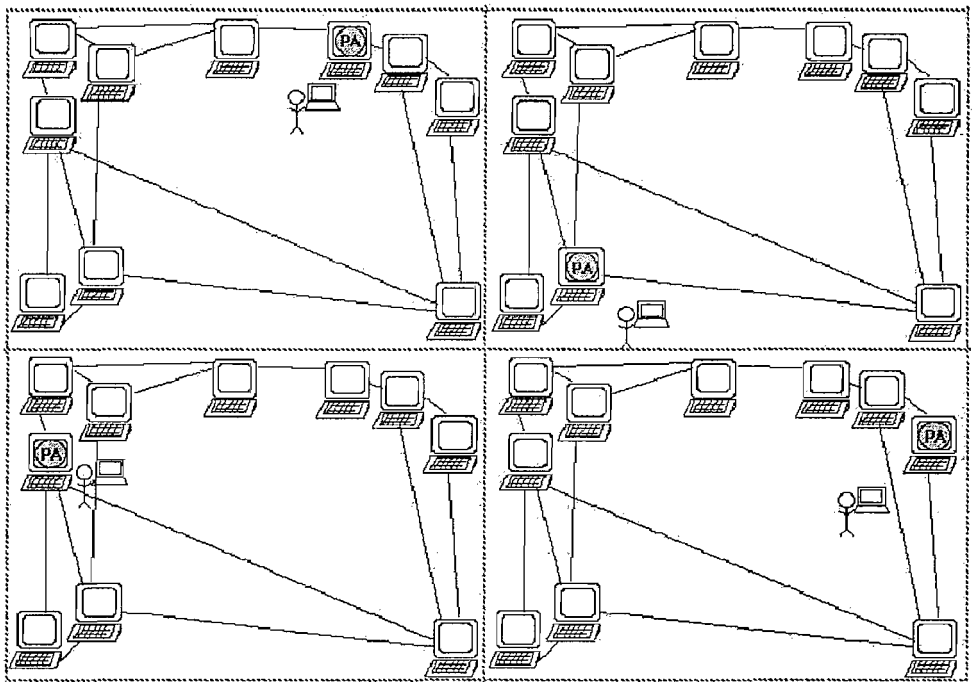


Figure 3.3: Personal Agent System Scenario

As in the M-Commerce framework, this architecture also requires some agents to run on the mobile device. This requirement can only be fulfilled by devices powerful enough to run an agent platform and support multiple application threads. Another issue is in the event where the mobile device cannot reach its personal agent, such as when the personal agent's platform is disconnected from the network. This paper does not explicitly explain the solution to this situation. One possible solution is to allow a new personal agent to be created for the user in order to carry on with new requests.

Communication between personal agents and information servers occurs over sockets. If the communications involved were complicated and data-intensive, there would be a long delay in getting the required information. One approach to overcoming the problem is by sending a mobile agent to the information server to interact with the information server locally. Furthermore, sending more than one mobile agent to simultaneously gather the needed information across the network would lead to faster performance of the system.

3.2.3 MobiAgent

The MobiAgent system [63] is another system that uses mobile agents to serve mobile users. The system consists of three main components connected together by both wired and wireless links. The components in this system are a handheld mobile wireless device, an agent gateway and network resources (cf. Figure 3.4).

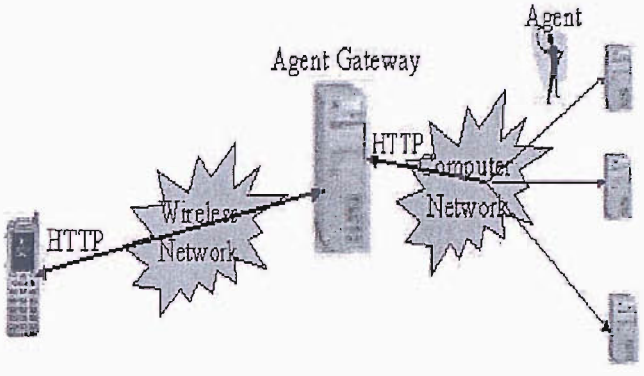


Figure 3.4: MobiAgent System Architecture

In this architecture, the agent platform runs on a remote host called the agent gateway. It is the heart of this system, which is comprised of both static and dynamic agents. Static agents provide resources and facilities to mobile agents that move between network resources, taking advantages of these resources to fulfill their goals. The agent gateway acts as the mediator between the wireless device and network resources. Network resources are basically Internet platforms containing online information. The agent platform supporting the agent gateway is the Voyager system, while no agent platform runs on the mobile devices. The Mobile Information Device Platform (MIDP) is used to develop MIDlets, which are applications that run on the mobile devices.

Once communication is initiated with the gateway, a mobile device downloads a MIDlet. The MIDlet can be used to create a user profile on the agent gateway, after which the user will be able to dispatch an agent to do some work on the user's behalf. When the agent finishes its task, it goes back to its host environment and pushes the results onto the device. On the other hand, if the user lost connection after the agent was dispatched and remains disconnected when the agent

finishes its task, the agent will go back to its host environment and record its result there. Once reconnected, the user can use a MIDlet to check the results.

Having mobile agents manipulating resources on the network to accomplish the tasks requested by users allows the agents to interact locally with needed resources. Accessing resources locally means fast and reliable communication with the resource, resulting in better performance and faster completion of tasks than with remote communication. The MobiAgent system has a centralized element, the agent gateway, which supports the main functionality of the system, i.e. initiating agents for mobile users and storing results in the event of a mobile user being disconnected. Such a centralized component is a potential single-point failure in a large-scale distributed environment. Besides, the static location of the agent gateway leads to variance in the distance between the gateway and the moving mobile devices. For example, in order to get a result a mobile device has to interact with a particular agent gateway, with which the device was communicating earlier. This applies regardless of the distance between the two components. Again, having a proxy (in this case the agent gateway) as close as possible to the mobile device is the best solution for achieving more reliable and faster communication, especially when wireless communication is involved, as the proxy can reduce bandwidth utilization.

3.2.4 Mobile Agents Platform (MAP)

Mobile Agents Platform (MAP) [63] is a platform for the development and the management of mobile agents, implemented at the University of Catania. It is used to develop a system that supports mobile users with the architecture shown in Figure 3.5. The system consists of one or more lookup servers, and some data servers. The lookup server maintains the information about where an agent temporarily stores its results while waiting for the user to reconnect to the system. Data servers are simple nodes on which agents store their results, in cases when they detect that the user is no longer connected to the system.

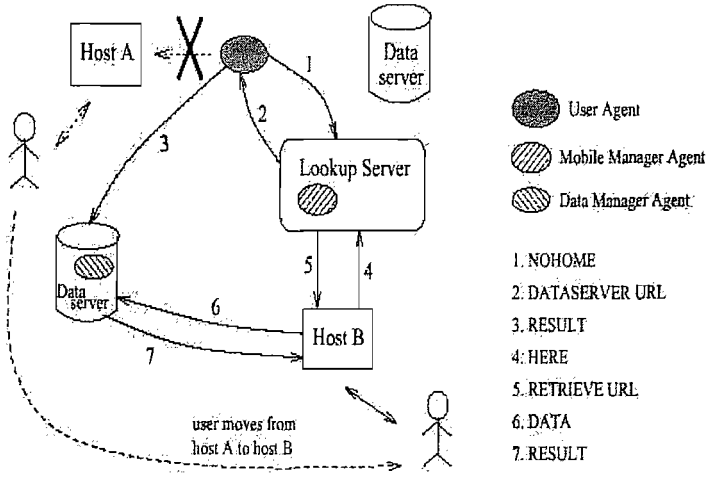


Figure 3.5: Mobile Computing in Mobile Agents Platform

Figure 3.5 shows the steps performed to serve a mobile user's requests. These steps are described below:

1. A user agent X initiated by a user failed to communicate with the user's mobile terminal called "node A" due to the user's disconnection.
2. The mobile manager agent, once notified about this failure, selects a data server and sends the information "DATASERVER URL" to agent X. Then it updates a table inside the lookup server, by storing the user's ID, the agent's ID and the URL of the data server communicated.
3. Agent X sends a message "RESULT" to the agent data manager present in the data server. Agent X specifies its ID, the user's ID, and the data to be stored. Then the agent ends its execution. The data manager agent will then store the data received on the data server.
4. When the user reconnects from a new station, i.e. "host B", the server MAP sends a message "HERE" to the mobile manager agent, to inform it of the user's reconnection and the new URL from which the user has connected.
5. The mobile manager agent refers to the table of the lookup server, after which it returns to the user all information needed to retrieve the results.
6. The server MAP on which the user is, then sends a request "DATA" to the data manager agents.
7. Data managers respond by returning the requested results.

The Mobile Agents Platform (MAP) architecture includes a centralized component, the mobile manager agent. Its function is to allocate a particular data server for an agent to store results on if the mobile device is disconnected, and to hold a lookup table for result discovery by the user once reconnected. In the case of the mobile user reconnecting to the wired network via a platform located at a long distance from the mobile manager agent, the multiple communication steps, such as having to query the lookup table for a data server address and then having to query the data server for the result, seem to be unnecessary.

A simpler alternative would involve less interactions, i.e. when the device reconnects it can directly request the result from a platform, provided that the address of the platform is known in advance. This would make the communication much simpler, through not having to discover the address of the data server from the lookup table before getting the result. Besides, agent X should not necessarily terminate itself when it finishes its tasks if the user is disconnected, as it can still be used by the user once reconnected. Agent X can also be the one migrating to the new location of the user, carrying the results of the previously accomplished tasks.

3.2.5 TACOMA Lite

Another mobile-agent-based application for mobile users is presented in [45], specifically applications on a PDA, which is supported by the TACOMA system. In TACOMA, an agent can be shipped among sites that support the TACOMA system. The shipping is done in a so-called briefcase, an encapsulating structure containing a set of folders. The folder contents can be arbitrary data or code. Examples include the code folder containing the source code of an agent, and the data folder containing agent specific data, e.g. an image that the agent will process. Folders can also be left in *file cabinets* for persistency, at sites an agent visits (cf. Figure 3.6).

The TACOMA application programming interface (API) is implemented as a set of library functions operating on these fundamental abstractions. In addition, TACOMA has a meet operation executed by an initiating agent. The TACOMA functionality is composed of two entities termed the firewall and the system agents. Conceptually, the TACOMA firewall controls whether mobile code should be allowed to enter the host or not. The firewall also sets up and

controls the runtime environments for mobile code that runs within it. System agents are preinstalled non-mobile agents that serve as an addition to the TACOMA firewall. Examples of system agents are the agents that compile or interpret mobile code. Another type of agent in TACOMA is the mobile agent which uses the system agents to, for instance, migrate to a remote host, install mobile code or notify a user about an exceptional event in the computing environment.

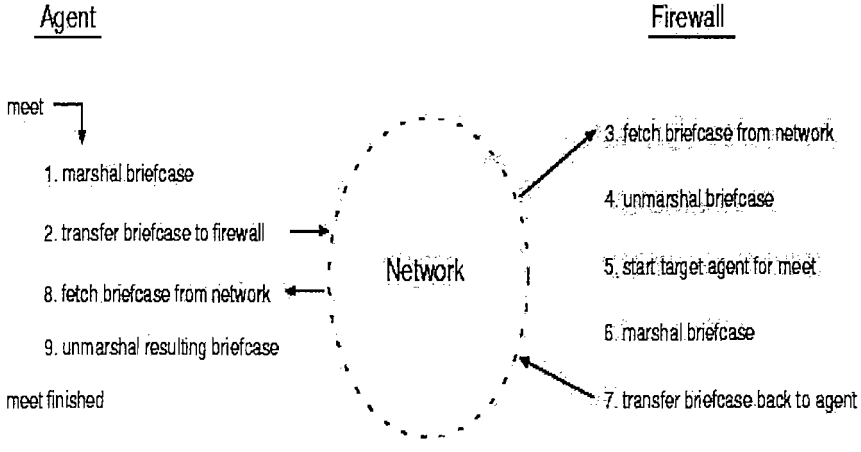


Figure 3.6: Execution steps in meet operation

With the limitations of PDAs in mind, there are several aspects of the initial TACOMA system architecture that have to be revised. The initial architecture is based on two layers. The first layer is the TACOMA firewall, offering the basic services necessary for mobile code to execute. The second layer is the applications using the TACOMA API, which makes it possible for them to, for instance, migrate to a remote firewall.

Considering the characteristics of the PDAs, this architecture is inappropriate for PDAs in several respects. First there is a good chance that the PDA is disconnected, or connected via unreliable and slow links. When, however, a PDA is connected, this is typically for short periods, for instance, when synchronizing data with a networked host. Furthermore, a PDA cannot provide large amounts of stable storage for mobile agents. Since a PDA has low processing power, it is unable to serve numerous concurrent mobile agents. Finally, the initial architecture does not take advantage of the right coupling between the PDA and the host it uses for synchronizing data.

Consequently, an extra layer is added to the initial TACOMA architecture. This layer consists of an entity called the hostel. Essentially, the hostel is the host the PDA normally uses to synchronize data with. The hostel is also assumed to act as the network provider or proxy for the PDA, i.e. the hostel is a networked workstation. The paper outlines the need for the hostel component in the application. For instance, the mobile agent gathering information on the wired network assumes the presence of a host that they can inquire from in case the PDA is not connected. Moreover, since a PDA cannot compile, for instance, C programs to executables, access to a host that possesses the ability to do so is needed. As such, the hostel is placed between the TACOMA API and the TACOMA firewall in the TACOMA Lite architecture.

In the presented architecture, each PDA has its own *hostel,* which is a fixed proxy on the network. It is a host with which the PDA always synchronizes its data. This application is suitable for a user who does not move around too much and for whom the hostel is the only connection point needed for the PDA.

In the case of a user always on the move and needing to connect to different hosts, the application is unsuitable. In order to have a generic application, the user should be given alternative proxies, or the proxy can be mobile.

3.2.6 MONADS

Different scenarios of agent communication in the wireless environment are presented in [38]. Specifically five scenarios using agent-based systems are described (cf. Figure 3.7). The first scenario is where the mobile device is powerful enough to run a full agent system and a number of agents. Agents at the mobile device communicate over the wireless link with other agents at fixed network, and possibly transfer themselves between the mobile device and the fixed network. A performance model is also presented in the paper, which can be used as a basis for selecting the appropriate action: whether to migrate or whether to use message-passing over the wireless link.

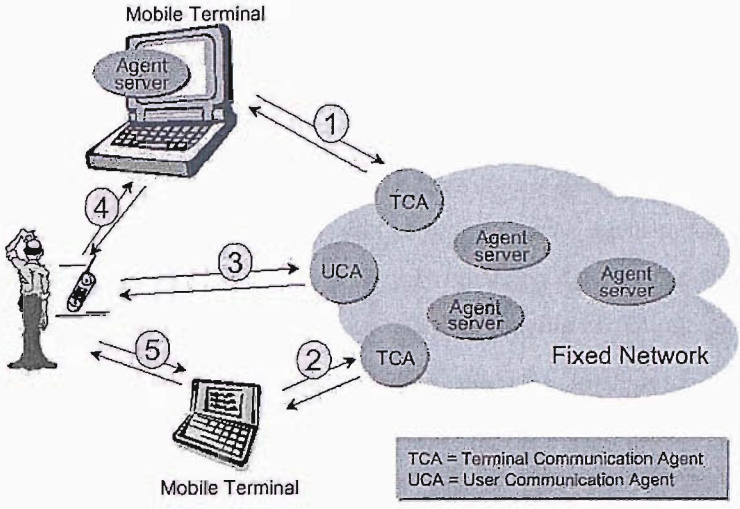


Figure 3.7: Agent communication scenarios

In the second scenario, the mobile device is a low-end device such as a PDA, which is unable to run a full agent system due to its hardware and software limitations. However, having a stand-alone agent control tool in a mobile device, a user may, for example, start agents in the fixed network and get results back to the mobile device even if the mobile device is itself unable to run agents. In both of these cases, the mobile device communicates with a terminal communication agent (TCA) located on the fixed network. The TCA acts as the proxy for mobile device while the mobile device is in a disconnected state. Furthermore, the TCA takes care of locating mobile devices whose IP-addresses change frequently. Once a mobile device reconnects to the fixed network, possibly with a new IP-address, it informs its TCA about its new IP-address. The TCA takes care of forwarding this information to the appropriate agents.

In the third case, the user may have for example a digital phone, which is used to control agents on the fixed network, and agents can use this phone while reporting results back to the user. For example, an agent may send a GSM short message (SMS) to the user once it has finished its task, or the user may use the same technology to cancel an agent operation without establishing a data connection to fixed network. For this kind of communication, there is a User Communication Agent (UCA) located on the fixed network. Cases 4 and 5 are communication between the user and the agents in the mobile device. This is similar to any communication between the user and the agents – the only difference is in the user interface. In powerful devices an agent may have an

advanced graphical user interface, but in some cases the mobile device hardware limits the agent's user interface.

The paper proposes several possible implementations for mobile user applications. An approach is presented for mobile devices that are incapable of running a full agent system. In this case, a mobile user is allowed to start agents on the fixed network using the agent control on the mobile device. Sometimes it is better to communicate by message-passing or remote method invocation than by an agent migrating. This is due to the fact that some communication involves only simple interactions between agents, which are much faster and convenient if done remotely. An agent migrating sometimes involves high bandwidth and sometimes a long time is needed to transfer the agent. These would be wasted if the migrated agent was not fully used or there was an easier alternative way to accomplish the task.

An important issue raised by the paper is whether the Terminal Communication Agent (TCA) is mobile. As the communication involves mobile devices expected to change location frequently (possibly to a location far away from the home network), having a mobile TCA would be an advantage. With the property of mobility, the TCA is able to move to the closest platform to the mobile device. The function of the TCA explained in the paper is to forward the information on the current location of the mobile devices to appropriate agents. By having a TCA, which is at a long distance from the mobile device, all agents wanting to communicate with the mobile device need to request its new location from the TCA before starting to communicate with the device. This would involve multiple interactions, possibly over a long communication channel.

By having a mobile TCA that tries to move as close as possible to the mobile device, the communication distance between itself and the mobile device may be reduced. Only one interaction would be required, as the TCA would be responsible for relaying the message directly to the mobile device. Locating a mobile TCA would not be a problem as mobile agents have mechanisms to track their current location.

3.2.7 Discussion

Looking at the existing architectures applying mobile agent technology to support applications for mobile users, some patterns are recognisable. Most of the architectures use mobile agents to gather information and to take advantage of the available resources on the wired network. In some cases, only some mobile agents act on behalf of the users and receive requests directly from the users, while other agents are delegated to do tasks for them.

For each mobile device connected to the wired network, there is normally a proxy that serves its requests. In some of the architectures, the proxy is a platform communicating with the mobile device, such as the *hostel* in TACOMA and the *agent gateway* in MobiAgent. In some other architecture, the proxy is an agent acting on behalf of the mobile user on the wired network such as the *personal agent* on Dartmouth Personal Agent System and the *service agent* on M-Commerce Framework. In most architecture, the proxy is defined as a stationary entity representing a user on the wired network. Having such a proxy requires the mobile device to communicate with it once the user connects to the wired network, regardless of the proxy's location. This raises many disadvantages, such as in the case of there being a long distance between the proxy and the mobile device. The need to communicate through a long communication medium leads to problems such as high latency, heavy network load and high error rates. Another possible problem is a situation in which a mobile device is unable to reach its proxy on the fixed network. This can happen due to the disconnection of the proxy from the network. Though this is an important issue to consider, it is not raised in any of the papers studied in this chapter. One possible solution to this problem is simply to allow the mobile device to communicate with a new proxy.

Some architecture requires agents to be run on the mobile device or to migrate to the mobile device. Such architectures prevent low capability devices from using them. In [38], an alternative approach is presented where a mobile device, which does not have the capability to run an agent system is allowed to create and control agents on the fixed network. This approach would allow all kinds of mobile devices to use the system. The issue of resource discovery in the current environment of the proxy was taken into consideration by some of the architectures, for example the application *mobile manager* in Mobile Agent Architecture.

Observations and issues that have been raised on the existing architectures applying mobile agent technology for mobile users are useful in our effort to design an architecture with the same aim, which is to support mobile user applications.

3.3 Conclusion

We have introduced the concept of mobile agent, presented its advantages and described some mobile agent systems. Based on the advantages it offers, mobile agent technology is used in many different areas including mobile users' applications. We have studied applications that are developed for mobile users and discussed their approaches. Our aim is to build a system for mobile users and we have identified that mobile agent technology will be useful in its implementation. Based on the study presented in this chapter, we draw a conclusion below that will be the basis of our work, which will be presented in the next chapter.

Having a mobile agent performing tasks on the fixed network on behalf of a mobile user allows a user's request to be accomplished without relying on the presence of a connection with the user's mobile terminal. The mobile agent can be regarded as the mobile user's proxy as it can also act as the user's representative on the fixed network, as well as performing tasks on behalf of the user. Considering the nature of a mobile user, having a mobile agent as a proxy on the fixed network is advantageous as it is capable of moving to a location closer to the user, helping to provide better and more reliable communication. The mobile agent can perform the task requested by the user by itself or by delegating it to other agents or applications on the network.

In the case of mobile users being disconnected before they receive results from their mobile agents, there should be a means of returning results back to the user. The mobile agent should be able to detect the presence of the mobile user once reconnected and the exact location of the user. Whenever the mobile user reconnects to the network, the mobile agent will migrate to a platform close to the mobile terminal. This can be any platform located in the same local network as the mobile terminal. Having a mobile terminal and its mobile agent in close proximity to each other allows for faster and more reliable communication than with remote communication.

The mobile terminal does not necessarily have an agent system installed on it. It can be installed on the mobile terminal if the terminal is powerful enough to run it. For a generic architecture, we

just consider an application that can be installed on any type of mobile terminal. Thus an agent platform is not required on the mobile terminal. The mobile terminal may have a local document or state which it carries around, such as personal schedules or contact information. This local information should be of reasonable size and be able to be manipulated by the user when offline. When a user is connected to the wired network, the documents on her terminal can be transferred to the wired network to be manipulated or shared and promoted to other online users. The architecture would also need to provide the means to discover the services available on the network, for example a service directory or a local lookup table that can be used by the mobile agent to discover other local applications or resources.

In the next chapter we formalize these ideas, and introduce a middleware system based on mobile agent technology to serve mobile users. This system, which addresses limitations in mobile users' environments, is introduced as a mechanism to support applications developed for mobile users, an example of which is the collaborative editing application for mobile users (cf. Chapter 6).

Chapter 4

Mobile-Agent based Middleware for Mobile Users

So far we have overviewed the limitations and challenges in mobile users' environments and we have also discussed some applications and their approaches to addressing the problems raised. Our survey has shown that mobile agent technology is a promising solution to limitations in mobile computing environments. Thus in this chapter, we introduce a mobile-agent based middleware supporting mobile users as our solution to the problems.

4.1 Overview

In this chapter we present two contributions, namely an abstraction layer providing a set of services in the form of an API, and an architecture for a middleware application. The middleware application, which we refer to as Mobile-Agent based Middleware for Mobile Users (MAMiMoU), acts as an abstraction layer that hides away communication and coordination details from the applications it is supporting. Its architecture makes use of multiple mobile agents to serve mobile users and to support coordination between them.

Being an abstraction layer, MAMiMoU allows applications for mobile users to be constructed on top of it, enabling transparent communication between applications hosted on mobile terminals and network-based applications (cf. Figure 4.1). In doing this, the middleware addresses limitations in mobile computing environment that include hiding intermittent connectivity and the mobility details of users from applications. MAMiMoU offers services in the form of an API to applications, which will be described in detail in Section 4.2. In its general form, MAMiMoU offers a substrate for building distributed applications across mobile terminals and fixed

infrastructure. We will demonstrate this by showing that MAMiMoU can be used to build a collaborative editing application for mobile users.

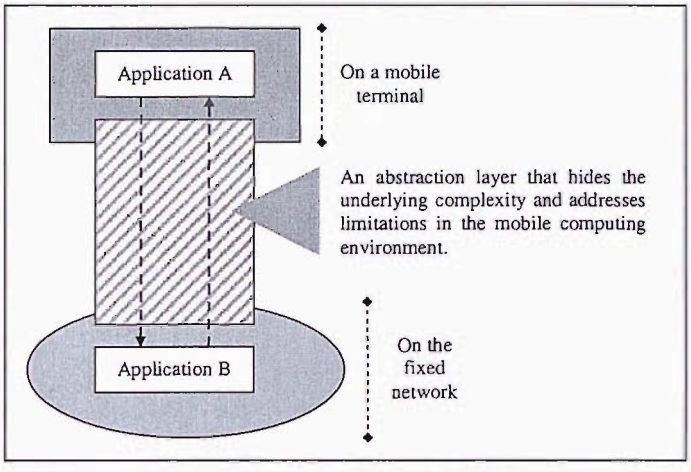


Figure 4.1: The middleware as an abstraction layer

The second contribution presented in this chapter is the architecture of our middleware. We present MAMiMoU's architecture on several levels, starting with a general overview of the architecture, and followed by descriptions of the lower layers of the middleware. The middleware is based on mobile agent technology, with a mobile agent as the main component of the middleware. This is based on the generic organisation illustrated in Figure 4.2 [71], where the mobile agent is acting as an intermediary between the mobile terminal and applications hosted on the fixed infrastructure, allowing applications on mobile terminals and the fixed infrastructure to interact with each other. To support such an organisation, we describe the architecture in detail, including the middleware's components and interactions and the coordination between them. Together with the description of the architecture, we analyse the rationale behind our design and describe the advantages it offers.

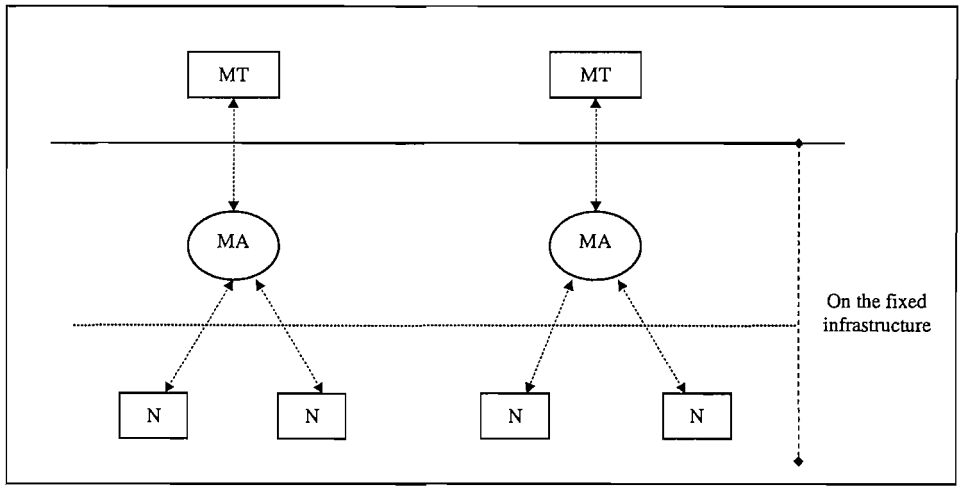


Figure 4.2: Generic organization

4.2 Middleware Services

MAMiMoU offers services to two types of applications, namely applications that are hosted on mobile terminals and network-based applications. To use these services, the applications can invoke the API provided by the middleware. The API offered to mobile terminal applications allows them to create or remove network-based applications and to transparently send and receive messages from them. In Figure 4.3a, application A, which is an application hosted on a mobile terminal, can invoke a create or remove request on MAMiMoU, after which MAMiMoU will create or remove the respective network-based application, i.e. Application B. Application A can also send a message to Application B by first forwarding the message to MAMiMoU (cf. Figure 4.3b). As for network-based applications, MAMiMoU allows them to send and receive messages from mobile terminal's applications. Application B can first send a message to the middleware, which will forward the message to Application A (cf. Figure 4.4).

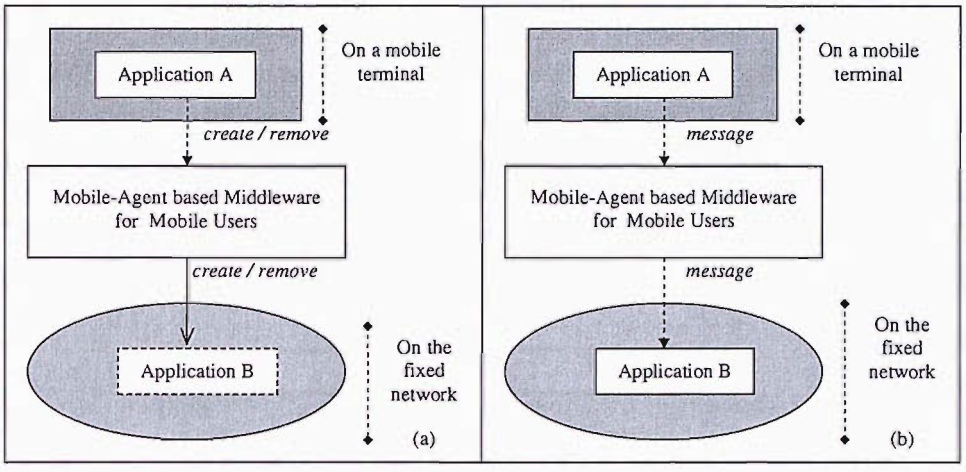


Figure 4.3: API offered to mobile terminal applications

Being served by the middleware, network-based applications are not required to be aware of the user's mobility and connectivity or to keep track of the mobile terminal's current location. As for the mobile terminal's applications, they are not required to store details of the network-based applications they are interacting with. Associations between interacting mobile terminal applications and network-based applications are stored and maintained by MAMiMoU.

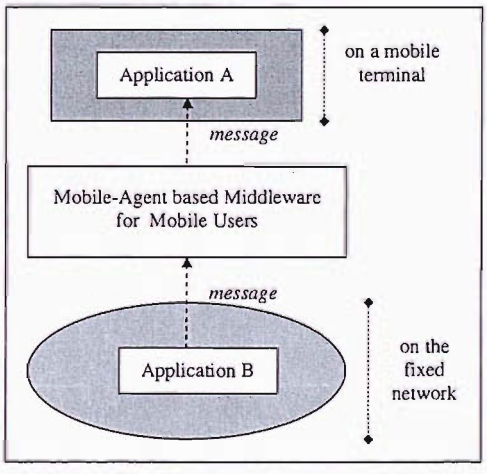


Figure 4.4: API offered to network-based applications

Figure 4.5 summarizes a list of operations provided by MAMiMoU to mobile terminal-based and network-based applications. The createApplication($type^{app}$) API is invoked by an application hosted on the mobile terminal to request the creation of a network-based application of the type $type^{app}$, while removeApplication($id^{n\text{-}app}$) is invoked to remove a network-based application

identified as $id^{n\text{-}app}$. Only the network-based application's creator is allowed to remove it. The sendMessageOut($id^{n\text{-}app}$,message) function is used by an application on a mobile terminal to send a message to a specified network-based application, known as $id^{n\text{-}app}$. To get a message from the network-based application identified as $id^{n\text{-}app}$, getMessage($id^{n\text{-}app}$) is invoked by a mobile terminal's application. This request will fail if the required message does not exist. For network-based applications, sendMessageIn($id^{mt\text{-}app}$,message) is invoked to send a message to an application on a mobile terminal identified as $id^{mt\text{-}app}$.

API
createApplication($type^{app}$)
removeApplication($id^{n\text{-}app}$)
sendMessageOut($id^{n\text{-}app}$,message)
getMessage($id^{n\text{-}app}$)
sendMessageIn($id^{mt\text{-}app}$,message)

Figure 4.5: API

4.3 Middleware Architecture

The main feature of our middleware is the intermediary, i.e. the mobile agent, which always attempts to be in the vicinity or in the same locality as the mobile terminal. It follows the mobile terminal by migrating to the current local environment of the mobile terminal (cf. Figure 4.6). This happens every time the mobile terminal changes its location. Because of this behaviour, we call the mobile agent a *shadow*. The shadow acts on behalf of the mobile terminal on the fixed infrastructure, carrying out functions for the mobile terminal.

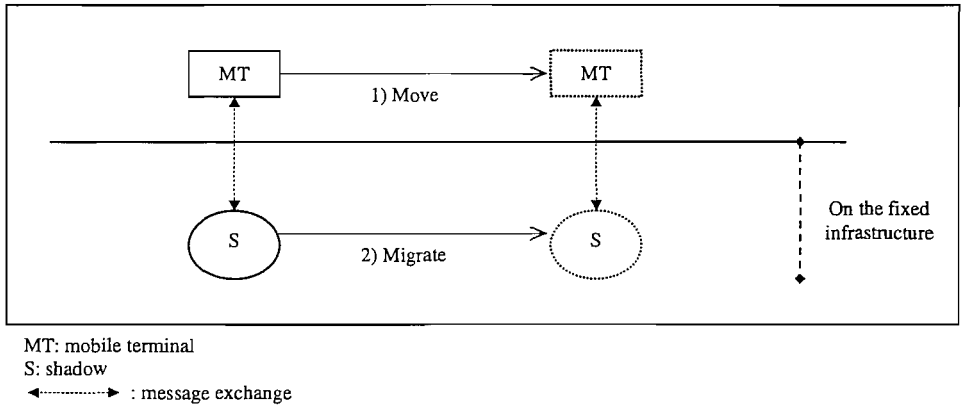


Figure 4.6: Shadow following mobile terminal

In some cases, a shadow may fail to migrate to the current vicinity of the mobile user (cf. Figure 4.7). This may be because the network in which the shadow is currently running is not connected to the rest of the Internet, or vice versa, i.e. the new user's shadow is not connected to the Internet. To deal with such problems, our approach is to dynamically create a new shadow in the current user's vicinity. However, this could lead to a situation where a user is associated with multiple shadows, in which case each shadow holds information about the user's mobile terminal. Maintaining all these shadows would be troublesome since the user's states would be distributed across all the shadows and would be needed most of the time. Furthermore, these shadows could be distributed across the network. To solve this problem, our approach is to allow the existing shadows to merge their tasks together whenever network connectivity allows, and hand the accumulated tasks over to a single shadow, which we refer to as the main shadow. Such reconciliation or hand over of tasks between shadows will be described in detail in Section 4.3.3.4.

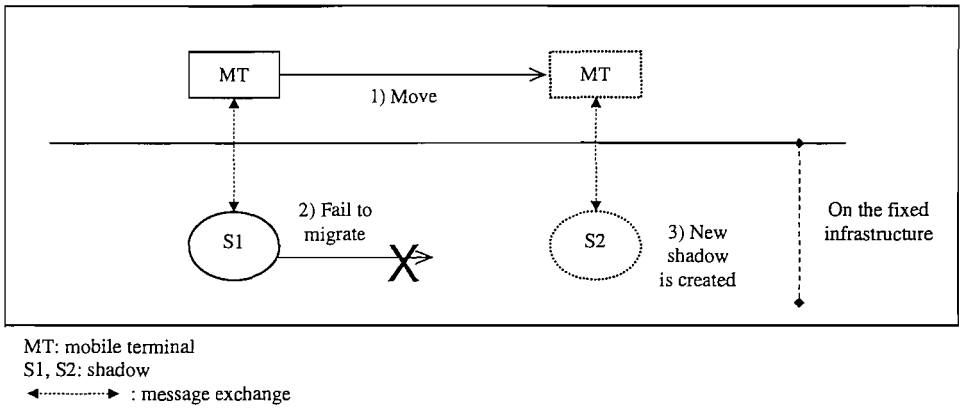


Figure 4.7: Dynamic creation of shadow

4.3.1 Components

The middleware comprises of three components, namely the mobile terminal, the shadow handler and the shadow. Each component will be described together with the assumptions we make concerning its communication capabilities.

4.3.1.1 Mobile terminal

A mobile terminal has the ability to connect to a network in its vicinity. It may use specific methods to communicate with network hosts, e.g. infra-red or Bluetooth. We assume that the terminal is allocated an address, which may change as the terminal connects to another network; alternatively, the terminal can be supported by Mobile IP, in which case the mobile terminal is assigned a permanent home address. Such address can be used by networked entities to communicate with the mobile terminal.

4.3.1.2 Shadow handler

A shadow handler acts as a local daemon in a local network, the first contact point of a mobile terminal with the local network. It is responsible for starting or migrating shadows on behalf of mobile terminals.

4.3.1.3 Shadow

A shadow is a mobile agent, acting as an intermediary between a mobile terminal and infrastructure applications. Being able to migrate allows it to move "closer" to the mobile terminal in order to communicate with it using the address allocated by the local network. The shadow acts on behalf of the mobile terminal on the fixed infrastructure, carrying out five main functions for the mobile terminal, namely i) creating and removing network-based applications on the mobile terminal's requests; ii) supporting a store-and-forward mechanism to allow interactions between the mobile terminal and network-based applications; iii) maintaining interaction mappings between the mobile terminal and network-based applications; iv) maintaining details of network-based applications that have been created by the mobile terminal; and (v) migrating to a location closer to the mobile terminal whenever the mobile terminal changes its location, network connectivity permitting.

4.3.2 Component Interaction

Our architecture may be summarized as follows. When connected to a network, a mobile terminal makes contact with a shadow handler, and requests it to migrate its shadows to the shadow handler's location. If no user's shadow is active, the shadow handler creates a new shadow for the mobile terminal. In the simplest case, a single shadow exists. If migration is successful, the shadow interacts locally with the mobile terminal. The shadow spawns new applications as requested by the terminal and forwards messages to and from them; in essence, the shadow acts as a router of messages to the applications. Communications between shadow and applications are robust to the migration of shadows, based on a transparent routing algorithm [69][70]; on the other hand, communications between terminal and shadow may fail as the terminal changes location. If the migration of all shadows fails, a new shadow is spawned locally, and the terminal keeps a log of all created shadows. When several shadows are requested to migrate to a specific destination, the first shadow to reach the location is assigned to be the main shadow; the others coordinate with it to offload information about the applications to which they were routing messages.

4.3.3 Coordination Model

Here we further describe the components' interaction by presenting a coordination model that categorizes interactions between MAMiMoU's components into several phases. Based on this model, a detailed protocol and algorithm is designed to coordinate interaction between components of the architecture (cf. Chapter 5). Specifically, the interactions are categorized into five phases, namely i) migrate shadow phase, ii) shadow arrival phase, iii) hand over phase, iv) shadow locating phase and iii) shadow termination phase. A phase may involve activities to be performed by at least one component, which will be described by several diagrams. A perfect scenario is first explained for each of them, followed by an explanation of possible failures and a handler for each failure. In describing the interactions, we use diagrams that show message exchange between the components and the location of each component. Reception of a message by a component triggers another activity to be performed internally by the component, while the failure to receive an awaited message may result in a timeout in the component and a failure handler being activated. The classification of failures in the interaction model will be further explained in the next section.

In the rest of the thesis, we will use the word *platform* to refer to a fixed terminal that is able to host a shadow-handler (SH) and shadows. A local platform is one that is located in the vicinity of the mobile terminal, i.e. connected to the same local network; and its availability is advertised by a local service directory. In the coordination model, we consider only communication failures that may occur between components running on different platforms.

A shadow can be the main shadow of a mobile terminal, which is responsible for creating or sending messages to and from applications on the fixed infrastructure; and for locating other shadows of the user. A mobile terminal stores information on all of its shadows in a list of shadows (LS), while a shadow stores information on applications in a list of applications (LA). Both mobile terminal and shadow have lists of messages called list of outgoing messages (LOM) that queues outgoing messages, list of incoming messages (LIM) that queues incoming messages, and list of requests (LR) that queues requests to create or remove network-based applications.

To identify a component, we define the component's identifier, ID. For example the ID for a component X will be denoted as ID^X. A mobile terminal holds its main shadow's identifier in a variable ID^{MS}, while the identifier of a regular shadow, S, is held in a variable ID^S. In a shadow,

the identifier of an application, App, is held in a variable ID^{App}. A shadow handler is responsible for starting a shadow on a mobile terminal's request and this operation will be represented as "perform(startShadow)", while the migration of a shadow will be denoted as "perform(migration)" in the diagrams. A summary of the notations used in the diagrams is presented below:

Variables:	
LS	List of shadows' details
LA	List of applications' details
LA^{new}	List of new applications received from other shadows
LA^{temp}	Temporary list of applications
$LA^{useRouter}$	List of applications that require router
LA^{router}	List of applications that are using shadow as router
LOM	List of outgoing messages
LIM	List of incoming messages
LR	List of requests
ID^{MS}	Main shadow's identifier
ID^{S}	Shadow's identifier
ID^{App}	Network-based application's identifier

Operations:	
perform(startShadow)	Spawn a new shadow
perform(migration)	Migrate to a new location

In Figure 4.8, we show the connection between the coordination phases that will be described in the rest of this section. We illustrate four different components, each with different phases to be carried out. New phases are started by a number of different triggers:

- The value of a state in the component has changed, e.g. the *online* state changes to *true*, which activates the migrate shadow phase in MT,
- When another phase has finished, e.g. the completion of shadow arrival phase in MS allows it to proceed to the shadow locating phase
- When the same phase in other components has started, e.g. the start of the shadow termination phase in S initiates the same phase in MS and MT.

We now describe each phase individually.

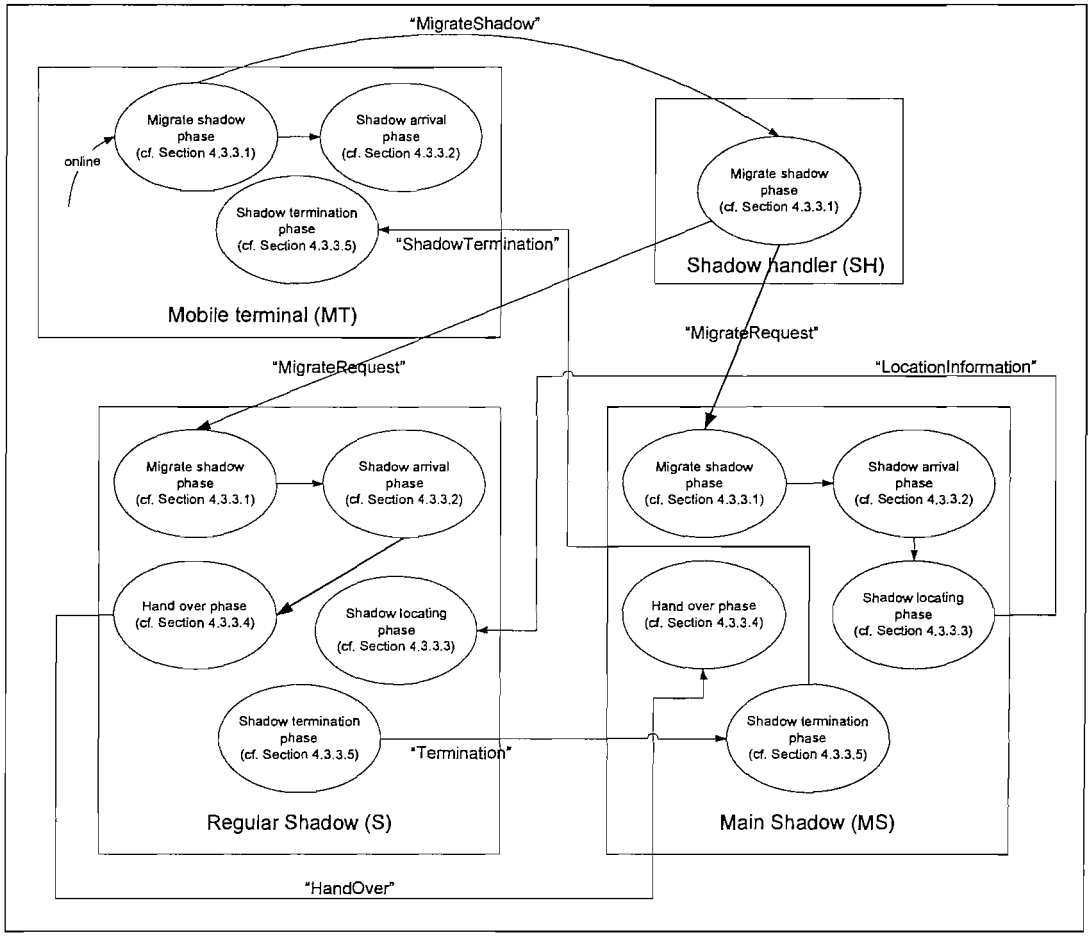


Figure 4.8: Connections between coordination phases

4.3.3.1 Migrate shadow phase

Once online, a mobile terminal starts with the migrate shadow phase. In this phase, the mobile terminal tries to migrate its shadows closer to its current location.

4.3.3.1.1 Migrate shadow phase: perfect scenario

A mobile terminal starts this phase by sending a query for a shadow handler to a local lookup service. Once the information on local shadow handlers is received, the mobile terminal starts sending a request to one of the shadow handlers. Sometimes, a mobile terminal may not receive any information on a shadow handler from the first lookup service it queries, in which case it would send the same query for shadow handlers to another lookup service in the local network. In this model, we assume that each local network has at least one shadow handler operating.

The type of request sent by the mobile terminal to the shadow handler is a "MigrateShadow" request, which consists of a list of shadows' details, LS (cf. Figure 4.9). If LS is empty, the shadow handler starts a new shadow for the mobile terminal. Failure in starting a shadow would be handled by the shadow handler itself.

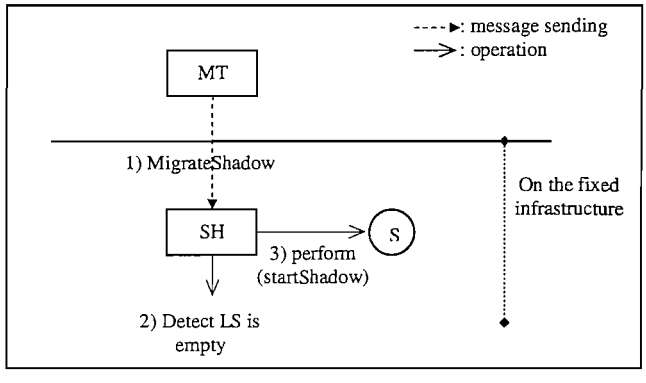


Figure 4.9: Migrate shadows: perfect scenario 1

A non-empty LS indicates that there are already some shadows or a shadow running on the network on behalf of the mobile terminal. On receiving such an LS, the shadow handler would send a "MigrateRequest" message to all shadows in LS, requesting them to migrate to the platform on which it is operating (cf. Figure 4.10).

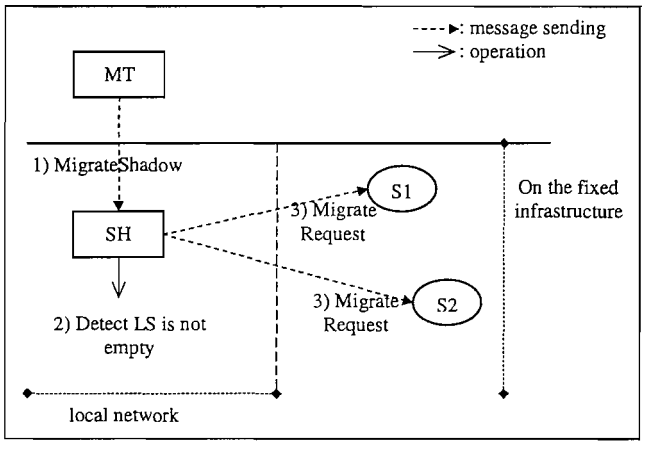


Figure 4.10: Migrate shadows: perfect scenario 2(a)

On receiving a "MigrateRequest" message, a shadow has to make a decision as to whether to migrate or not. By migrating to the shadow handler's platform, the shadow would be in the same local network as the mobile terminal. Figure 4.11 illustrates the migrating shadows. On arrival, these shadows enter the shadow arrival phase, which will be covered in Section 4.3.3.2.

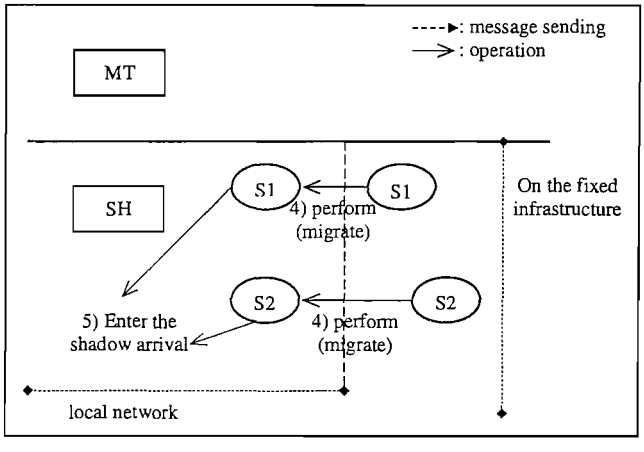


Figure 4.11: Migrate shadows: perfect scenario 2(b)

4.3.3.1.2 Migrate shadow phase: failure cases

In some cases, the mobile terminal may fail to send a "MigrateShadow" request to the shadow handler, SH (cf. Figure 4.12). This may be due to the fact that the SH is no longer operating or connected. When this happens, the mobile terminal sends the same "MigrateShadow" request to another local shadow handler, SH2.

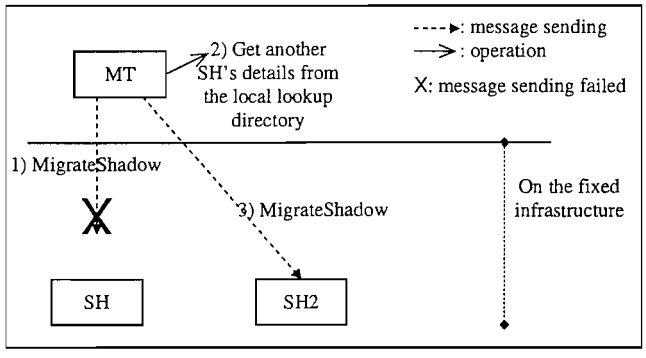


Figure 4.12: Migrate shadows: failure case 1

Figure 4.13 illustrates the failure of a shadow handler (SH) to send "MigrateRequest" to all shadows of a mobile terminal (MT). In this case, SH believes that a failure has occurred in sending such a message and it will start a new shadow for MT. This is made on the assumption that SH will be able to start a shadow on behalf of MT with no failure or the failure being handled by SH itself. It is expected that the newly created shadow, S, will report its existence to MT before a timeout occurs in MT.

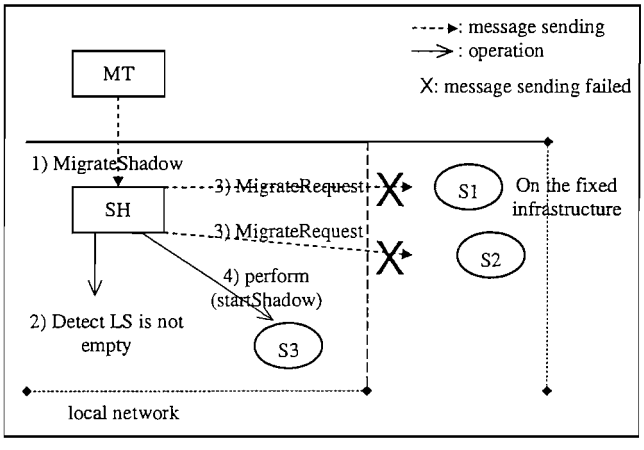


Figure 4.13: Migrate shadows: failure case 2

Although some shadows of MT may have successfully received a "MigrateRequest" message from SH, they may fail to migrate (cf. Figure 4.14). If all the shadows of MT failed to migrate, a timeout would occur in MT and this would drive it to send a "StartShadow" message to SH, requesting SH to start a new shadow.

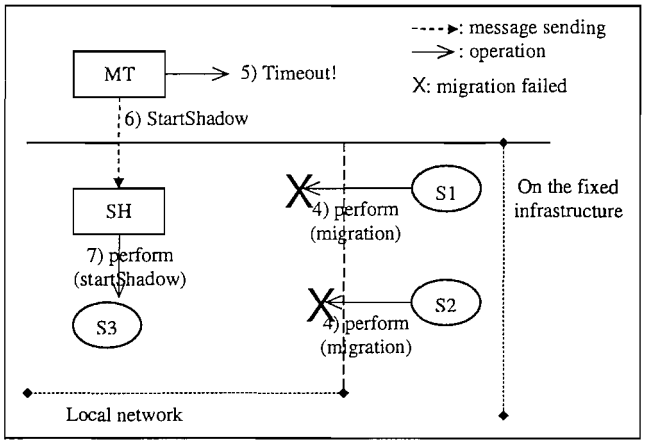


Figure 4.14: Migrate shadows: failure case 3

4.3.3.2 Shadow arrival phase

The shadow arrival phase starts in a shadow once the shadow arrives at a new platform after migration, while for a mobile terminal, this phase starts after a "MigrateRequest" message has been successfully sent to a shadow handler. In this phase, a shadow informs the mobile terminal about its arrival at the new location and waits for a reply from the mobile terminal.

4.3.3.2.1 Shadow arrival phase: perfect scenario

Figure 4.15 illustrates a scenario where MT only has one shadow running on the fixed infrastructure. The shadow, S, begins the shadow arrival phase by notifying its arrival to the mobile terminal, MT, in a "ShadowArrival" message (cf. Figure 4.15). In return, MT sends a "MainShadowAssignment" or a "MainShadowInformation" message to S. A "MainShadowAssignment" message is returned if MT still does not have a main shadow and this message indicates that MT is requesting S to be its main shadow. Otherwise, if MT already has a main shadow, then a "MainShadowInformation" message is returned to S, which contains information on MT's main shadow.

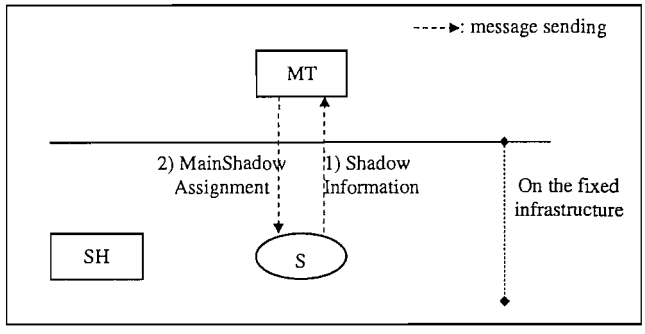


Figure 4.15: Shadow arrival phase: perfect scenario 1

In the case of multiple shadows arriving at a destination platform, MT assigns the first shadow to arrive as its main shadow, while a "MainShadowInformation" message is sent to all other subsequent shadows arriving (cf. Figure 4.16).

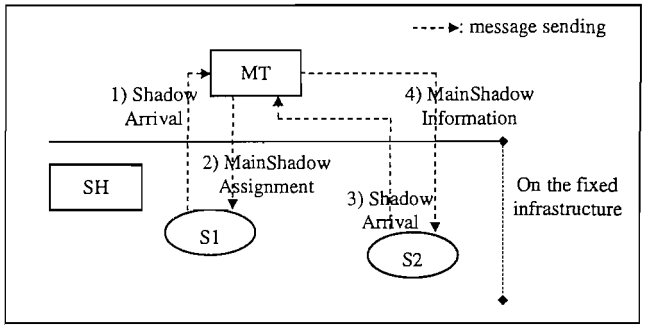


Figure 4.16: Shadow arrival phase: perfect scenario 2(a)

4.3.3.2.2 Shadow arrival phase: failure cases

Figure 4.17 illustrate a case where a newly started shadow, S, fails to inform a mobile terminal, MT, of its existence. After a number of failed attempts to send the "ShadowInformation" message to MT, S terminates itself. This kind of failure may possibly be caused by a broken connection with MT and there would be no way for S to know the new address of MT if the MT had changed its location; vice versa, MT would not be aware of S's existence. Thus in such a case it is safe for a shadow to terminate itself (cf. Figure 4.18).

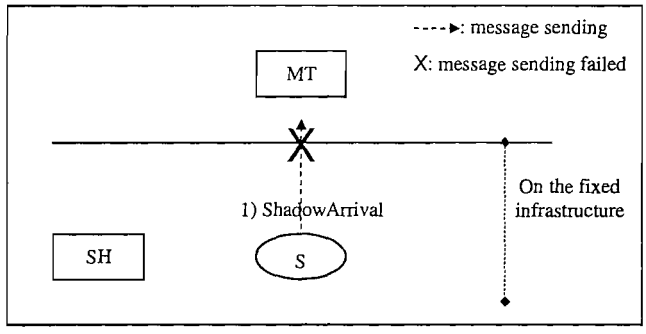


Figure 4.17: Shadow arrival phase: failure scenario 1(a)

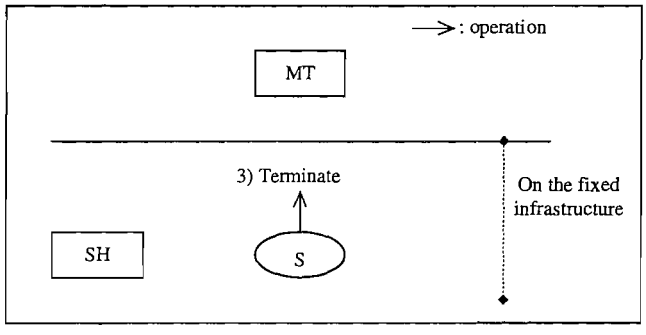


Figure 4.18: Shadow arrival phase: failure scenario 1(b)

In a quite similar scenario, a newly arrived shadow, S1, may not be able to inform the mobile terminal, MT, of its arrival (cf. Figure 4.19). On its arrival, S1 will repeatedly send the "ShadowArrival" message on failure until successful or until it reaches the predefined maximum number of attempts necessary. One reason for such a failure could be that MT has moved to a different location. This is not a problem since, even though MT has not updated S1's location, S1 has left forwarding pointers that can be used by MT to route messages to it (cf. Figure 4.20). In this case, we assume there is a layer that can forward messages to the mobile agent when they move [69][70].

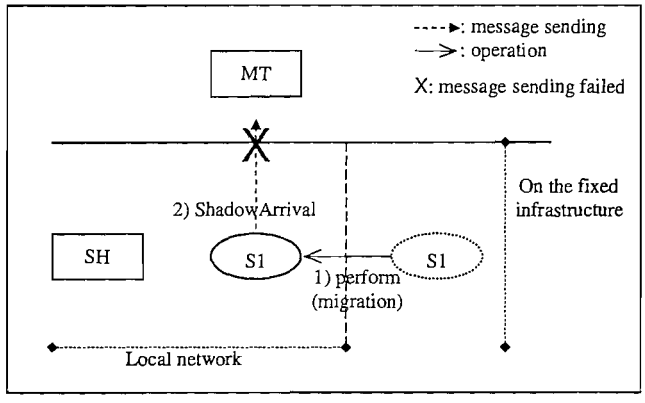


Figure 4.19: Shadow arrival phase: failure case 2(a)

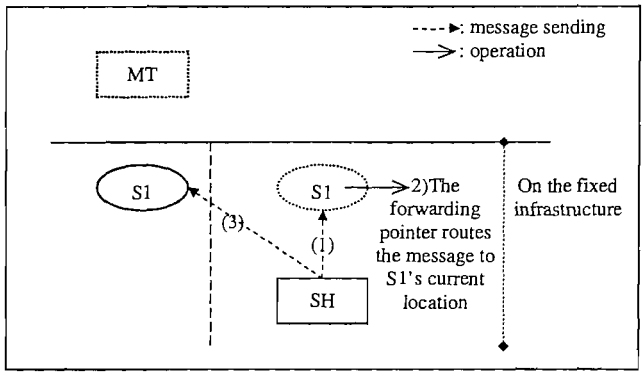


Figure 4.20: Shadow arrival phase: failure case 2(b)

On reception of a message from a shadow informing of its creation or its arrival, the mobile terminal, MT may fail to return to it a "MainShadowAssignment" message or a "MainShadowInformation" message. This may be caused by the disconnection of MT from the platform on which the shadow is operating. With this failure, the shadow would not be able to know the current status, i.e. whether MT has assigned either itself or another shadow to be the main shadow (cf. Figure 4.21).

To solve this, MT once more sends a "MigrateShadow" request to another shadow handler, SH2, in the local network, (cf. Figure 4.22). By doing this, MT's existing shadows may migrate to the new platform, allowing communication to be established with MT.

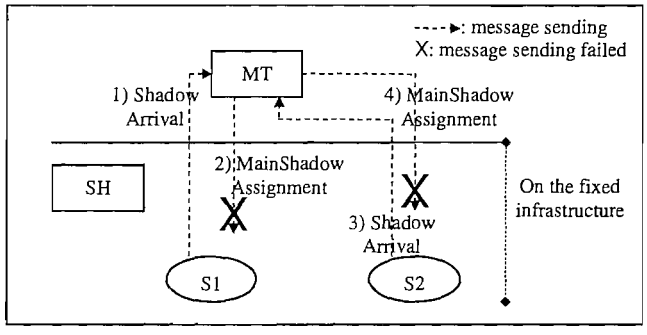


Figure 4.21: Shadow arrival phase: failure case 3(a)

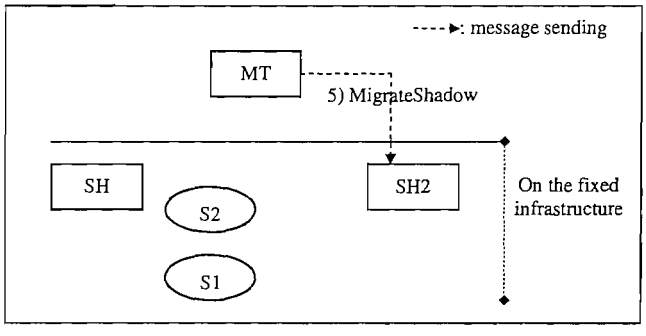


Figure 4.22: Shadow arrival phase: failure case 3(b)

4.3.3.3 Shadow locating phase

One function of a main shadow (MS) is to locate other regular shadows of the mobile terminal (MT) and this function is included in the shadow locating phase.

4.3.3.3.1 Shadow locating phase: perfect scenario

When MS is first assigned to be the main shadow, it receives LS, which is a list of active shadows of MT. From this list, MS is able to know the location of all mobile terminals' shadows, and with this information; MS sends each one of them a "LocationInformation" message, requesting them to migrate. A "LocationInformation" message contains the current location of MT and the platform on which MS is operating as the destination platform for the shadow's migration.

Figure 4.23 illustrates a scenario where MS is sending a "LocationInformation" message to S2 that has already migrated. In this case, MS may send the message to S2's previous location and the message would eventually be routed to S2 by following the forwarding pointer at its previous location.

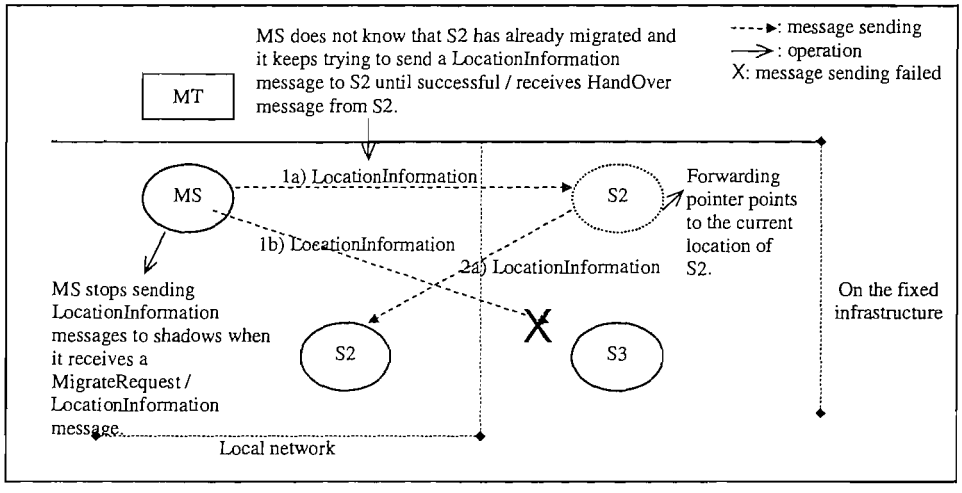


Figure 4.23: Shadow locating phase: perfect scenario 1

As for a shadow, on receipt of a "LocationInformation" message, it immediately recognizes the sender as the current Main shadow and also knows the current location of the mobile terminal. To handle the possibility of a delayed "LocationInformation" message, which may bring outdated information about the main shadow, the message also contains a Main shadow counter. The higher the value of the counter the more recent the information is.

A "LocationInformation" message from a main shadow carries the same request as a "MigrateRequest" message from a shadow handler, i.e. requesting the receiving shadow to migrate to a new location, which is closer to the mobile terminal's current location. On receipt of such a message, a shadow (S3) has the option to perform the request or stay at its current location (cf. Figure 4.24). If S3 decides to migrate, it migrates to the platform on which the main shadow (MS) is operating, and on arrival at the new location, it enters the shadow arrival phase (cf. Figure 4.25).

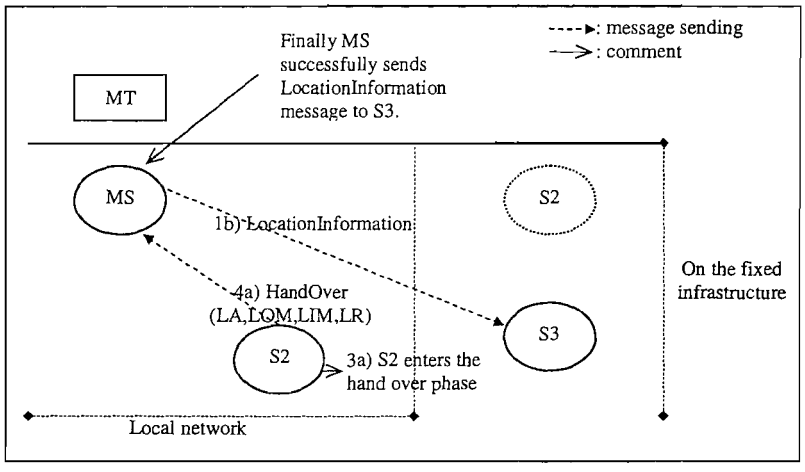


Figure 4.24: Shadow locating phase: perfect scenario 2(i)

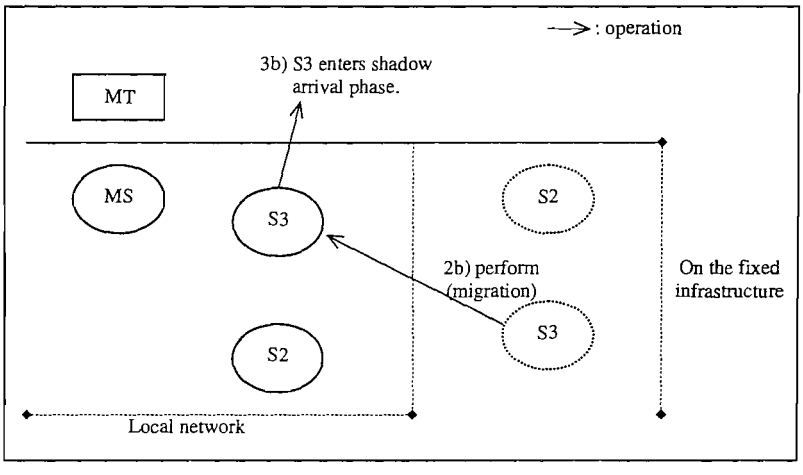


Figure 4.25: Shadow locating phase: perfect scenario 2(ii)

4.3.3.3.2 Shadow locating phase: failure cases

Sometimes, a main shadow (MS) may fail to send a "LocationInformation" message to a shadow. In this case, MS repeatedly sends the message to the shadow on failure until the message is successfully delivered or MS ceases to be the main shadow. This is performed to handle a temporary disconnection, in which case the shadow eventually receives the message (cf. S3 in Figure 4.23 and 4.24).

As illustrated in Figure 4.26, a shadow (S3) that has failed to migrate to a new location remains at its current location, from where S2 remotely transfers its functions to the main shadow (MS).

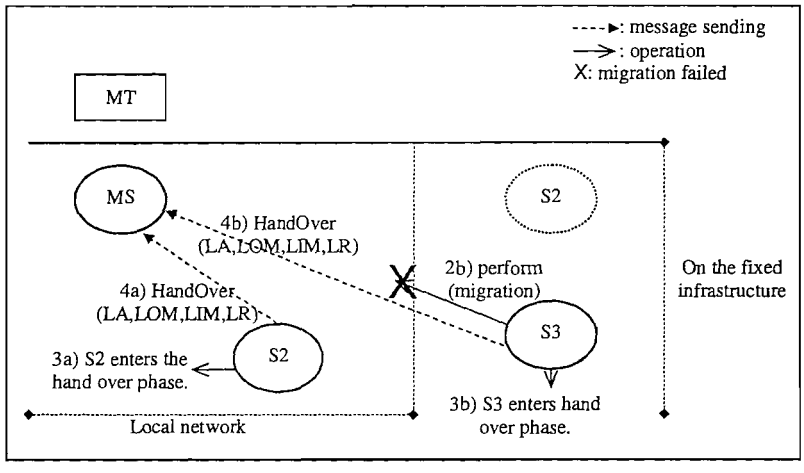


Figure 4.26: Shadow locating phase: failure case 1

4.3.3.4 Hand over phase

The hand over phase is a phase undertaken by the shadows of a mobile terminal for their reconciliation after a migration or when a new main shadow is assigned by the mobile terminal.

4.3.3.4.1 Hand over phase: perfect scenario

Once a shadow (S2) receives information of the main shadow (MS), S2 hands over its functions and lists of messages to MS. To do this, S2 sends a "HandOver" message, which contains a list of application mappings (LA), a list of outgoing messages (LOM), a list of incoming messages (LIM), and a list of requests (LR) (cf. Figure 4.27).

MS adds the received LA into a temporary list, LA^{new}, and the LOM, LIM and LR are extended to its local LOM, LIM and LR accordingly. To each application (App), listed in LA^{new}, MS sends a "NewMainShadowInformation" message, which informs App that it is now acting as the new intermediary between App and the mobile terminal (MT). The "NewMainShadowInformation" message contains S2's identifier for validation purposes in App.

When an application (App) receives a "NewMainShadowInformation" from MS, it first checks the enclosed shadow's information. If it contains the same information as its current shadow (S2), it proceeds by sending a "NewMainShadowInfo_ack(ID^{MS})" acknowledgement to S2. By doing this, App indicates that it has accepted MS as its new intermediary for interaction with the mobile terminal.

When S2 receives a "NewMainShadowInfo_ack(ID^{MS})" message from App, it first recognizes that MS has successfully acknowledged the hand over and second realizes that MS has taken over the interaction with App. Knowing the former, S2 sends an acknowledgement, "HandOver_ack", to MS and sets its hand over flag as true. The latter drives S2 to remove App's information from its list of application mappings (LA) and decreases its handOverCounter. The variable handOverCounter indicates the number of applications still not handed over to MS that rely on a shadow to be their intermediary. A handOverCounter that has been successfully decreased to zero and an empty list of application mappings (LA) show that there are no more applications relying on the shadow to be the intermediary. With this status the shadow is free from any task and may terminate by entering the shadow termination phase.

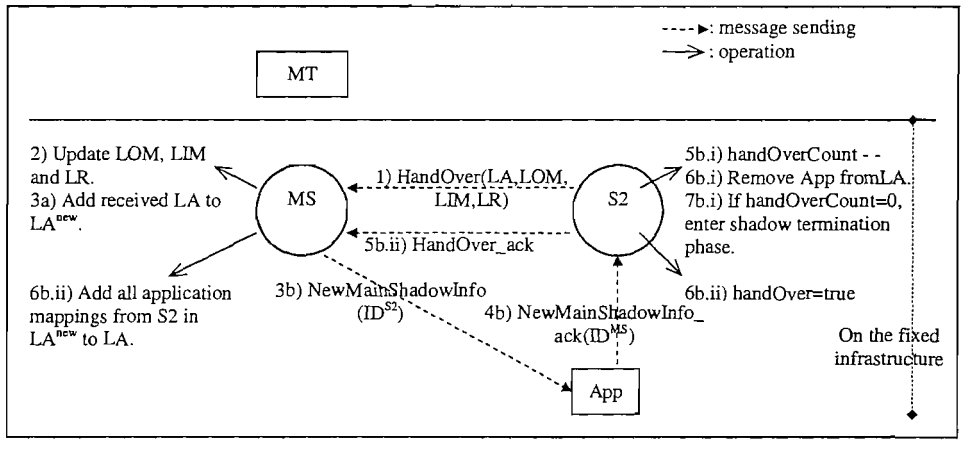


Figure 4.27: Hand over phase: perfect scenario 1

4.3.3.4.2 Hand over phase: failure cases

A shadow (S2) may fail to transfer its function to the main shadow (MS), in which situation it keeps sending a "HandOver" message on failure until MS eventually receives it (cf. Figure 4.28).

If S2 receives a request to migrate before it can transfer its functions to MS, S2 immediately stops its attempt to send the "HandOver" message to MS.

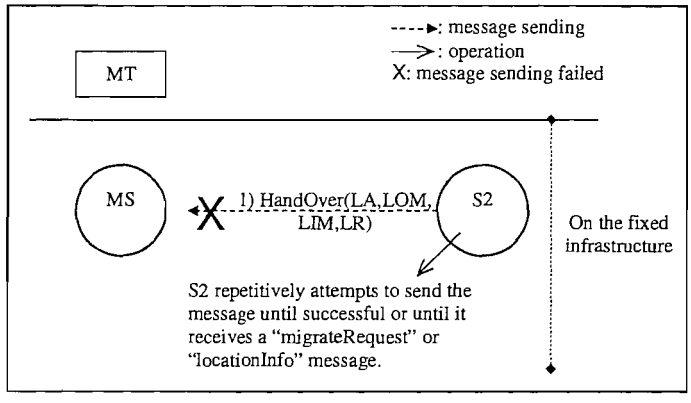


Figure 4.28: Hand over phase: failure case 1

One possible failure in a main shadow (MS) is not being able to send a "NewMainShadowInformation(ID^{S2})" to an application (App), which has recently been handed over by a shadow, S2 (cf. Figure 4.29). With this failure, App would not be aware that MS is its new intermediary. In this situation, MS would need S2 to be the router between itself and App. MS does this by sending a "BeTheRouterRequest(ID^{App})" message to S2 as a request. Once a link with App is established via S2, MS stores the routing information, which includes the fact that S2 is the router for App, in the "$LA^{useRouter}$" list.

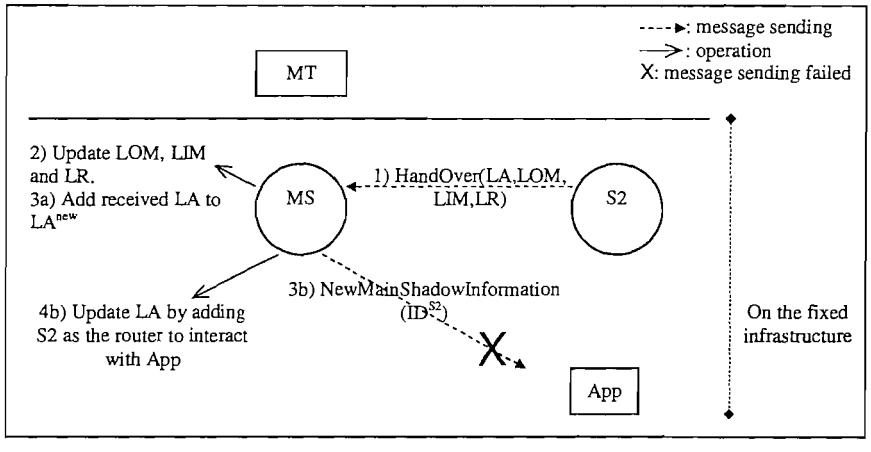


Figure 4.29: Hand over phase: failure case 2 (i)

As for S2, on receipt of the "BeTheRouterRequest(ID^{App})" message from MS, it first checks the availability of App's information in its LA. If App's information is found, S2 proceeds by adding App's information to another list called "LA^{router}", which is used by S2 to store information on applications for which it has to act as a router (cf. Figure 4.30). Once this is done, S2 may then receive a "SendMessageToApp(ID^{App},Msg)" message from MS, requesting it to relay a message to App, while in the reverse direction S2 may also receive a message from App, which will be relayed to MS. In this situation, S2 cannot terminate itself because it has to act as the router for MS to interact with App.

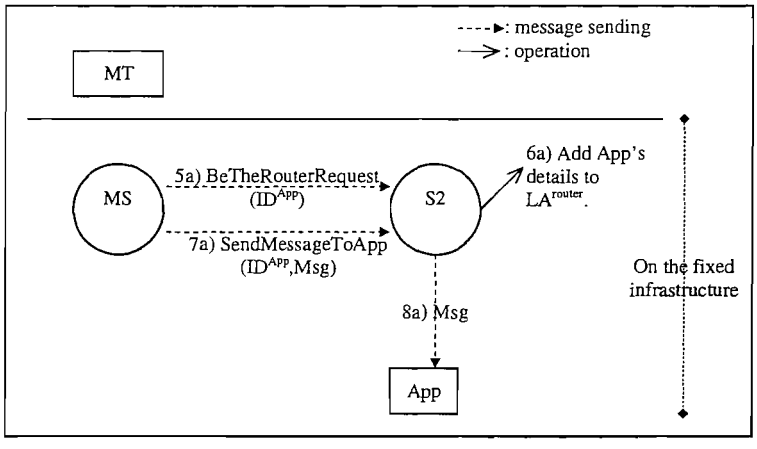


Figure 4.30: Hand over phase: failure case 2 (ii)

Like a regular shadow, a main shadow (MS) may also receive a "MigrateRequest" message or a "LocationInformation" message, which requests it to migrate to a new destination (cf. Figure 4.31). If MS decides to migrate, it removes all application information listed in $LA^{useRouter}$ because it would no longer be the applications' intermediary at the new location. This is done to eliminate complex interactions with applications that require a router. In such a situation, the applications involved would use their previous intermediary shadow to interact with the mobile terminal. Other temporary application lists, LA^{new} and LA^{router}, are also reset.

When a regular shadow (S2), which acts as the router between an application (App) and the main shadow (MS), has migrated to a new location, it is no longer the router between App and MS, but again becomes the intermediary between App and the mobile terminal.

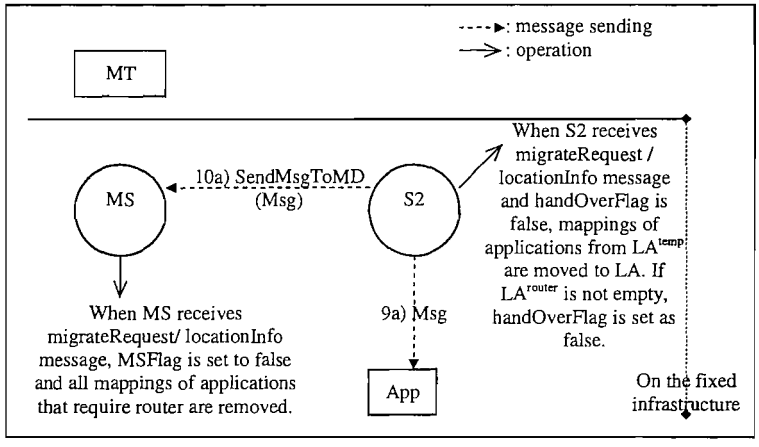


Figure 4.31: Hand over phase: failure case 2(iii)

In some cases, an application App may not be able to send a "NewMainShadowInfo _ack(ID^{MS})" message to its originating shadow, S2 (cf. Figure 4.32). With this type of failure, App would not be able to notify S2 that it has already accepted MS as its new intermediary with the mobile terminal. Unaware of this new information, S2 would always assume that application App is still relying on it to be the intermediary and therefore S2 would not remove the App's information from its LA, preventing its handOverCounter being decreased to zero and thus preventing it from proceeding to the shadow termination phase.

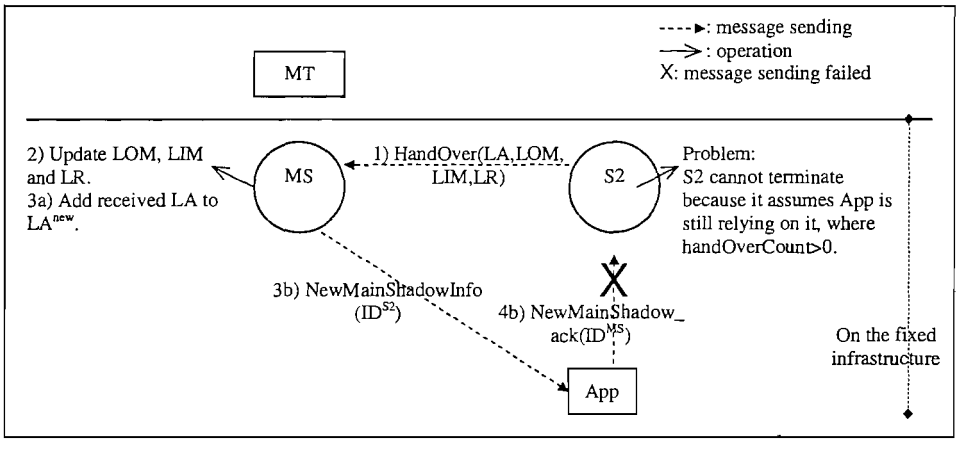


Figure 4.32: Handover phase: failure case 3 (i)

To prevent S2 from having such a problem, another way for S2 to get a notification from App about its new intermediary (MS) is through MS (cf. Figure 4.33). This means App may use MS to relay a message with the same information as carried by a "NewMainShadowInfo _ack(ID^{MS})"

message to S2. This is done by App by sending a "SendAckToShadow(ID^{S2})" message to MS. On receipt of this message, MS forwards a "SendApp_ack(ID^{App})" message to S2, after which S2 is aware that MS is now acting as the App's new intermediary and App is no longer relying on it. S2 is then able to remove App's information from its LA and decrease its handOverCounter.

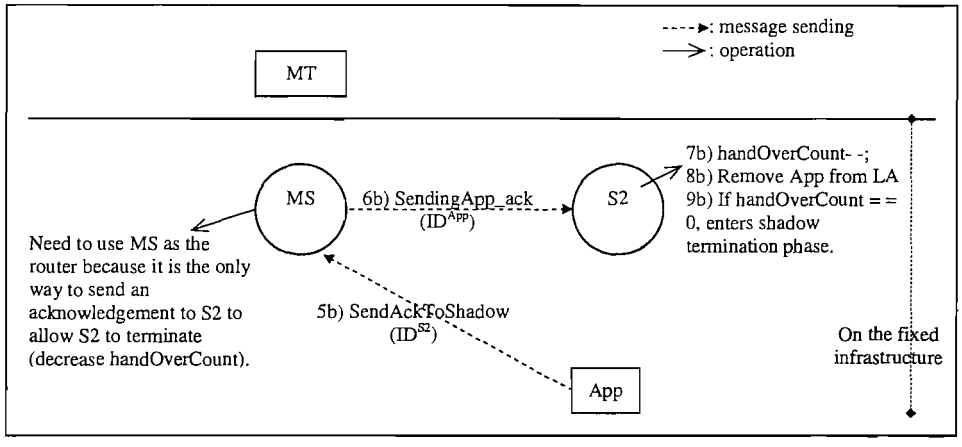


Figure 4.33: Hand over phase: failure case 3 (ii)

4.3.3.5 Shadow termination phase

A shadow (S2) enters the shadow termination phase when its handOverCounter is zero, its handOverFlag is true, and its list of application mappings LA is empty. In this phase, the shadow informs the main shadow about its intention to terminate, followed by the main shadow relaying the information to the mobile terminal.

4.3.3.5.1 Shadow termination phase: perfect scenario

Before its termination, S2 notifies the main shadow (MS) about its intention to terminate by sending a "Termination" message. On reception of the message, which contains S2's information, MS forwards this termination information to the mobile terminal (MT) in a "ShadowTermination" message and at the same time it returns an acknowledgement, "Termination_ack" to S2. After receiving such acknowledgement, S2 may terminate (cf. Figure 4.34).

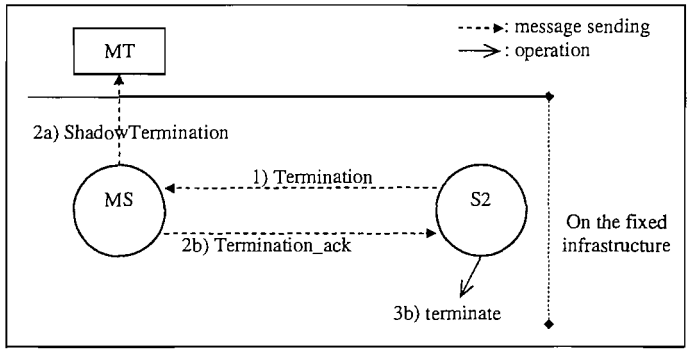


Figure 4.34: Shadow termination phase: perfect scenario 1

4.3.3.5.2 Shadow termination phase: failure cases

S2 may fail to send a "Termination" message to MS (cf. Figure 4.35), in which case S2 repeatedly tries to send the message until eventually the message is received by MS. But if S2 receives a "MigrateRequest" or a "LocationInformation" message, which requests it to migrate, S2 may migrate and continue the shadow termination phase at its new location.

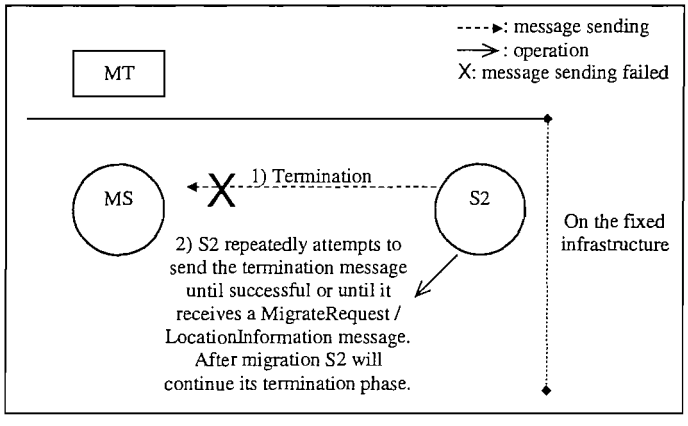


Figure 4.35: Shadow termination phase: failure case 1

Another possible failure that may happen in this phase is when a main shadow (MS) fails to inform the mobile terminal (MT) about the termination of a shadow (S2). This may be due to the disconnection of MT from the local network (cf. Figure 4.36). In this case, MS keeps trying to relay this information to MT until it is successful, even after MS's status has changed to a regular shadow at its new location. The failure of MS to inform MT about S2's termination always makes MT assume that S2 is still alive, which leads to a continuous effort to locate a terminated S2.

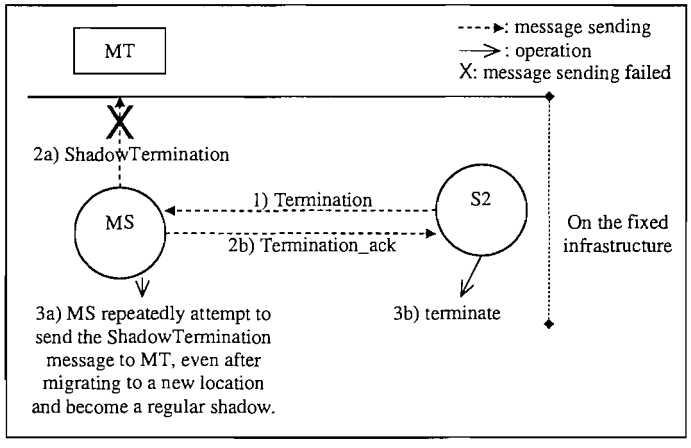


Figure 4.36: Shadow termination phase: failure case 2

After receiving a "Termination" message from S2, MS may not be able to return an acknowledgement message, "Termination_ack", to S2 (cf. Figure 4.37). With this failure, S2 is unable to terminate itself as it assumes that MS has not received its "Termination" message. Similarly to handling the failure to inform MT about S2's termination, MS keeps trying to relay this acknowledgement to S2 until it is successful, even if its status has changed to a regular shadow by this time. The permanent failure to relay the acknowledgement may leave S2 as garbage in the system, since S2 will not be able to terminate itself until it has received the acknowledgement message.

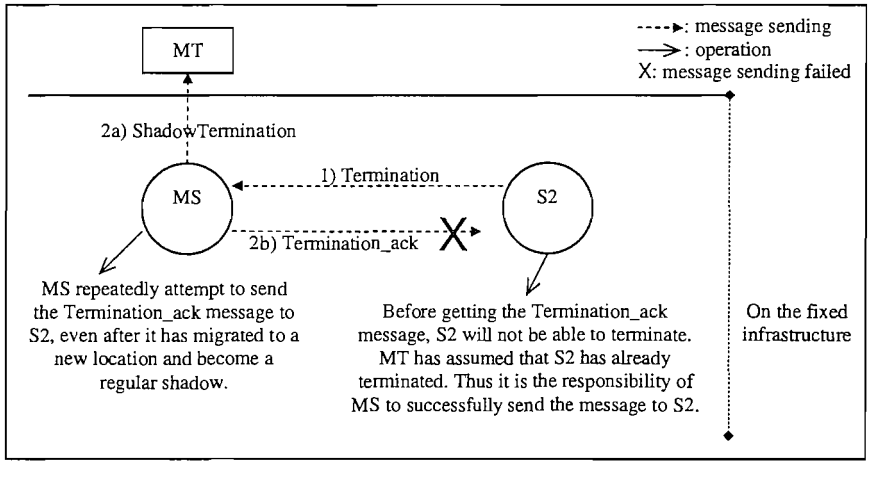


Figure 4.37: Shadow termination phase: failure case 3

4.3.3.6 Conclusion

In the interactions between MAMiMoU's components, we introduce a global property that allows the mobile terminal to interact only with a local main shadow, while regular shadows are allowed to interact directly with the mobile terminal only if it is connected locally. The most challenging task in specifying the interactions is to maintain consistency between the components, especially in the case of multiple shadows for a mobile terminal. Careful coordination was considered, especially in the hand over of tasks between shadows, as the failure to do so may lead to losing links to the applications on the fixed network. Although cases with failures were described together with their handlers, the handlers themselves may also fail. Thus, in some such cases, a permanent effort to handle the failure, such as a permanent attempt to send a message, is introduced in order to eventually reach a consistent state. In other cases, failures are handled with the best effort to correct the situation. An instance of this is the case of a main shadow that tries to have another shadow acting as a router to interact with an application, due to its failure to interact directly with the application. This effort may well fail, but would not create permanent damage and inconsistency.

4.4 Discussion

In this section we discuss the fundamental ideas behind the design of MAMiMoU's architecture. The main role of a shadow is to act as a proxy for a user on the network, performing tasks on the user's behalf especially when the user is disconnected. The use of a mobile agent as a shadow promotes flexibility in the sense that it is able to migrate and follow the user on the network. This way it can always be in the close vicinity of the user when the user is connected to the network. Such flexibility can help reduce the bandwidth required for the application to improve its performance [34]. The use of a mobile agent as a shadow also allows the shadow to communicate with the mobile terminal using specialized protocols, possibly dynamically chosen according either to the current location or to a negotiation between parties. Besides this, applications can communicate reliably using the transparent routing of messages to shadows.

One specific function provided by the shadow is to support asynchronous interactions between applications that are hosted on a mobile terminal and network-based applications. Such

asynchronous support is needed since synchronous interactions between these two types of applications are not practical due to the frequent disconnections experienced by mobile terminals. Such connectivity interruptions can cause mobile terminal's applications to fail to run. Thus, as an alternative, asynchronous interactions are more suitable for such an environment, where the shadow will store and forward messages between the applications. When the mobile terminal is disconnected, a network-based application can still send a message to the mobile terminal by forwarding the message to the shadow first. Once the mobile terminal reconnects, the message will be forwarded by the shadow to the mobile terminal. Vice versa, a message received from the mobile terminal can be forwarded by the shadow to the destined network-based application even though the mobile terminal is no longer connected. This way, we allow interactions to continue despite frequent disconnections being experienced by the mobile terminal.

From one perspective, MAMiMoU allows applications to be offloaded onto the fixed network; this is used to address the problems due to the limited capability of mobile terminals. This way, communication- and computation-intensive mobile terminal applications can be shifted from mobile terminals to the fixed infrastructure. On the one hand, fewer computation and communication activities left on the mobile terminal will result in less of the resources of the mobile terminal being used. On the other hand, the offloaded applications can take advantage of the resources in their new local environments, e.g. higher processing power and memory capacity. Once a user moves to a new location, the user will be attached to a new local network. More advantages can be gained if the mobile terminal is allowed to manipulate resources in its local vicinity; including the offloaded applications, instead of trying to manipulate remote resources. This is especially useful when the local network is not connected to the rest of the Internet.

In terms of shadow implementation, weak mobility is adopted, allowing the shadow to transfer only the code and state to the new location, as opposed to the strong notion of mobility, which requires that the code, the data state (i.e. the values of the internal variables) and the execution state (i.e. the stack and the program counter) be transferred [7]. The shadow can be implemented as a mobile agent with either strong or weak mobility, since the information necessary for operation is always stored in the mobile agent variables whose values are preserved during all movements. For example, a shadow stores incoming messages in LIM before forwarding them to the mobile terminal; to deal with such store-and-forward operations, weak mobility is enough. Besides, if the shadow is not permitted to migrate due to an unfinished critical operation, it is

allowed to delay migration or reject the migration request, in which case it will remain in its current location. Furthermore, failure handlers are introduced and explicitly defined in the coordination model to prevent inconsistency in the system and the loss of messages.

Referring to another aspect of mobility, [65] has described multi-hop migration as a scenario where an agent successively visits several network nodes while performing a given task. This description fits the behaviour of our shadow, which we believe is an important capability that allows the shadow to react based on its current environment.

4.5 Conclusion

In this chapter, we introduce MAMiMoU as an abstraction layer that allows mobile users to seamless roam the network while manipulating services on the fixed infrastructure. MAMiMoU is designed to overcome the limitations of the wireless environment and the limitations of mobile terminals, while hiding away the underlying communication complexities from the supported applications.

The main component of MAMiMoU is a mobile agent called shadow. In some cases, multiple shadows may exist, and coordination between them is described in a coordination model (cf. Section 4.3.3) that also explains other types of interactions that exist in the system. The coordination model is divided into five phases, namely migrate shadow, shadow arrival, shadow location, hand over and shadow termination phase. Based on this coordination model, a detailed protocol and algorithm is designed to coordinate interaction between the three components of the architecture, which will be presented in the next chapter.

Chapter 5

MAMiMoU Coordination Algorithm

In the previous chapter, we introduced MAMiMoU's components and a coordination model that outlines their interactions and activities. Based on this coordination model, we have designed a coordination algorithm that will be presented in detail in this chapter.

5.1 Overview

Our next contribution is a comprehensive coordination algorithm based on our study of how MAMiMoU's components should interact and perform their activities. The coordination algorithm is an innovative feature of MAMiMoU, consisting of the following:

- Interaction protocol between the middleware's components.
- Description of the shadow's functionalities that include i) supporting a store-and-forward mechanism, ii) maintaining existing associations between mobile terminal and network-based applications, iii) maintaining details of applications on the fixed network that have been created by the user.
- Shadow's migration protocol.
- Failure handler: On failure of a shadow to migrate, a new shadow is created dynamically for the user in order to provide an instant support to the user at his / her current location.
- Reconciliation of multiple shadows: In the scenario where multiple shadows exist, the shadows are able to merge their tasks together and hand these accumulated tasks to a single main shadow.

The coordination algorithm is presented in the form of pseudo-code, formalizing the interactions, activities and failure handlers of MAMiMoU's components, ready to be implemented. In particular, the algorithm is designed based on the phases previously defined in the coordination model (cf. Section 4.3.3) and the application and message management process, which will be described in Section 5.3. The algorithm is one of the main contributions of this thesis, also encompassing reconciliation between multiple shadows and the APIs that support MT-applications. The algorithm has been presented in two publications [133][134].

We will start with an introduction to the notations used in describing the algorithm, followed by a further description of MAMiMoU's components and their sub-components. In Section 5.3 we present the algorithm, followed by Section 5.4, which describes the implementation. Finally, we present a discussion and a conclusion.

5.1.1 Notations

Some of notations presented in Section 4.3.3 will still be used in the algorithm throughout this chapter. Additional notations such as the components' variables are shown in Figure 5.1.

Global Variables:

LAN_i :	Local network i
$P_{i,j}$:	Platform j on LAN_i
$SH_{i,j}$:	Shadow handler j on LAN_i
ID^{MT}, α^{MT} :	Mobile terminal: identifier, address
$ID^{MS}, \alpha^{MS}, \gamma^{MS}$:	Main shadow: identifier, address, counter
ID^{S}, α^{S} :	Shadow: identifier, address
$ID^{SH}_{i,j}, \alpha^{SH}_{i,j}$:	Shadow handler: identifier, address
$ID^{App}, \alpha^{App}, \varepsilon^{App}$:	Application: identifier, address, type

MT-agent's Variables:

mt_online	True if the mobile terminal is connected to the fixed network
new_location	True if the mobile terminal is connected to a different location from the previous one
LS :	List of active shadows
LOM:	List of outgoing messages
LIM:	List of incoming messages
LLT:	List of local lookup tables
SHInfoList:	List of local shadow handlers

Shadow's Variables:

LIM :	List of incoming messages
LOM^{App} :	List of outgoing application messages
LOM^{MT} :	List of outgoing messages for mobile terminal
LOM^{MS} :	List of outgoing messages for main shadow
LOM^{S} :	List of outgoing messages for shadow
LA :	List of applications
LA^{new} :	List of new applications received from other shadows
LA^{temp} :	Temporary list of applications
$LA^{useRouter}$:	List of applications that require router
LA^{router} :	List of applications that are using shadow as router

Message operations:

send(ID,α,Msg) :	Send message Msg to entity identified by ID,α
receive(ID,α,Msg) :	Receive message Msg from entity identified by ID,α

List operations:

[M]:L	List composed of a head M and tail L)
L_1:L_2	Concatenation of two lists L_1, L_2)
L:[M]	Message M added to the end of list L)
enqueue(M,L)	L:=L:[M]

Variable value:

⊥ : null

Figure 5.1: Notation

5.2 MAMiMoU's components and sub-components

Before we describe the coordination algorithm, we define the entities that are involved in the interactions. As explained earlier, one role of a shadow is to support asynchronous interactions between applications that are hosted on a mobile terminal and network-based applications (cf. Section 4.2). Thus, to simplify the explanation, we distinguish these two types of application by referring to the former as *MT-applications* and the latter as *N-applications* (cf. Figure 5.2). Other entities are MAMiMoU's components (cf. Section 4.3.1), which are built up of sub-components; each deals with a specific functionality (cf. Figure 5.1). This will be presented next.

Here we explain the system's behaviour in detail, where all the components are described further in terms of their states and operations, based on the coordination model presented in the previous chapter. The behaviour of a mobile terminal, shadow handler and shadow will be first explained in a high-level view, followed by the detailed explanation of each sub-component. Each component will also be explained by presenting its internal states. A component may have variables, which may act as triggers that activate new operations to be performed in the component. Basically, this has the same effect as when a component receives a message from another component.

The shadow is MAMiMoU's main component, and runs on the fixed network. To support interactions between a shadow and MT-applications, an application that we call *MT-agent* is hosted on the mobile terminal. We will first describe the MT-agent, followed by a description of the shadow-handler and shadow.

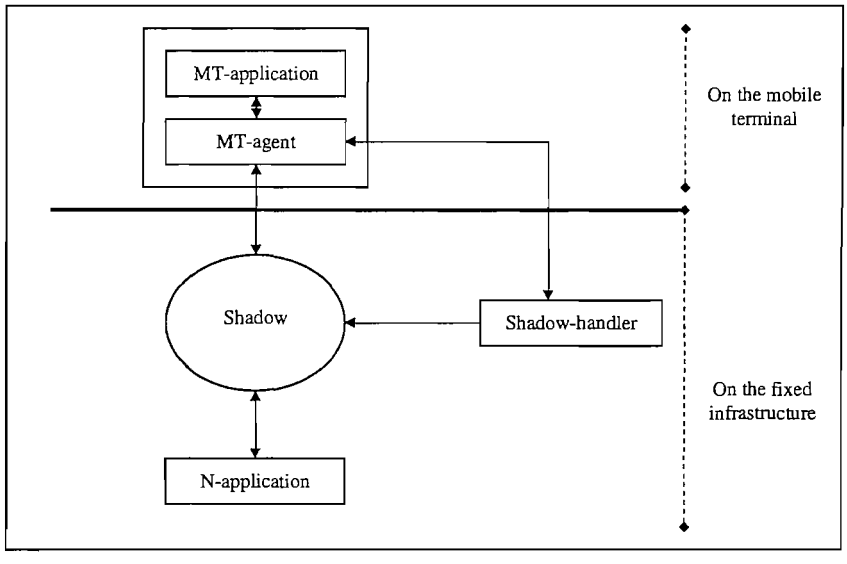


Figure 5.2: Interacting components

5.2.1 MT-agent

The MT-agent consists of four sub-components, namely application-interface, message-manager, application-manager and shadow-manager (cf. Figure 5.3). Each of these sub-components will be described next.

The MT-agent acts as an intermediary between the MT-applications and a shadow, receiving messages and requests from MT-applications and forwarding them to the shadow. In reverse, the MT-agent receives messages from the shadow, which will be forwarded to the respective MT-applications. We distinguish application messages from requests. Application messages are sent by MT-applications to N-applications or sent by N-applications to MT-applications. Requests are the type of messages that require the shadow to perform tasks. Both application messages and requests are queued by the MT-agent before they are forwarded to the shadow or MT-applications. For this purpose, the MT-agent has three types of variables, called i) list of outgoing application messages (LOM), ii) list of incoming application messages (LIM), and iii) list of requests (LR). We introduce two LOMs, specifically LOM^{App} to list application messages intended for N-applications, while LOM^{S} is used for coordination purposes, and queues messages specifically for shadows.

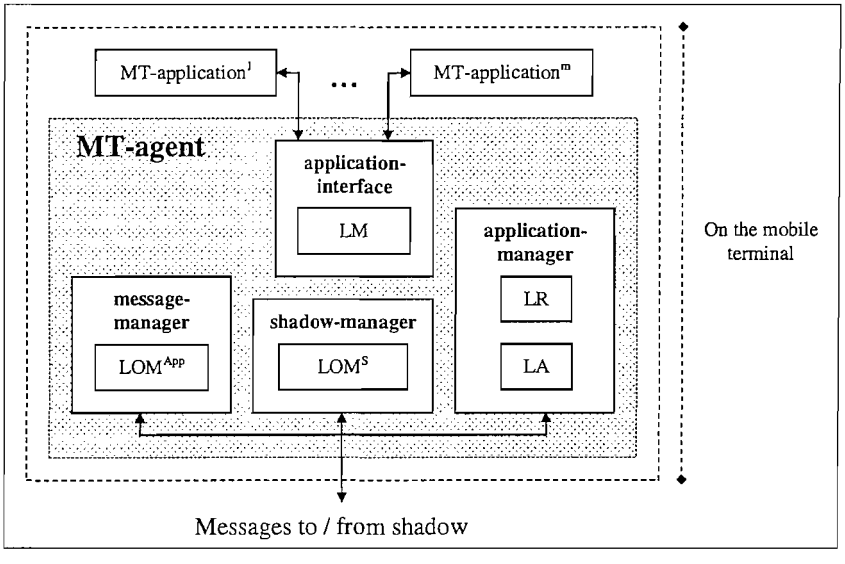


Figure 5.3: MT-agent's sub-components

5.2.1.1 LOM, LIM and LR

The variable LOM^{App} queues messages from MT-applications for N-applications, LIM queues messages from N-applications; via shadow, for MT-applications, and LR queues requests received from MT-applications. Having LOMs and LR to enqueue messages before forwarding them to the shadow is essential since the shadow will not be permanently available to the MT-

agent due to the intermittent connection of the mobile terminal. To perform the queuing of application messages and requests to their respective lists, the MT-agent has two sub-components called *message-manager* and *application-manager*. The message-manager queues messages to LOM^{App} and LIM, while the application-manager queues requests to LR.

5.2.1.2 Application-interface

Before we describe message-manager and application-manager, we now introduce the MT-agent's *application-interface*. The application-interface is an access point for MT-applications to services offered by MAMiMoU. It interacts with MT-applications by receiving messages or requests from them and returning acknowledgments to them. On receipt of requests from MT-applications, the application-interface forwards the requests to create or remove applications to the application-manager, while requests to get messages from N-applications are forwarded to the message manager. Messages from MT-applications for N-applications are forwarded to the message-manager.

Before forwarding these requests or messages to the message-manager or application-manager, the application-interface assigns an $ID^{message}$ to each one of them and includes these IDs in the requests or messages that are forwarded. $ID^{message}$ is used as a session identifier and included in a request or a message and later in the corresponding acknowledgement message. A session starts when a message is sent by an MT-application to the MT-agent, and ends when a reply or an acknowledgement message is returned to the MT-application. Besides this, the mappings between the $ID^{message}$s and IDs of the sending or requesting MT-application are recorded in a list of messages (LM). The details include the $ID^{message}$, the request or message itself and the identifier of the requesting MT-application. After a message or request is forwarded, the application-interface may in return receive from the message-manager or application-manager an acknowledgement message that contains the corresponding $ID^{message}$. On receipt of such an acknowledgement message from LM, the application-interface extracts the identifier of the MT-application to which the acknowledgement message should be returned. This is done by finding the $ID^{message}$ in LM that matches the $ID^{message}$ contained in the acknowledgement message. The application-interface then forwards the acknowledgement message to the respective MT-application. In the absence of an acknowledgement message, an MT-application may resend the same application message or request to the MT-agent.

5.2.1.3 Message-manager

On receiving messages from the application-interface, the message-manager queues them in LOM^{App}, while messages received from the shadows are queued in LIM. These messages are kept in the lists together with the IDs of their senders and receivers. On receipt of the request from an MT-application that is to get a message from a specific N-application, the message-manager first finds the matching sender's and receiver's IDs in LIM and returns the relevant message. Besides this, in parallel, the message-manager continually checks the status of LOM^{App}. If LOM^{App} is not empty, the message-manager attempts to send the first message in LOM^{App} to the shadow if the mobile terminal is online. As for incoming messages received from the shadow, the message-manager queues them in LIM.

5.2.1.4 Application-manager

The application-manager queues incoming requests to LR, while in parallel it also keeps details of the created N-applications. Activities involving LR will be described first, followed by activities involving another list called list of applications (LA). If LR is not empty and the mobile terminal is online, the application-manager sends the requests in LR to the shadow. In return the application manager receives acknowledgement messages from the shadow. On receipt of the acknowledgement messages from the shadow, the application-manager adds details of the newly created N-applications to LA, or removes details of a newly removed N-application from LA.

5.2.1.5 Shadow-manager

To keep track of shadows, the MT-agent has another sub-component called shadow-manager. The shadow-manager stores details of the existing shadows in a list called the list of shadows (LS). The details include the shadows' IDs and their addresses, which are used to locate the shadows. When a mobile terminal arrives at a new location, the shadows need to be notified of the new location of the mobile terminal to allow the shadows to migrate to the mobile terminal's vicinity. To know the current mobile terminal's connection and location, the shadow-manager depends on a callback for the mobile terminal's connection and location status, represented in two variables called *mt_online* and *new_location.* The variable mt_online is *true* if the mobile terminal is connected to the fixed network or false if otherwise, and the new_location variable has the value

true if the current location being visited by the mobile terminal is a new location. On arrival at a new location, the shadow-manager discovers a local shadow-handler on the fixed infrastructure, to which a request is sent in order to start the shadows' migration. The shadow-manager then detects shadows that have successfully migrated and updates their details in the LS. If no shadow was able to migrate, the shadow-manager requests the shadow-handler to create a new shadow, after which the new shadow's details are added to the LS. The shadow-manager's roles include assigning a new main shadow or informing regular shadows about a new main shadow. To do this the shadow manager needs to send messages to the respective shadows. Supporting this message sending is LOM^S, which is used to queue messages for shadows.

5.2.2 Shadow-handler

Once started, a shadow-handler advertises its presence through a local directory service. In terms of functionality, as reflected by its name, a shadow-handler's only function is to handle shadows on requests from mobile terminals. A shadow-handler may receive a request from a locally connected mobile terminal to migrate shadows. Such request includes LS that contains a list of shadows' details. To each of these shadows, the shadow-handler sends a message, requesting it to migrate to a local platform, specifically the one on which the shadow-handler is currently operating. A shadow is created by the shadow-handler when requested by a mobile terminal to start a shadow. The shadow-handler continually attempts to create a shadow on a creation failure.

5.2.3 Shadow

Like MT-agent, a shadow has an application-manager and a message-manager, which deal with requests to create or remove N-applications and messages exchanged between MT-applications and N-applications. To distinguish these components from the MT-agent's sub-components, we will refer to them as *s-message-manager* and *s-application-manager*. Besides these two components, a shadow also has a *migration-manager* to handle migration and also a *main-shadow component* and a *regular-shadow component* (cf. Figure 5.4). These two sub-components are used to coordinate activities according to the shadow's status, i.e. main shadow or regular shadow. These components will be further described next.

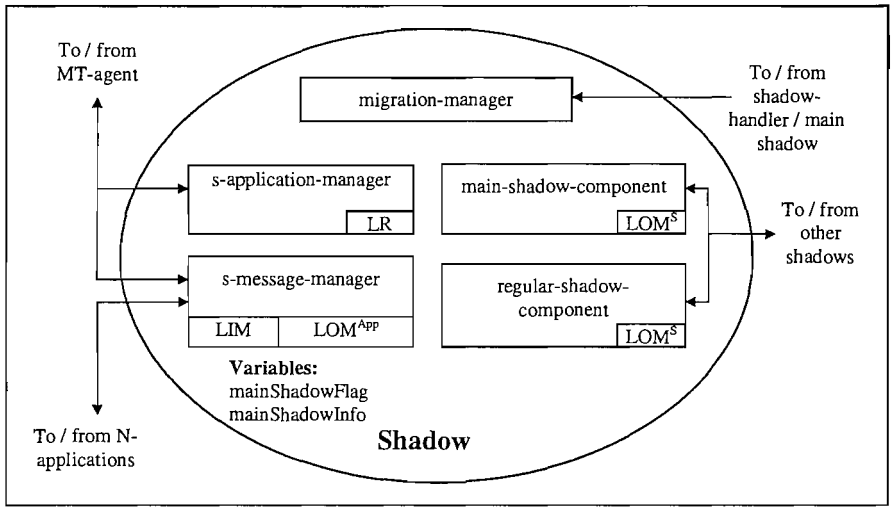


Figure 5.4: Shadow's sub-components

5.2.3.1 LOM, LIM, LR and LA

A shadow has some variables used to queue messages and requests and to keep details of existing N-applications. They are given the same names as given to MT-agent's variables. The list of outgoing messages (LOM) queues messages from MT-applications for N-applications, the list of incoming messages (LIM) queues messages from N-applications for MT-applications, the list of requests (LR) queues requests to create or remove N-applications, and finally the list of applications (LA) is used to keep details of the created N-applications. There are several LOMs and LAs used by a shadow; each will be described next. A shadow has four LOMs, namely LOM^{App}, LOM^{MT}, LOM^{S} and LOM^{MS}. The LOM^{App} is used to queue messages from MT-applications for N-applications, while the LOM^{MT} is used to queue coordination messages for MT-agent, e.g. a "ShadowInformation" message to notify MT-agent of the existence of a new shadow. The LOM^{S} and LOM^{MS} are both used to queue messages for shadows, where LOM^{S} queues message for any shadows, while LOM^{MS} queues messages for the latest main shadow, which may change over time. The LOMs are categorized as such since the shadow has a different round-robin processing mechanism for each LOM to deliver the queued messages. Besides the primary LA, there are four other LAs called LA^{new}, LA^{temp}, LA^{router} and $LA^{useRouter}$, which will be described in Section 5.3.3.4.

5.2.3.2 S-application-manager

The s-application-manager receives requests to create or remove N-applications from the MT-agent. On receipt of these messages, it queues them to LR. In parallel, the s-application-manager creates or removes N-applications according to the request extracted from LR. For a newly created N-application, its details are added to LA, whereas details of a removed N-application are removed from LA.

5.2.3.3 S-message-manager

The s-message-manager receives messages for N-applications from MT-applications through the MT-agent, which are queued in LOM^{App}. It also receives messages from N-applications for MT-applications, which are queued in LIM. In parallel, the s-message-manager also forwards messages extracted from LOM^{App} to the respective N-applications and forwards messages extracted from LIM to the MT-agent if the mobile terminal is online. The first message queued in the lists will be sent first.

5.2.3.4 Migration-manager

A shadow receives a request every time the mobile terminal relocates to a new location. This request contains information on the current location of the mobile terminal. By using this information, the shadow can migrate to the mobile terminal's vicinity. On receipt of a migration request from the shadow-handler, the shadow's migration-manager is responsible for starting the shadow's migration to the new location. In a shadow, there is a callback for intelligent decision making about migration; such a decision is not part of this algorithm, and may depend on the state of the application or prevailing network condition. The output of this decision-making process is obtained by the "callback" canMigrate(), which returns *true* if the application layer decides to migrate. Thus, the migration manager will start the shadow's migration only if canMigrate() returns *true*. The migration manager also has a variable called migrateCount that keeps track of the number of migrations of a Shadow. This variable is included in the messages queued to be sent to the MT-agent, preventing the shadow from sending outdated messages to the MT-agent. Outdated messages are also removed from the list. An example of an outdated message is a "ShadowArrival" message that should be delivered to the mobile terminal when it was at its

previous location. Such a message is not applicable when the mobile terminal has relocated to a new location, in which case it potentially has a new address. No harm will be caused by this outdated message, but it is no longer useful and will lead to a message delivery failure every time delivery is attempted, as the address of the mobile terminal is no longer valid.

5.2.3.5 Main-shadow and regular-shadow components

At any one time, a shadow can either be a main or a regular shadow and this status is reflected by a variable mainShadowFlag, value *true* denoting a main shadow and *false* if otherwise. Depending on this status the main-shadow and regular-shadow components become active or inactive accordingly, e.g. when mainShadowFlag is *true*, the main-shadow component is active and the regular-shadow component is inactive. A newly created shadow is initially defined as a regular shadow, in which case by default its mainShadowFlag is *false*.

5.3 Coordination Algorithm

Activities performed by each component and sub-components of MT-agent and the shadow were described in the previous section. In this section, inter-component activities involving interactions between components and occurring events are described. Activities are categorized into two types, namely processes and phases. A handler-process is a kind of daemon that is not invoked explicitly, but waits for some conditions to occur before performing a sequence of activities repetitively and continuously. A phase is a sequence of activities that are performed only occasionally. Process will be described first, followed by the explanation of phases.

Two processes are defined, specifically i) the message-handling process, and ii) the application-handling process, while the phases correspond to the five phases introduced earlier in the coordination model, namely i) migrate shadows phase, ii) shadow arrival phase, iii) shadow locating iv) hand over phase, and v) shadow termination phase. In describing the handler processes and phases, we want to be able to show them from three different perspectives, namely i) the time sequence of the components participating in the interaction; ii) the components' internal processing, i.e. action states; and iii) the components' locations. Since a standard UML sequence diagram and an activity diagram can be used to show the first and second perspectives,

we combine both diagrams into one. In addition, the diagram will also include the components' location indicators. This diagram is illustrated and described in Figure 5.5.

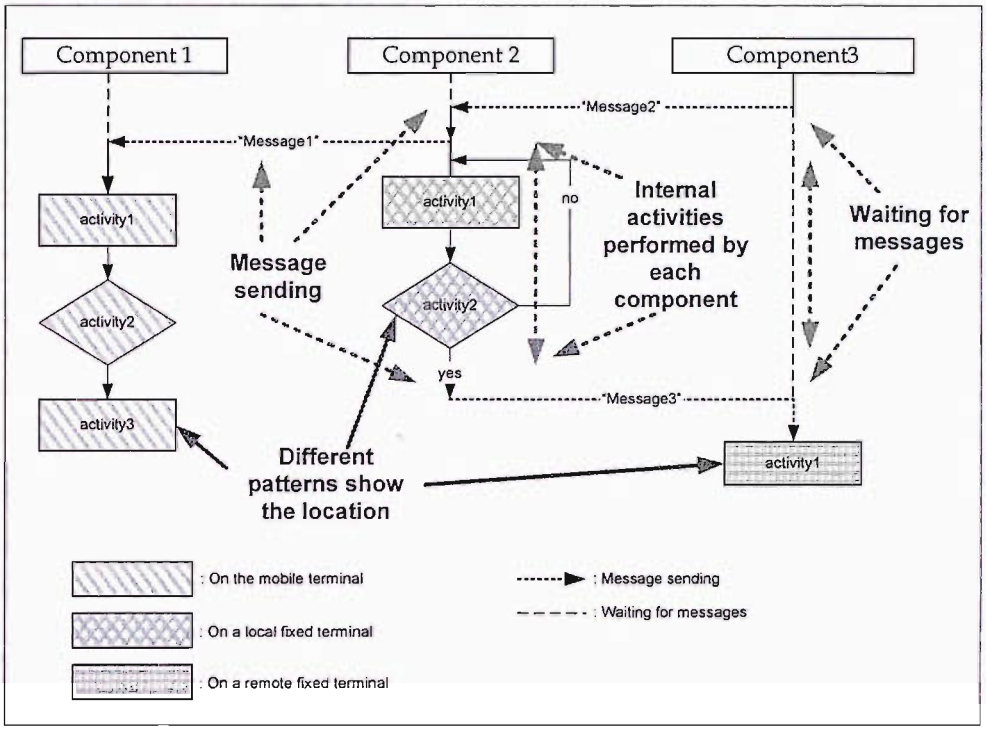


Figure 5.5: Combination of sequence and activity diagram

The sequence diagram is useful in showing the overall sequence of the activities and interactions between the components while the activity diagram allows us to view the activities internally performed by each component. In addition, the diagram is able to illustrate the different locations at which the components are performing the activities. To describe the phases further, we present the algorithm to be implemented by each component involved.

5.3.2 Processes

In this section we describe the application-handling and message-handling processes. The application-handling process describes the sequence of activities involved in creating and removing applications, while the message-handling process describes the sequence of activities involved in storing and forwarding messages between MT-applications and N-applications. To

present these processes, we give a lower level description of message passing between MT-agent and a shadow's sub-components, specifically the application-manager and message-manager.

5.3.2.1 Application-handling process

We illustrate the application-handling process in two figures. Figure 5.6 shows activities that are triggered when a "CreateApplication" or "RemoveApplication" API is invoked by an MT-application, while Figure 5.7 shows the continuous activities performed in this phase, which requires MT-application-manager to periodically check the status of LR. For simplicity, we show the invocation of the API by the application as a message being sent to the MT-application-interface.

On receipt of a "CreateApplication" or a "RemoveApplication", the MT-application-interface forwards it to MT-application-manager (cf. Figure 5.6). When the MT-application-manager receives such messages, it enqueues them in LR. In parallel, the MT-application-manager also runs another set of activities. It periodically checks the status of LR. If LR is not empty, the next message from LR is retrieved. Figure 5.7 shows the set of activities that are performed when the retrieved message from LR is a "CreateApplication" or "RemoveApplication" message. The variable createOrRemoveApp represents either one of these two types of messages. The MT-application-manager first checks the mt_online status. If the value of mt_online is *true* it proceeds by sending the retrieved "CreateApplication" or "RemoveApplication" message to the shadow, specifically to the s-application-manager. Otherwise, if mt_online's value is *false*, it waits until mt_online's value is *true*.

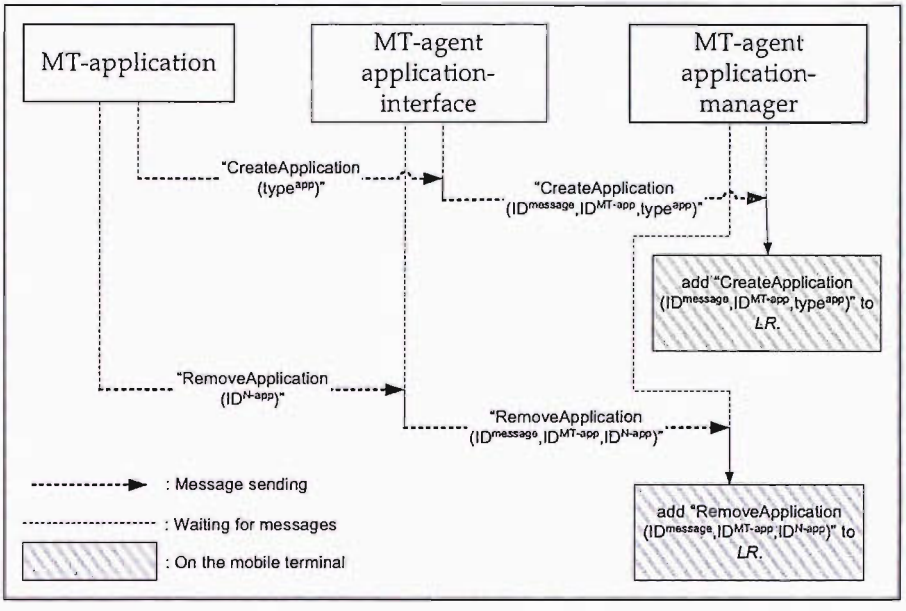


Figure 5.6: Handling requests from the MT-applications manager.

On receipt of a "CreateApplication(type^{app})" message, the s-application-manager creates an application of type type^{app} and assigns to it an identifier ID$^{N\text{-}app}$. Once this is done, s-application-manager adds details of the newly created application to LA. Then an acknowledgement message, "CreateApplication_ack", is returned to the MT-application-manager. This message contains the new N-application's ID (ID$^{N\text{-}app}$) and the creation status, i.e. "success" or "failed".

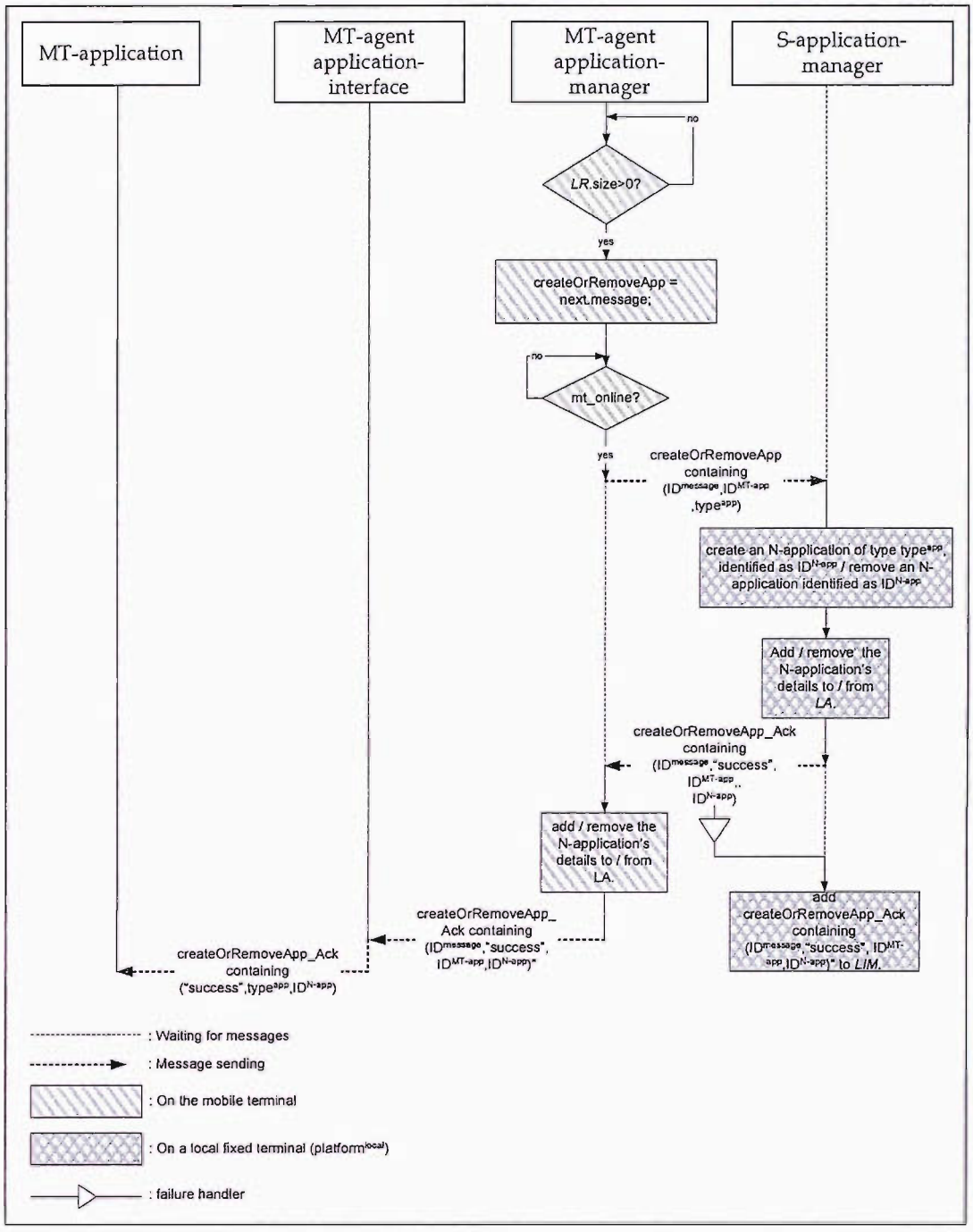


Figure 5.7: An N-application creation / removal

On receipt of a "RemoveApplication($ID^{N\text{-}app}$)" message, the s-application-manager removes the N-application identified as $ID^{N\text{-}app}$. Once this is done, s-application-manager then removes details

of the N-application from LA only if the identifier of the requesting MT-application is the same as that of the creator of the N-application. Then an acknowledgement message, "RemoveApplication_ack", is returned to the MT-application-manager. This message contains the N-application's ID ($ID^{N\text{-}app}$) and the removal status, i.e. "success" or "failed".

If the acknowledgement message fails to be delivered to the MT-application-manager, it is added to LIM, in which case the message will eventually be delivered to the MT-agent (cf. Section 5.3.3.6). On receipt of the acknowledgement message, the MT-application-manager respectively adds or removes details of the N-application from its LA. Then the MT-application-manager forwards this message to the MT-application-interface, which returns the acknowledgement message to the requesting MT-application.

5.3.2.2 Message handling process

Figure 5.8 illustrates the sequence of activities involved in handling outgoing messages, i.e. messages from an MT-application to an N-application, while Figure 5.9 illustrates the sequence of activities involved in handling incoming messages, i.e. messages from N-application to MT-application. These activities will be described next.

An MT-application can send a message to an N-application by sending a "SendMessageOut" message to the MT-application-interface, which forwards this message to the MT-message-manager. On receiving such a message, the MT-message-manager adds the message to LOM^{App}. In parallel, the MT-message-manager is running another sequence of activities, in which it periodically checks the status of LOM^{App}. If LOM^{App} is not empty, the next message from LOM^{App} is retrieved. The variable mt_online status is also checked, in which case if its value is *true*, the MT-message-manager proceeds by sending a retrieved "SendMessageOut" message to the S-message-manager. Otherwise if the mt_online's value is *false*, the MT-message-manager blocks until its value is *true*.

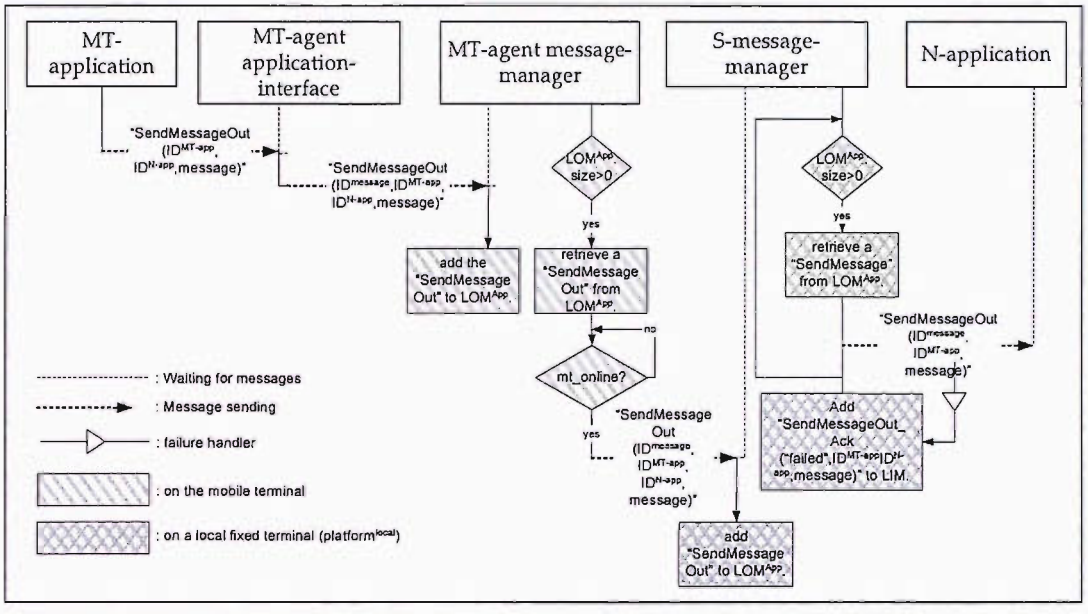


Figure 5.8: Handling outgoing messages

At the other end, on receipt of a "SendMessageOut" message from the MT-message-manager, the s-message-manager adds the message to its LOM. In parallel the S-message-manager also has another sequence of activities, which involves periodical checking of LOM's status. If LOM is not empty, the s-message-manager retrieves the next message from LOM. Once a "SendMessageOut($ID^{MT\text{-}app}$,$ID^{N\text{-}app}$,message)" message is retrieved from LOM it is then forwarded to the respective N-application, identified as $ID^{N\text{-}app}$. If the message fails to be sent, the s-message-manager creates an acknowledgement message "SendMessageOut_ack('failed',$ID^{MT\text{-}app}$,$ID^{N\text{-}app}$,message)", which is added to LIM. This message is delivered by the S-message-manager to the MT-agent.

Figure 5.9 shows how an MT-application identified as $ID^{MT\text{-}app}$ receives a message sent by an N-application identified as $ID^{N\text{-}app}$. First of all, the MT-application sends a "GetMessage($ID^{N\text{-}app}$)" request to the application-interface, which forwards the request to the MT-message-manager. On receipt of this request, the MT-message-manager retrieves the requested message from LIM and includes it in an acknowledgement message, "GetMessage_ack($ID^{N\text{-}app}$,$ID^{MT\text{-}app}$,message)". This acknowledgement message is then returned to the application-interface, which then forwards the message to the requesting MT-application. In some cases, the MT-message-manager may detect that the requested message does not exist, i.e. the message is not found in LIM. In such cases, a "GetMessage_ack ('failed',$ID^{MT\text{-}app}$,$ID^{N\text{-}app}$)" message is returned to the application-interface.

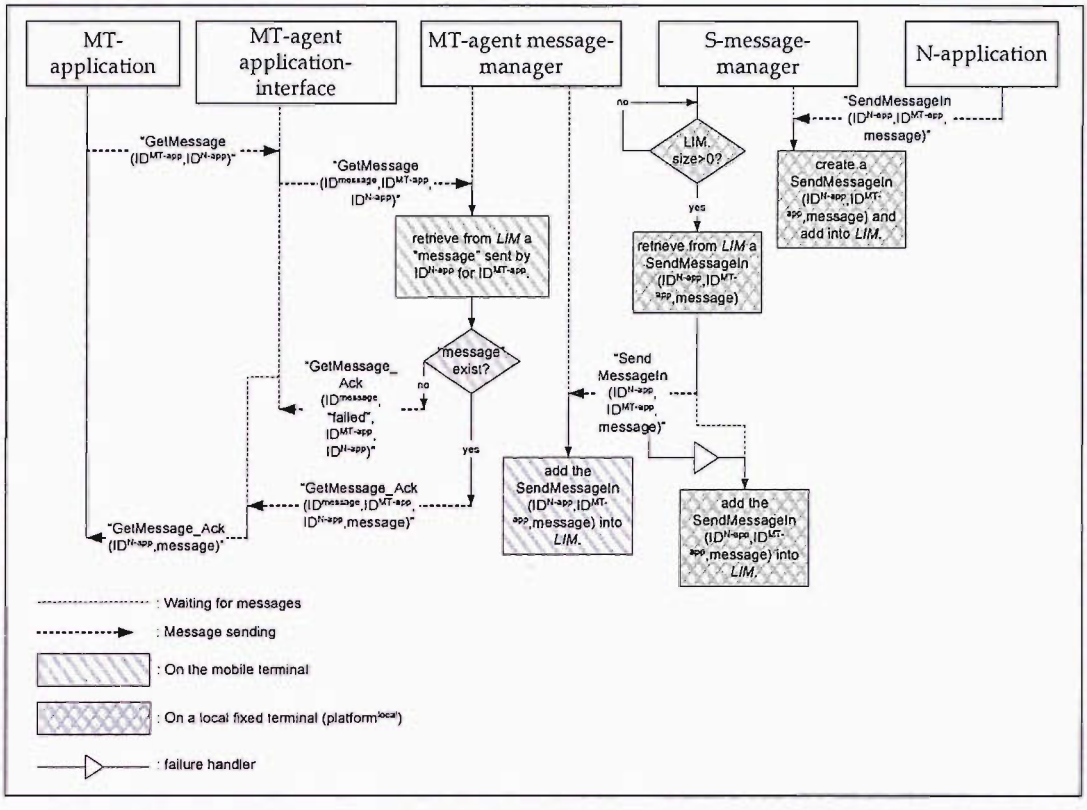


Figure 5.9: Handling incoming messages

Besides the activities described above, the MT-message-manager also operates another sequence of activities, where it continually listens to incoming messages from N-applications. To help the explanation, we will describe how an N-application with the identifier $ID^{N\text{-}app}$ sends a message to an MT-application with the identifier $ID^{MT\text{-}app}$. First of all the N-application sends a "SendMessageIn($ID^{N\text{-}app}$,$ID^{MT\text{-}app}$,message)" message to the S-message-manager, which adds the message to LIM. In parallel to the process of receiving messages from N-applications, the S-message-manager continually checks the status of LIM. If LIM is not empty, the S-message-manager retrieves the next message from LIM and forwards it to the MT-message-manager. The MT-message-manager then adds the received message to LIM. In some cases, the S-message-manager may fail to send the message to the MT-message-manager. The undelivered message is then added to LIM. This message will eventually get sent when the mobile terminal reconnects to the network.

5.3.3 Coordination phases

To describe the coordination phases, we present a summary of each phase in a sequence-activity diagram, followed by the presentation of the algorithm to be implemented by each component involved. The sequence diagram shows the general view of activities taking place in the components, while the algorithm specifically presents the activities carried out by each component.

5.3.3.1 Migrate Shadows Phase

The migrate shadow phase was previously introduced in Section 4.3.3.1. Figure 5.10 shows the events that are taking place in the migrate shadows phase, which is started when a mobile terminal connects to a local network. Figures 5.11 – 5.13 show the algorithm implemented by each component involved in this phase. The process of discovering a local shadow-handler is presented in Figure 5.11 (cf. "RequestingSH"), where firstly the MT-agent requests a list of locally available shadow-handlers. From this list, the MT-agent extracts a shadow-handler's details, to which the MT-agent attempts to send a "MigrateShadow" message. On failure, the MT-agent sends the same request to the next shadow-handler in the list.

In a creation of a new shadow, the shadow-handler sends to the shadow an "MTInformation" message that contains the details of MT-agent (cf. Figure 5.12). On receipt of this message, the shadow sends a "ShadowInformation" message to the MT-agent. By receiving this message, the MT-agent acknowledges the existence of the new shadow and adds the shadow's identifier (ID^s) and address (α^s) to the LS. Since there is no other shadow on the fixed network, on receipt of such a message, the MT-agent assigns the new shadow to be the main shadow by sending it a "MainShadowAssignment" message. On receipt of this message, the shadow updates its status by setting its mainShadowFlag as *true*, after which the shadow acts as the main shadow.

After a "MigrateShadow" message is sent to the shadow-handler, a "shadowArrival" timer is started by MT-agent (cf. Figure 5.11), which is used in the shadow arrival phase (cf. Section 5.3.3.2). On receipt of "MigrateShadow" request, the shadow-handler sends a "MigrateRequest" message to all shadows listed in LS that are remotely located, requesting them to migrate to a local platform (cf. Figure 5.12). On receipt of this message, a shadow decides to migrate

depending on the "callback" canMigrate(). Once a shadow has migrated to the local platform, it starts the shadow arrival phase (cf. Figure 5.13).

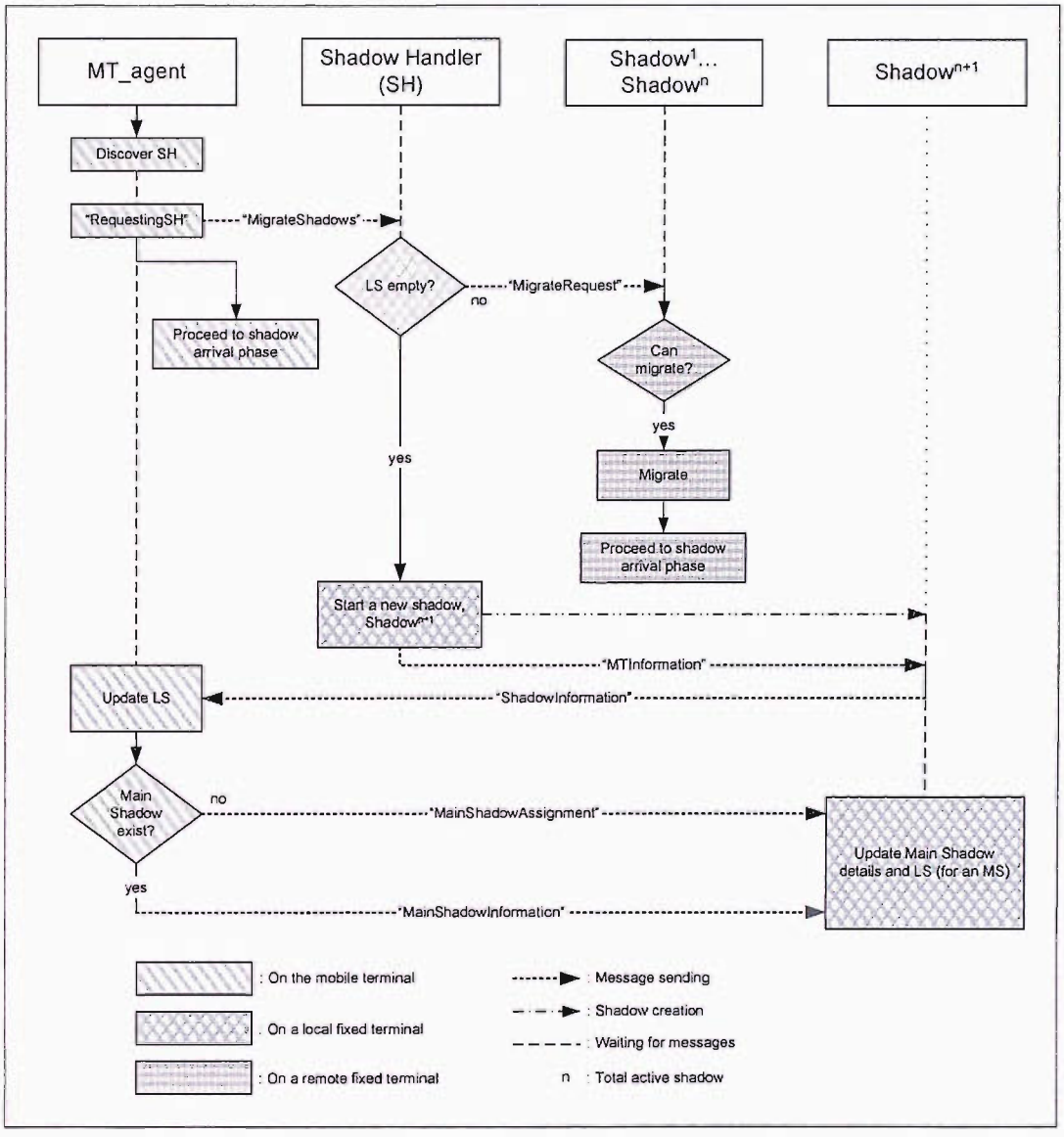


Figure 5.10: Migrate shadows phase (cf. Section 4.3.3.1)

```
MT-agent (implemented by the MT-shadow-manager)
Migrate shadows phase:

Variables: LLT, found^SH, SH_infoList, requestSent^SH, shadowArrival,
timer1Stopped, shadowArrived, ID^MS, α^MS, γ^MS;

LLT:=discoverLocalLookupTable();
found^SH:=false;

while !(found^SH):
      if LLT=[(ID^LT,α^LT)]:LLT', then:
            LLT:=LLT';
            SH_infoList:=query(ID^LT,α^LT, findLocalSH());
            if SH_infoList=⊥, then:
                  LLT:=LLT':[(ID^LT,α^LT)];
            else:
                  found^SH:=true;

RequestingSH:  //a label
      requestSent^SH:=false;
      while !(requestSent^SH):
            if SH_infoList=[(ID^SH,α^SH)]:SH_infoList', then:
                  SH_infoList=SH_infoList';
                  send(ID^SH,α^SH,MigrateShadow(LS));
                  if failed, then:
                        SH_infoList=SH_infoList':[(ID^SH,α^SH)];
                  else:
                        requestSent^SH:=true;

startTimer(shadowArrival, delay); // delay is the duration before timeout
timer1Stopped:=false; shadowArrived:=false;
//start shadow arrival phase (cf. Figure 5.15)

Wait for messages:
      if receive(ID^S,α^S,ShadowInformation(ID^MT)),
      then:
            if shadowArrived=false, then:
                  stopTimer(shadowArrival);
```

```
                timer1Stopped:=true;
                shadowArrived:=true;

        LS[ID^S]:=α^S;

        if ID^MS=⊥, then:
                ID^MS:= ID^S; α^MS:= α^S; γ^MS++;
                send((ID^MS, α^MS, MainShadowAssignment(LS, γ^MS)));

        else:
                enqueue((ID^S,
                α^S,MainShadowInformation(ID^MS,α^MS,γ^MS),LOM^S);
```

Figure 5.11: MT-agent: migrate shadows phase (cf. Section 4.3.3.1)

```
Shadow handler
Migrate shadow phase:

Variables: sentRequestCounter;
Repeatedly process the incoming messages:

if receive(ID^MT, α^MT, MigrateShadow(LS)), then:

        sentRequestCounter:=0;   // a counter
        for each pair (ID^S, α^S) in LS:
                send(ID^S, α^S, MigrateRequest(ID^MT, α^MT,α_i,k));
                if successful, then
                        sentRequestCounter ++;

        if sentRequestCounter = 0, then:
                startShadow(ID^MT, α^MT);


if receive(ID^MT, α^MT, startShadow), then:
        startShadow(ID^MT, α^MT);


Subroutine:

startShadow(ID^MT, α^MT):
```

```
start a shadow S with new identifier ID^S at address α^S;
send(ID^S, α^S, MTInformation(ID^MT,α^MT));
resend on failure;
```

Figure 5.12: Shadow handler: migrate shadow phase (cf. Section 4.3.3.1)

```
Shadow (implemented by the migration-manager)
Migrate shadows phase:

Variables: mainShadowFlag, ID^MS, α^MS, α^MT, justMigrated;

Wait for messages:
      if receive(ID^SH_u,q, α^SH_u,q, MigrateRequest(ID^MT,α^MT',α^P)), then:
            mainShadowFlag=false;
            ID^MS:=⊥;α^MS:=⊥;α^MT:= α^MT';
            if canMigrate(), then:
                  migrate(α^P);
                  //proceed to shadow arrival phase (cf. Figure
                  5.15)

Subroutine:

migrate(α^P_x,y):
      if (LAN_x≠LAN_i), then:
            migrate to α^P_x,y;
      migrateCount++;
      justMigrated:= true //for hand-over reset purpose (cf.
                          Figure 5.23)
     //start shadow arrival phase (cf. Figure 5.16)
```

Figure 5.13: Shadow: migrate shadow phase (cf. Section 4.3.3.1)

5.3.3.2 Shadow arrival phase

The shadow arrival phase was introduced in Section 4.3.3.2. Here it is summarized in Figure 5.14. Figures 5.15 – 5.16 present the algorithm implemented in the shadow and MT-agent for this phase. In this phase, the mobile terminal waits for "ShadowArrival" messages from its Shadows, after which it responds according to the algorithm presented in Figure 5.15. A "ShadowArrival" timer is used to detect the failure of shadows to migrate. On a timeout, a request is sent to the

shadow-handler to start a new shadow. Another timer called "ShadowArrival2" is started when the mobile terminal fails to send a reply to the arriving shadow. Such failure may be due to the disconnection of the local platform that hosts the shadows. Thus, ShadowArrival2's timeout is likely to occur since no newly arriving shadow can send a "ShadowArrival" message to the MT-agent, forcing it to send a request to another local shadow-handler to migrate its Shadows. This is illustrated in the handle_shadowArrival_II() subroutine invoked by the MT-agent (cf. Figure 5.15).

The shadow's algorithm for this phase is presented in Figure 5.16. It shows how a shadow sends a "ShadowArrival" message to the MT-agent on its arrival at a new location. For verification purposes, the "ShadowArrival" message that is queued for delivery to the MT-agent contains the migrateCount value. This value allows the shadow to detect whether the message, which was queued in LOM^{MT} is still valid before the message is delivered. A message is considered outdated if the migrateCount contained in the message is less than the current migrateCount of the shadow. For example, a "ShadowArrival" message, which is used to inform the mobile terminal about the shadow's arrival on a platform is no longer valid if the shadow has already migrated to another platform. In this case, the migrateCount variable recorded in the "ShadowArrival" message would be less than the current migrateCount, thus showing that the message is outdated and invalid. As a result, such messages are not sent to the MT-agent and are discarded from the list.

The shadow updates its state once a reply is received from the MT-agent. If a "MainShadowAssignment" message is received, the shadow sets its mainShadowFlag as *true*, which activates its main-shadow-component. As seen in the algorithm (cf. Figure 5.11), the main-shadow counter ($\gamma^{MS'}$), which is included in the message, is first compared against the latest main-shadow counter (γ^{MS}) known by the shadow. A $\gamma^{MS'}$ that is less than γ^{MS} indicates that the message is coming from the previous shadow and thus considered outdated. As a result this message is ignored. Otherwise if a "MainShadowInformation" message is received, the shadow updates its main-shadow's details, activates its regular-shadow-component and proceeds to the hand over phase (cf. Section 5.3.3.4).

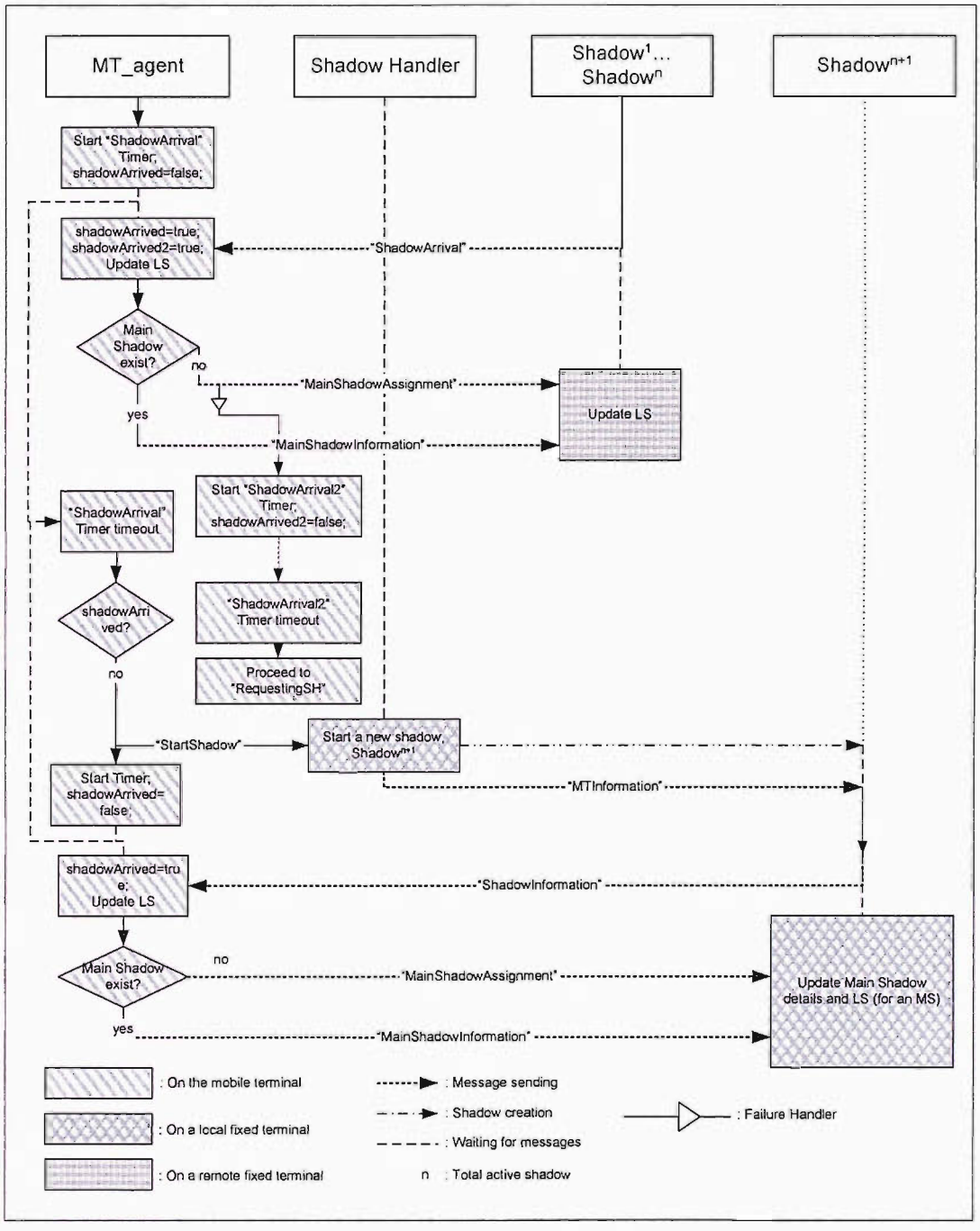


Figure 5.14: Shadow arrival phase (cf. Section 4.3.3.2)

```
MT-agent (implemented by MT-shadow-manager)
Shadow arrival phase:

Variables: timer1Stopped, shadowArrived, timer2Stopped, LS, ID^MS, α^MS, γ^MS,
shadowArrival, shadowArrival2;

//continued from the migrate shadows phase (cf. Figure 5.11)
while !(timer1Stopped):
      if timer.shadowArrival^timeout, then:
            stopTimer(shadowArrival);
            send(ID^SH,α^SH,startShadow);
            startTimer(shadowArrival, delay);

Wait for messages:
      if receive(ID^S,α^S,ShadowArrival(ID^MT)), then:
            if shadowArrived=false, then:
                  stopTimer(shadowArrival);
                  timer1Stopped:=true;
                  shadowArrived:=true;
            if (shadowArrived=true ^ timer2Stopped=false), then:
                  timer2Stopped:=true;

            LS[ID^S]:=α^S;
            if ID^MS=⊥, then:
                  ID^MS:= ID^S; α^MS:= α^S; γ^MS++;
                  send((ID^MS, α^MS, MainShadowAssignment(LS, γ^MS));
                  if failed, then:
                        handle_shadowArrival_II();

            if ID^MS≠⊥, then:
                  enqueue((ID^S,
                  α^S,MainShadowInformation(ID^MS,α^MS,γ^MS),LOM^S);

Subroutine:

handle_shadowArrival_II():
      ID^MS:= ⊥; α^MS:= ⊥; γ^Ms-−;
      startTimer(shadowArrival2,delay);
```

```
timer2Stopped:=false;
while !(timer2Stopped):
      if timer.shadowArrival2^timeout, then:
            stopTimer(shadowArrival2);
            //proceed to RequestingSH; (cf. Figure 5.11)
```

Figure 5.15: MT-agent: shadow arrival phase (cf. Section 4.3.3.2)

```
Shadow (implemented by the s-migration-manager)
Shadow arrival phase:

Variables: ID^MS, α^MS, mainShadowFlag, γ^MS;

enqueue(ShadowArrival(ID^MT,migrateCount),LOM^MT);
ID^MS:=⊥;α^MS:=⊥; mainShadowFlag:=false;

Wait for messages:

      if receive(ID^MT, α^MT,MainShadowAssignment(LS, γ^MS')),
      then:
            if γ^MS'>γ^MS, then:
                  mainShadowFlag:=true;
                  γ^MS := γ^MS';
                  //start the shadow locating phase (cf. Figure 5.17)

      if receive(ID^MT, α^MT,MainShadowInformation(ID^MS',α^MS',γ^MS')),
      then:
            if γ^MS'>γ^MS, then:
                  mainShadowFlag:=false;
                  ID^MS := ID^MS'; α^MS := α^MS'; γ^MS := γ^MS';
                  //start the hand-over phase (cf. Figure 5.21)
```

Figure 5.16: Shadow: shadow arrival phase (cf. Section 4.3.3.2)

5.3.3.3 Shadow locating phase

The shadow locating phase was introduced in Section 4.3.3.3. It is summarized in Figure 5.17, followed by the algorithm of the main shadow and regular shadow in Figure 5.18 and Figure

5.19. In this phase, two types of components are involved, namely a main-shadow and a regular-shadow (Figure 5.17). The main shadow sends a "LocationInformation" message to each shadow listed in the LS; in the event of a failure, the message is resent (cf. Figure 5.18). For verification purpose, the "LocationInformation" message contains the main-shadow counter (γ^{MS}). The latest main-shadow being assigned by MT-agent would have the highest γ^{MS}. This way the receiving regular-shadow can detect outdated messages from shadows that have ceased to be the main shadow. On reception of a "LocationInformation" message, a regular-shadow first verifies that the contained γ^{MS} in the message is not less than the latest recorded γ^{MS} (cf. Figure 5.19). Next, the shadow updates its main shadow's details including the γ^{MS}.

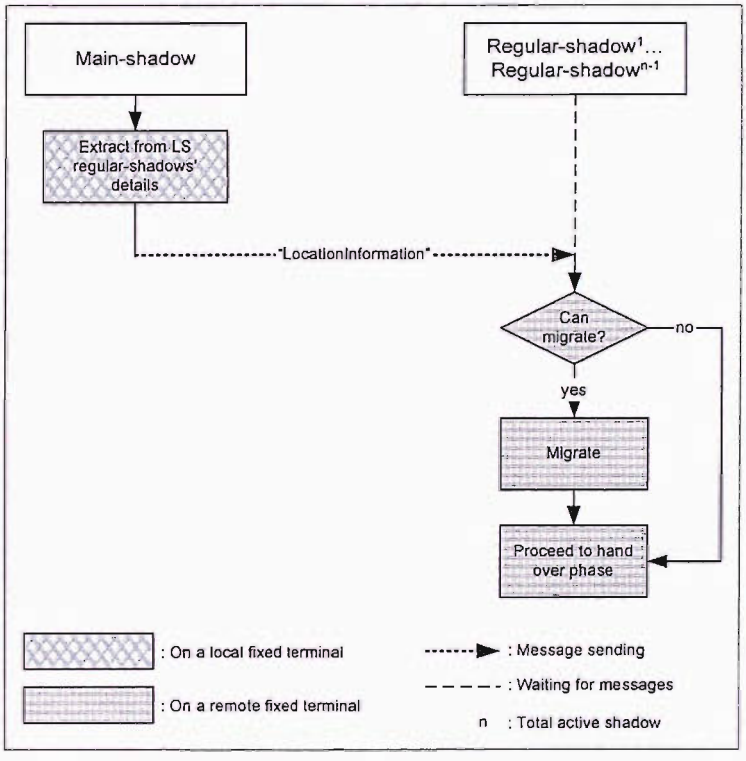


Figure 5.17: Shadow locating phase (cf. Section 4.3.3.3)

```
Shadow (implemented by the main-shadow-component)
Shadow locating phase (main shadow):

Variables: message^sent;

In parallel, for each (ID^S, α^S) in LS, do:
```

```
message^sent[ID^S]:=false;
while !(message^sent[ID^S]):
        send(ID^S, α^S, LocationInformation(ID^MT, α^MT, α^P, γ^MS));
        if success, then:
                message^sent[ID^S]:=true;
```

Figure 5.18: Main shadow: shadow locating phase (cf. Section 4.3.3.3)

```
Shadow (implemented by the s-migration-manager)
Shadow locating phase (regular shadow):

Variables: α^MT, ID^MS, α^MS, γ^MS, mainShadowFlag;

Wait for messages:

        if receive(ID^S, α^S, LocationInformation(ID^MT, α^MT', α^P, γ^MS')) ^
        (γ^MS' >= γ^MS), then:
                α^MT := α^MT'; ID^MS := ID^S; α^MS := α^S; γ^MS := γ^MS';
                mainShadowFlag:=false;
                if canMigrate(), then:
                        migrate(α^P); //(cf. Figure 5.13)
                else:
                        proceed to hand-over phase; //(cf. Figure 5.21)
```

Figure 5.19: Regular shadow: shadow locating phase (cf. Section 4.3.3.3)

5.3.3.4 Hand over phase

The hand over phase was introduced in Section 4.3.3.4. The overview of this phase is illustrated in Figure 5.20, while Figure 5.21 – 5.23 show the algorithm implemented by the regular shadow and main shadow in this phase. In this phase, the main shadow may receive a "HandOver_ack" message from the regular-shadow, indicating that the regular-shadow has acknowledged that all of its tasks have been handed over to the main shadow. On receipt of this acknowledgement message, the main shadow can now extend the LA' to its LA (cf. Figure 5.22).

From a regular-shadow's perspective, the hand over phase is started by initialising its handOverCounter value to the number of items stored in LA. Then the shadow sends a "HandOver" message to the main shadow. When a shadow receives a "NewMainShadowInfo_ack" message from an N-application, the shadow adds the N-application's information to a temporary list of application mappings, LA^{temp}, for backup purposes. If a failure occurs during the hand over process, the application mappings in LA^{temp} can always be added back to LA and the hand over process can be started again. Basically, this prevents the system from losing any of the application mappings. If no failure is detected, the application mappings in LA^{temp} are removed when the shadow migrates or when it enters the termination phase.

In some failure cases, an N-application is unable to send the "NewMainShadowInfo_ack" message to the creating shadow. In this case, the N-application requests the main shadow to send the acknowledgement on its behalf to the shadow. In this case, the shadow would receive a "SendingAppAck" message from the main shadow. On receipt of this message, the shadow would perform the same actions as when it receives a "NewMainShadowInfo_ack" message. A shadow may also receive a "BeTheRouterRequest" message from the main shadow. The shadow then adds the N-application's details to its list of routed applications, LA^{Router}. From this list, the shadow is able to know for which applications it has to act as a router.

For the purpose of resetting the hand over variables after a migration, both main-shadow and regular-shadow continuously check the justMigrated value. The justMigrated variable with a value *true* indicates that the shadow has just arrived at a new location. In this case the shadow updates its LA^{new}, LA^{temp}, $LA^{useRouter}$ and LA^{router} as presented in Figure 5.23.

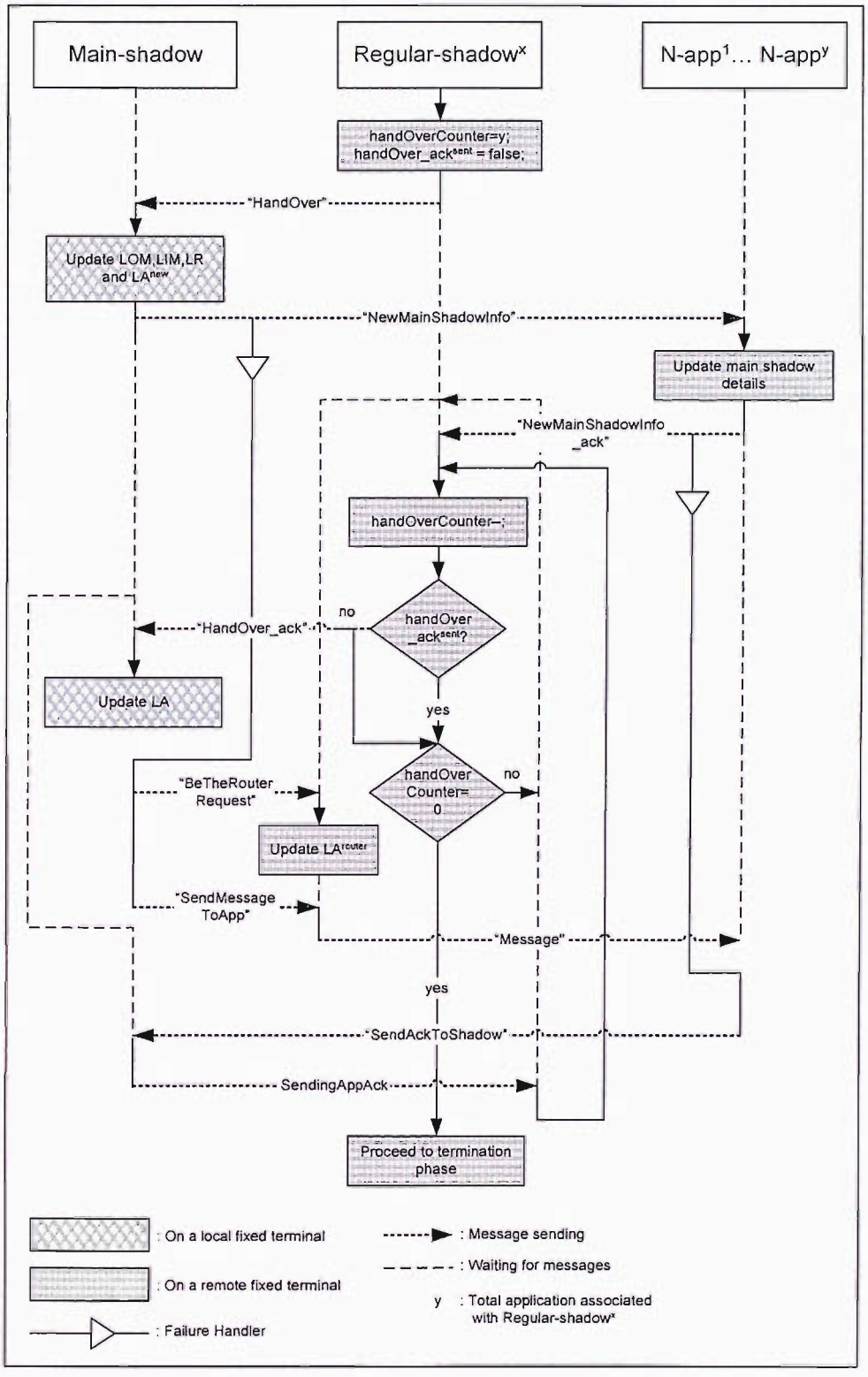


Figure 5.20: Hand-over phase (cf. Section 4.3.3.4)

Shadow (implemented by the regular-shadow-component)
Hand over phase:

```
Variables: handOverCount, handOverFlag, handover_ack^sent, LA, LA^temp, LA^router,
ID^MS, α^MS, γ^MS;

handOverCount:=length(LA); handOverFlag:=false;
enqueue(HandOver(ID^MT,LA,LOM^MT,LOM^App,LOM^S,LIM,LR),LOM^MS);

Wait for messages:

    if receive(ID^App, α^App, NewMainShadowInfo_ack(ID^MS) v receive(ID^MS, α^MS,
    SendingApp^ack(ID^App)) then:
        if !(handOverFlag), then:
            handOver_ack^sent:=false;
            while !(handOver_ack^sent):
                send(ID^MS,α^MS,HandOver_ack);
                if successful, then:
                    handOverFlag:=true;
                    handOver_ack^sent:=true;
        handOverCount--;
        if handOverFlag=true, then:
            LA[ID^App]:=⊥;
        else:
            LA^temp[ID^App]:=LA[ID^App]; LA[ID^App]:=⊥;

        if handOverCount=0, then:
            //proceed to shadow termination phase (cf. Figure 5.25)

    if receive(ID^MS, α^MS, BeTheRouterRequest(ID^App), then:
        LA^router[ID^App]:=LA[ID^App];

    if receive(ID^S, α^S,MainShadowInformation(ID^MS',α^MS',γ^MS')), then:
        if γ^MS'>γ^MS, then:
            ID^MS := ID^MS'; α^MS := α^MS'; γ^MS := γ^MS';
        if handOverFlag=false, then:
            //start the hand-over phase again;
```

Figure 5.21: Regular shadow: hand over phase (cf. Section 4.3.3.4)

Shadow (implemented by the main-shadow component)

Hand over phase:

Variables: $LS^{new}, LOM^{App}, LOM^{S}, LR, LA, LA^{new}$;

Wait for messages:

if receive(ID^{S}, α^{S}, HandOver(ID^{MT}, LA', $LOM^{App'}$, $LOM^{S'}$, LIM', LR')), then:

$LA^{new}[ID^{S}]$:= LA'; LIM:=LIM:LIM';

LOM^{App}:=LOM^{App}:$LOM^{App'}$; LOM^{S}:=LOM^{S}:$LOM^{S'}$;

LR:=LR:LR';

start InformApplication(ID^{S});

if receive(ID^{S}, α^{S}, HandOver_ack), then:

LA := LA : $LA^{new}[ID^{S}]$;

if receive(ID^{App}, α^{App}, SendAckToshadow(ID^{S})), then:

enqueue((ID^{S}, α^{S}, $SendingApp^{ack}$(ID^{App})), LOM^{S});

Subroutine:

InformApplication(ID^{S}):

In parallel, for each (ID^{App}, α^{App}) in $LA^{new}[ID^{S}]$, do:

send(ID^{App}, α^{App}, NewMainShadowInformation(ID^{S}));

if failed, then:

enqueue((ID^{S}, α^{S}, BeTheRouterRequest(ID^{App})), LOM^{S});

Figure 5.22: Main shadow: hand over phase (cf. Section 4.3.3.4)

```
Shadow (implemented by the regular-shadow-component)
Resetting hand over variables:

Variables: LA, LA^new, LA^useRouter, LA^temp, handOverFlag;

if (justMigrated), then:
      LA^new:=⊥;
      if LA^useRouter≠⊥, then:
            for each LA^useRouter[ID^App]≠⊥ in LA^useRouter, do:
                  LA[ID^App]:=⊥;
                  LA^useRouter[ID^App]:=⊥;

      if  !(handOverFlag) ^ (LA^temp≠⊥), then:
      //failures in the hand-over phase at previous location
            add LA^temp to LA;
            for each (ID^App,α^App) in LA^temp:
                  send(ID^App,α^App,NewMainShadowInformation(ID^myIdentifier));
                  resend on failure;
      else:
            if (LA^router≠⊥) ^ (handOverFlag=true), then:
                  handOverFlag:=false;

      LA^temp:= ⊥; LA^router:=⊥;
      justMigrated:=false;
```

Figure 5.23: Shadow: resetting hand over variables after a migration

5.3.3.5 Shadow termination phase

The shadow termination phase was introduced in Section 4.3.3.5. Here it is summarized in Figure 5.24, which shows the activities performed before a regular shadow terminates. Figures 5.25 – 5.27 show the algorithms of the components involved in this phase. In this phase, on receipt of the "ShadowTermination" message, the MT-agent removes the regular-shadow's details from the LS (cf. Figure 5.27).

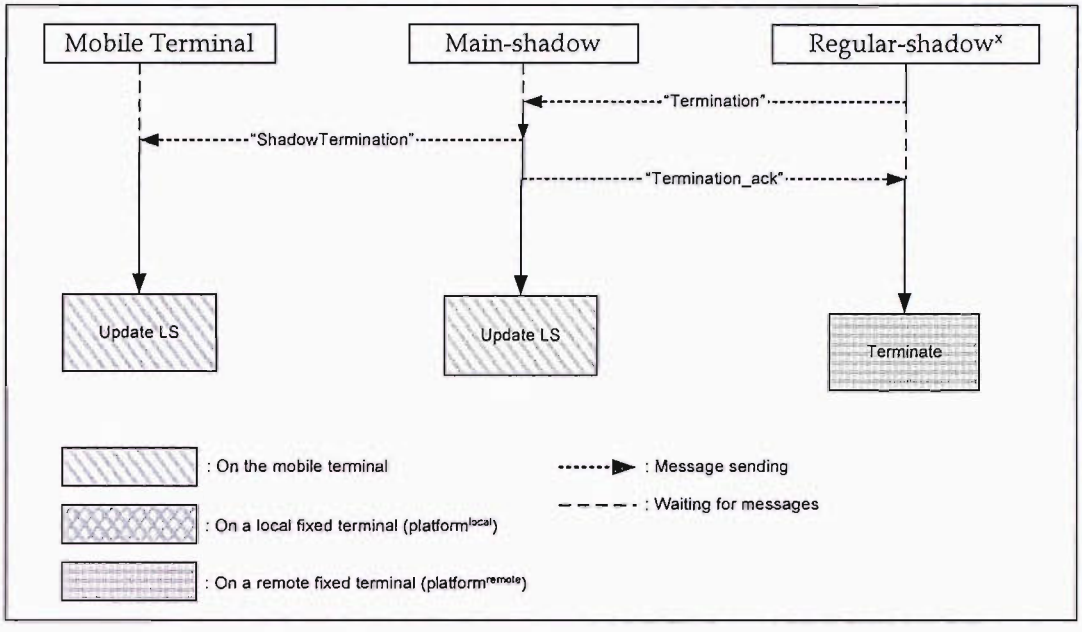


Figure 5.24: Shadow termination phase (cf. Section 4.3.3.5)

```
Shadow (implemented by the regular-shadow component)
Shadow termination phase:

enqueue(TerminationMessage(ID^MT),LOM^MS);
Wait for messages:
       if receive(ID^MS, α^MS, Termination_ack), then: terminate();
```

Figure 5.25: Regular shadow: shadow termination phase (cf. Section 4.3.3.5)

```
Shadow (implemented by the main-shadow component)
Shadow termination phase:

Variables: LS;

if receive(ID^S, α^S, TerminationMessage(ID^MT)), then:
       enqueue(ShadowTermination(ID^S),LOM^MT);
       enqueue((ID^S, α^S,Termination_ack),LOM^S);
       LS[ID^S]:= ⊥;
```

Figure 5.26: Main shadow: shadow termination phase (cf. Section 4.3.3.5)

```
MT-agent (implemented by the MT-shadow-manager)
Shadow termination phase:

Variables: LS;

Wait for messages:
       if receive(ID^MS, α^MS, ShadowTermination(ID^S)), then:
              LS[ID^S]:=⊥;
```

Figure 5.27: MT-agent: shadow termination phase (cf. Section 4.3.3.5)

5.3.3.6 Message delivery

In this additional section, we present the algorithm that describes the message delivery mechanism used by MT-agent and shadow in supporting interactions in the coordination phases. Figure 5.28 shows how MT-agent sends messages from LOM^{App} and LOM^{S} to the shadow. These messages are delivered only when a main shadow has been assigned at the new location. For each LOM, a round-robin processing mechanism is used to deliver the queued message in turn.

```
MT-agent (implemented by the MT-application-manager and MT-shadow-manager)
Outgoing message handler:

Variables: ID^MS, α^MS, γ^MS, LOM^S, LOM^App;

Continuously in parallel:

if (ID^MS≠⊥) ^ (α^MS≠⊥), then:
       if LOM^S=[(ID^S,α^S, MainShadowInformation(ID^MS',α^MS',γ^MS'))]:LOM^S', then:
              LOM^S:=LOM^S';
              if γ^MS'=γ^MS then:
                     sendOut(ID^S,LS[ID^S],MainShadowInformation(ID^MS,α^MS,γ^MS),LOM^S
                     ); // use ID^S's latest address

       if LOM^App=[Msg]:LOM^App', then:
              LOM^App:=LOM^App';
              sendOut(ID^MS,α^MS,Msg, LOM^App);
```

```
Subroutine:

sendOut(ID^R, α^R, Msg, Queue):
        send(ID^R, α^R, Msg);
        if failed, then:
                if Queue=LOM^APP, then:
                        enqueue(Msg, LOM^APP);
                else:
                        enqueue((ID^R, α^R, Msg), Queue);
```

Figure 5.28: MT-agent: outgoing message handler

In Figure 5.29 we show how a shadow delivers messages from its LIM, LOM^{MT}, LOM^{APP}, LOM^{MS} and LOM^{S} to the respective recipients. Messages from LIM can only be delivered to the MT-agent if the shadow is currently a main shadow. Otherwise the messages are forwarded to the current main shadow, each in a "SendMessageToMT" message. On receipt of this message, the main shadow enqueues it to its LIM.

The LOM^{MT} contains coordination messages, namely "ShadowInformation" and "ShadowArrival" to be sent to the MT-agent. These coordination messages can be outdated, i.e. when the shadow has migrated to another location. To prevent delivery of an outdated message, the shadow checks the migrateCount value included in the message before any delivery. If the value is less than the current migrateCount value recorded by the shadow, this indicates that the message is outdated and thus the message is then discarded. In the case of a new shadow trying to inform the MT-agent about its existence, the "ShadowInformation" message is repetitively sent to the MT-agent on failure for a predefined maximum number of times. After the maximum number of attempts due to failures the shadow can terminate itself since it has not been allocated any task and the MT-agent does not have any record that it exists.

LOM^{MS} and LOM^{S} are both lists that enqueue messages for shadows. The former contains messages for the current main shadow while the latter contains messages for any shadow. Messages from LOM^{S} are continually extracted and delivered to the respective shadows, while messages from LOM^{MS} are delivered to the current main shadow only if the sender itself is not a main shadow.

In delivering messages from LOM^{App}, there are two possibilities. Firstly, the message can be sent according to the application mapping provided by LA, or secondly the message has to be delivered via an intermediary or a router. In the latter case, details of the shadow acting as the router are first obtained from the $LA^{userRouter}$ and to this shadow, the message will be directed.

```
Shadow (implemented by all sub-components)
Message Handler:

Variables: LOM^MT, LOM^App, LIM, LOM^S, attemptCounter, maxAttempt, MTinformed,
mainShadowFlag;

Outgoing messages:

Continuously in parallel:

      if LIM=[Msg]:LIM', then:
            if mainShadowFlag=true, then:
                  send(ID^MT,α^MT,Msg);
                  if failed, then: enqueue(Msg,LIM);
            else:
                  send(ID^MS,α^MS,SendMessageToMT(Msg));
                  if failed, then: enqueue(Msg,LIM);

      if LOM^MT=[ShadowInformation(ID^MT,migrateCount')]:LOM^MT', then:
            if (migrateCount'= migrateCount)), then:
                  attemptCounter:=0;
                  MT^informed:=false;
                  while ((attemptCounter<maxAttempt) ^  !(MT^informed)):
                        send(ID^MT,α^MT,ShadowInformation(ID^MT));
                        if failed, then:
                              attemptCounter++;
                        else:
                              MT^informed:=true;

                  if attemptCounter>=maxAttempt, then:
                        //proceed to shadow termination phase (cf. Figure
                        5.24)

      if LOM^MT=[ShadowArrival(ID^MT,migrateCount')]:LOM^MT', then:
            if (migrateCount'= migrateCount)), then:
```

```
                    attemptCounter:=0;
                    MT^informed:=false;

                    while (attemptCounter<maxAttempt) ^ !(MT^informed)
                          send(ID^MT,α^MT,ShadowArrival(ID^MT));
                          if failed, then:
                                attemptCounter++;
                          else:
                                MT^informed:=true;

      if LOM^MS=[Msg]:LOM^MS', then:
            LOM^MS:=LOM^MS';
            if !(mainShadowFlag), then:
                  send(ID^MS,α^MS,Msg);
                  if failed, then: enqueue(Msg,LOM^MS);

      if LOM^S=[(ID^S,α^S,Msg)]:LOM^S', then:
            LOM^S:=LOM^S';
            send(ID^S,α^S, Msg);
            if failed, then: enqueue((ID^S,α^S,Msg),LOM^S);

      if LOM^APP=[(ID^APP,Msg)]:LOM^APP', then:
            LOM^APP:=LOM^APP';
            if (LA^useRouter[ID^APP]=(ID^S,α^S)), then:
                  send(ID^S,α^S, SendMessageToApp(ID^APP,Msg));
                  if failed, then: enqueue((ID^APP,Msg),LOM^APP);

            else:
                  send(ID^APP,LA^APP[ID^APP], Msg);
                  if failed, then: enqueue((ID^APP,Msg),LOM^APP);

Wait for messages:
      if receive(ID^S,α^S,SendMessageToMT(Msg)), then:
            enqueue((ID^S,α^S,Msg),LOM^MT);
```

Figure 5.29: Shadow: message handler

5.4 Implementation

We have developed MAMiMoU using the Southampton Framework for Agent Research (SoFAR) [72], which supports weak mobility. The algorithm of MAMiMoU is implemented by three agents, namely an MT-agent that is hosted on a mobile terminal, together with a shadow-handler and a shadow agent that are hosted on a stationary machine. MAMiMoU is currently applicable for high-capability mobile terminals such as laptops. In relation to shadow mobility, SoFAR provides "migrate()" and "restart()" APIs. The "migrate()" API allows the shadow to migrate to a new location, while "restart()" allows the shadow to start its operations at the new location. Although stationary on the hosting laptop, an MT-agent benefits from the physical mobility of its hosting environment.

5.5 Discussion

Our work focuses on the coordination of multiple shadows, and on the communication between mobile terminals and shadows. In our approach, we allow multiple shadows to be created, according to the prevailing network conditions, and by allowing shadows to make intelligent decisions as to whether to migrate. The hand over of the functions of a regular-shadow to the main shadow shortcuts chains of the regular-shadow's forwarding pointers when the regular-shadow does not migrate.

In MAMiMoU, the use of mobile agents and the transparent routing of messages to them [69][70] solved the problem of message delivery from the network-based applications to mobile terminals. By addressing the problem of reliable delivery and of the routing of messages in MAMiMoU, we have designed a generic solution reusable by other applications that need to support mobile users. Alternatively, without MAMiMoU support, a Mobile IP [86] solution may be used to route messages to the mobile terminal's mobile address, where it allows mobile terminal users to move from one network to another while maintaining their permanent IP address. Mobile IP adds to the Internet Protocol (IP) mechanisms for forwarding data to mobile terminals when they are connecting through a network other than their home network. But relying totally on this approach lacks the advantages offered by MAMiMoU, i.e. support on the network when the mobile terminal is disconnected.

A scalability issue may be raised in a situation where all the shadows of a user attempt to migrate to the same platform. One potential problem when this happens is that more resources are required by the shadows than when there are less shadows migrating, which may lead to a situation where the platform does not have sufficient resources to run an incoming shadow. The coordination algorithm avoids such problematic situations by having a user's regular shadows continuously attempt to hand over their tasks to a single main shadow, after which they can terminate themselves. Thus the number of shadows associated with a user is kept to the minimum, in which case the maximum number of shadows a user can realistically have is equivalent to the total number of different networks that have been visited by the user.

Another issue with regard to platform resources is a situation where the MAMiMoU message lists are becoming too long, consuming large amounts of the memory of a mobile terminal or a platform. To address such problems, a potential solution is to offload the messages into the file system or a database.

In some situations, the coordination algorithm requires a decision to be made, namely to set the value of a timer's delay before a timeout on the mobile terminal (cf. Section 5.3.3.1) and to set the maximum number of attempts for a shadow to deliver a particular message on failure (cf. Section 4.3.3.2). Such decision making does not matter for the correctness of the algorithm, but the right decision made for these values leads to better reactivity of the system. The MAMiMoU prototype is hosted on a fast connection network. In such a context, a delay of 1.5 times the average time needed for a shadow to migrate is sufficient to allow the mobile terminal to successfully acknowledge the shadow's arrival.

5.6 Conclusion

A mobile agent able to migrate around the network trying to stay as close as possible to the mobile user gives a major advantage by allowing local communication to be established with the mobile terminal. With this capability, the mobile agent is designed to be the main component in our middleware, allowing transparent interactions between fixed infrastructure applications and applications on mobile terminals. The main challenge in developing an application supporting

mobile users is to construct an intermediary layer that supports seamless communication between a traveling mobile user and a stationary application, e.g. a collaborative editing application hosted by the fixed infrastructure. In this chapter we have presented in detail the algorithm of MAMiMoU, which takes care of the coordination of multiple shadows, as well as the communication between a mobile terminal and its shadows. MAMiMoU hides the details of communication and coordination, allowing transparent interactions between fixed infrastructure applications and applications on a mobile terminal.

Having MAMiMoU support interactions between mobile terminals and network-based applications has made the implementation of the application straightforward, since the underlying communication between mobile terminal and network-based applications have been taken care of by MAMiMoU. The applications hosted on the fixed infrastructure are allowed to interact transparently with a mobile terminal through MAMiMoU, and we believe such ability is important to allow more applications for mobile users to be easily developed. Our next step is to complete a formal evaluation of the MAMiMoU's architecture, but first we will introduce an application for mobile users, which is designed to support the collaborative editing of a shared document between a group of mobile collaborators. This application will be described in detail in the next chapter and will later be used to evaluate MAMiMoU (cf. Chapter 7).

Chapter 6

Collaborative Editing Application

MAMiMoU was presented in Chapter 4 and its coordination algorithm was explained in depth in Chapter 5. MAMiMoU can be used to support mobile applications by hiding intermittent connectivity and users' mobility details. One of such mobile applications will be presented in this chapter. We now shift our focus to another problem in mobile users' environments, namely the lack of support for mobile document sharing and collaborative editing between mobile users. Our solution to address this problem will be presented in this chapter.

6.1 Introduction

In this chapter we present an application that enables collaboration between mobile users when they are geographically distributed and intermittently connected to the network. We designed the application based on the scenario we presented in Section 1.1. The application offers a mechanism to allow collaborative editing of mobile documents, which are documents hosted on the mobile terminals. It allows a group of mobile users to work on the same version of a document in parallel, thereby allowing them to save time. A major part of the application is a collaboration protocol that specifies the interactions between the mobile terminals and applications on the fixed infrastructure. This way, the collaborative activities between mobile collaborators and the edited documents are driven and tracked by the system. In relation to collaboration works described in CSCW field [19][24][25], the support that we present here focuses on a specific aspect of collaboration, namely one centered on the access and change of a shared document.

A challenging aspect of this application is to store and forward messages transparently to mobile users. In particular, we wish to be able to support both users who are temporarily disconnected from the Internet, and users who may be connected in an ad-hoc manner (possibly without Internet access). Hence, we envisage two modes of collaboration, namely local collaboration, involving locally connected users, i.e. on a LAN, and global collaboration, including users connected across the network. To that end, we are making use of MAMiMoU to hide communication and coordination details from mobile device and infrastructure applications (cf. Section 6.6).

The collaboration protocol incorporates a mobile document inconsistency handler, where changes made to a document's copies by users are distinguished and classified based on the users' different roles. It also promotes local and remote collaboration support. In a local collaboration, users connected to the local network have access to all shared mobile documents, while in a remote collaboration, a user can continue being involved in a collaboration started at a previous location once they move and reattach to a new location.

We start by introducing the application in the next section, followed by the description of the application's services. In Section 6.4 we present the application's architecture and in Section 6.5 we describe its collaboration protocol. The application's integration with MAMiMoU is presented in Section 6.6 followed by a discussion and a conclusion.

6.2 Application Overview

In today's world, many people are regularly on the move, traveling away from their homes or workplaces for several days or weeks. Wherever their location, they may join up and collaborate with locally connected users, while at the same time they may continue their collaboration with others on remote networks, such as their workplace's network. At any one time, other collaborators might also be on the move. Collaborative editing is a common form of collaboration, where a group of mobile users are working on a document together. In such collaboration, copies of a shared document are normally cached on mobile collaborators' terminals and are locally modified by them. Thus, the original document and its copies may become different and inconsistent with each other. By inconsistent, we refer to a state where the

document's copies are outdated when the original document has been modified by the owner, or the document's copies are in conflict with each other due to the modifications made by the users.

To address potential inconsistencies, we identify the original document hosted on a mobile terminal as a *master-document*, while the user editing such a document is known as the document's *master editor*. Other mobile users that have the document's copies cached on their mobile terminals are known as *regular editors*. We distinguish master editor from regular editor: the former is given full authorization to modify the master document, whereas the latter can only make tentative changes that need to be approved by the master editor. In this setting, the approved changes made by a regular editor will be merged with the latest version of the master-document on the master editor's terminal. To support interactions, e.g. in the form of notification delivery between master and regular editors, we provide a mechanism that allows regular editors to submit changes to the master editor and to receive asynchronous acknowledgement from the master editor. Distinguishing master editor from regular editor has some advantages: *i*) precise ownership helps to identify a good consistency model; and *ii*) document congruency, which we define as the state of a document's copies being similar to each other, can be maintained based on changes made by master and regular editors.

To further promote flexibility in the collaboration, we allow the master editor to be reassigned. The master editor may request that another collaborator becomes the next master editor. Such a request, once approved by the other collaborator, results in an irrevocable delegation of editing rights. At any one time, the system ensures that there is only one master editor per document. To implement this, we adopt the editing token metaphor, according to which a document is associated with an editing token, which is kept by the master editor. Every time the master editor changes, i.e. when the editing right is delegated to another user, the editing token will be passed to the next master editor.

Supported by the collaboration protocol, our approach can be summarized as follows:

- Documents are cached on users' mobile terminals to allow parallel and asynchronous editing. Having a locally cached document is essential for mobile terminals to permit user to manipulate the document while being disconnected from the fixed infrastructure.
- Copies of documents are stored by repositories on the infrastructure to allow parallel access to other users. A document hosted on a mobile terminal is rarely available to other users due to frequent disconnections experienced by the mobile terminal. This prevents

the document from being shared with other users. By having the document's copies hosted on the fixed infrastructure, other users have continuous access to the document.

- A mechanism exists that allows asynchronous interactions between master and regular editors. Since mobile users are frequently disconnected from the fixed infrastructure, a synchronous or real-time interaction between them is almost impossible, as they are rarely online at the same time. Thus, by having support for asynchronous interactions between the users, their collaborative work can be continued regardless of their connection status.

6.3 Application's services

In this section, we introduce the services offered by the application to mobile users. We categorized the services into two sets: one set offered for a master editor, and the other for a regular editor. These services are described next.

6.3.1 Services for a master editor

A user may create a document on the mobile terminal and wish to share it with other locally connected users. They can do this by adding the document to the system, permitting other users to view or modify the document's copy. The user is then recognized by the system as the master editor, who is given the sole permission to edit the master document (i.e. the corresponding document hosted on the user's terminal). The master editor can then continue working on the master document, and once the document is modified, the changes made to it can be shared with the other users by committing the new version of the document to the system. When relocates at a new local network, committing the document allows the master editor to share the document with another group of locally connected users. This way, the master editor can still maintain remote collaboration with users located at the previous locations, while at the same time collaborating with local users.

When the master editor requests the document's log from the system, he / she is made aware that other users may have checked out the document's copy and have modified it. Any changes

committed by the other users are known as tentative versions. The master editor would then review such tentative versions, and approve or reject the changes made in them, applying the accepted changes to the current version of the master document, i.e. merging the tentative versions with the latest version of the master document.

The system allows the master editor to delegate the editing rights to another collaborator, i.e. a regular editor. This delegation is successful only when it is approved by the requested regular editor. Once such delegation is successful, the master editor no longer has the editing right to the master document, which now is the corresponding document being hosted on the newly appointed master editor's mobile terminal.

Finally, one other service offered to the master editor is to remove the document from the fixed infrastructure. This way the system removes all versions of the document across the network, preventing other users from further access to document. Details of all copies and versions of a specific document are stored in the repository directory.

6.3.2 Services for a regular editor

The system allows a user to view information on all locally available documents. On request, a list of documents' details can be provided by the system, allowing the user to choose a particular document to be checked out. Once checked out, a document can later be viewed and modified by the user while being disconnected from the network. After modifying the document, the user can then commit the modified version to the system as a tentative version, awaiting review from the master editor. This way the changes made to the version can be applied to the master document upon approval of the master editor, at which time the system would notify the user of an acceptance or rejection by the master editor of the committed tentative version.

To view the current state of the document, the system allows the user to view the document's status, followed by a check-out of its latest available version. By editing and committing changes to a particular document, a user is regarded by the system as a regular editor of the document, and being a regular editor the user may receive a request from the system to be the new document's master-editor. The user can accept or reject the request. On accepting the request, the user is

given the editing rights to the master document, which is hosted on their mobile terminal, and is offered the services explained in Section 6.3.1.

6.4 The architecture

In this section, we present the collaborative editing application's architecture. It consists of two components, namely a user application that runs on mobile terminals and a repository that runs on the fixed network (cf. Figure 6.1), which we describe next.

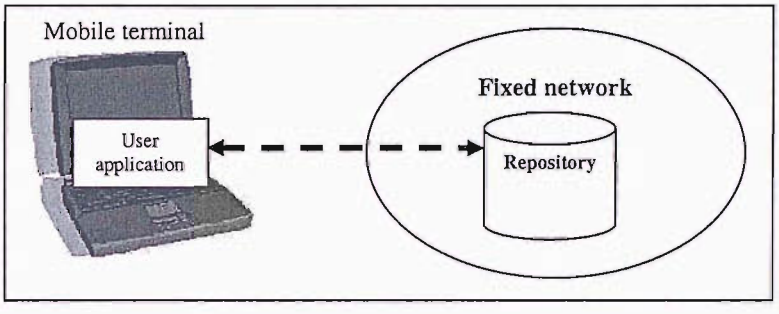


Figure 6.1: Collaborative editing application architecture

6.4.1 User application

A user application is the user's access point to services provided by the collaborative editing application. It consists of four sub-components: a user interface, a coordinator, a document database and a repository directory (cf. Figure 6.2).

The *user interface* has two functions. On the one hand, it supports interactions with users by offering them a list of activities in accordance with their roles: add a document, check out a document, commit changes and display the results of requested activities. On the other hand, the user interface interacts with the coordinator by forwarding users' requests and receiving results of the requests from the coordinator.

The *coordinator* creates requests for a repository once it receives requests from the user interface. Such request creation involves queries to be made to the document database and the repository

directory in order to get details of a document and associated repositories. Once created, the requests are forwarded to the repository.

The *document database* maps a document's identifier to the document's details, which include the document's version number and editing token. The repository assigns an identifier to the document the first time it is added to the fixed infrastructure. This identifier is used throughout the document's lifetime, i.e. until it is removed from the infrastructure. Master documents are associated with editing tokens, whereas checked out documents are not. When the document database receives a query about a given document's identifier, it returns the document's version number and editing token (when it exists).

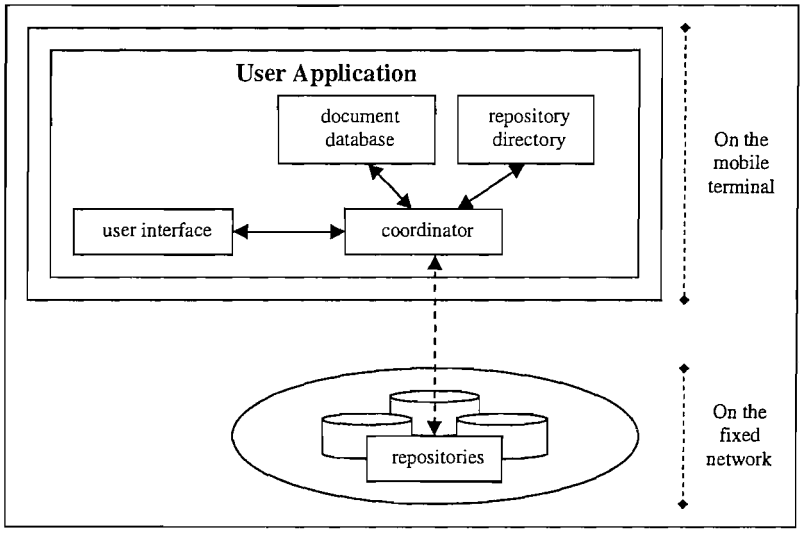


Figure 6.2: User application and repository

The *repository directory* maps a document's identifier to the identifiers and addresses of repositories that contain copies of the document on the fixed network. These repositories may host one or more versions of the same document, in which case the repository directory keeps the latest version number of the document held by each repository.

6.4.2 Repository

Repositories are the part of the application hosted on the fixed network that need to be available at all times to provide mobile users with access to copies of shared documents. The repository to

which a document's copy is initially added will create an editing token and assign an identifier to the document. The uniqueness of editing tokens and documents is required to distinguish them across the network; this can be achieved by using structured strings, e.g. URI or UUID (Universally Unique Identifiers) [96] to identify each one of them.

The collaborative editing application does not rely on only one repository; a user application can be associated with multiple repositories that are distributed across the fixed network. A user application on a mobile terminal is assumed to interact with a local repository due to our belief that it is important to have a local interaction rather than a remote one. The short-range connection between a mobile terminal and a repository can overcome network delays, which is particularly significant for a mobile terminal as it could face wide variations and rapid changes in network conditions. Additionally, manipulation of a local repository is also useful when the local network is not connected to the Internet.

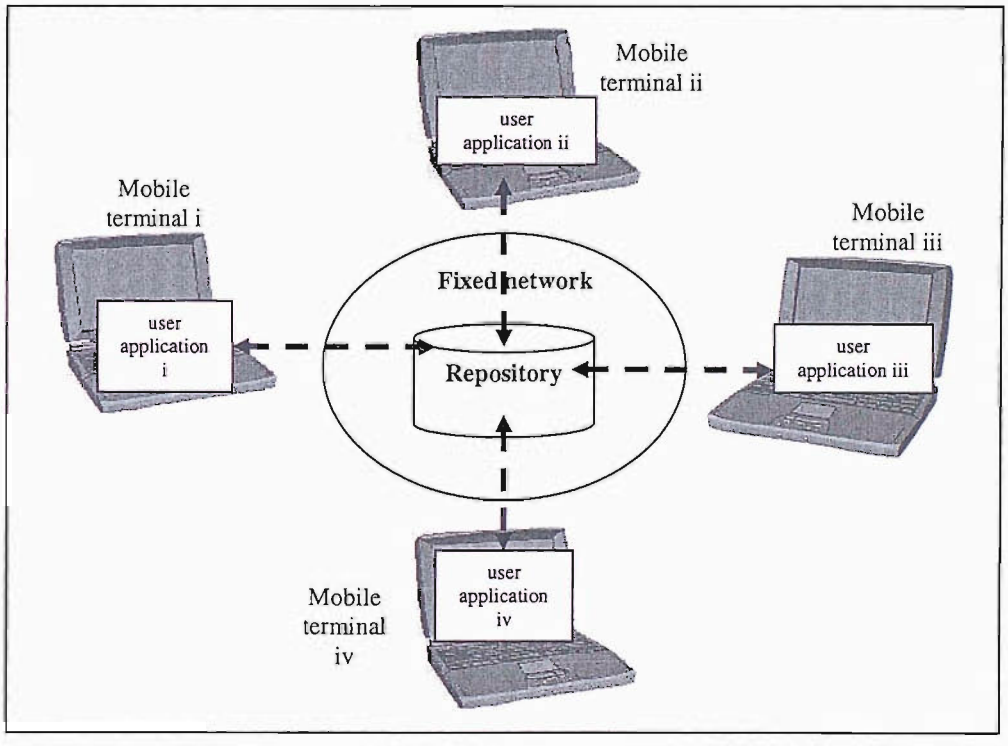


Figure 6.3: One repository serving multiple users

Thus, on connection to the fixed network, a user application tries to initiate an interaction with a local repository. It starts by finding information on a local repository from a local lookup

directory or local services directory such as Jini [81]. Using the information obtained, the user application is then able to start interacting with the repository. The mobile terminal continues to be served by the repository for as long as the mobile terminal is connected to the same network. From a repository's point of view, at any one time it may need to serve more than one user application hosted on different mobile terminals that are connected to the same local network (cf. Figure 6.3).

6.5 Collaboration Protocol

In this section, we explain the collaboration protocol that supports collaborative editing between mobile users, by adopting some definitions from [24]. These definitions are based on the *coordinator* aspect, i.e. one aspect of the collaborative technologies related to the ordering and synchronization of individual activities that make up the whole process, specifically adopted in the Computer Supported Cooperative Work (CSCW) / groupware field [19][24][25]. A *process*, with an associated goal or specific purpose, can be summarized as a sequence of *activities* that are carried out by groups of *actors* and result in the handling of some *objects*. Thus, specifically, our collaboration protocol defines the ordering and synchronization of the collaborative editing processes and activities included in each process. We elaborate this further below.

6.5.1 Actors

An actor is a human user or an application, who or which is responsible for initiating or performing a process (cf. Figure 6.4). We consider a *mobile user* as an initiator of processes who uses the user application to access services offered by the collaborative editing application. At any time, the user may choose an entry from the user interface's menu, which is translated into a request that initiates a new process. We characterize mobile users further with different roles. A *role* for a user is regarded as the given rights to initiate specific types of processes. We have defined two roles for mobile users, namely *master editor* and *regular editor* for each document. For example, for a given document, a process such as "pass-editing-token" is authorized to be initiated only by a user who has the role of "master editor".

The *user application* and *repository* are also two actors; they are components of the application that were described in Section 6.2.1 and Section 6.2.2. Both user application and repository have to react to occurring events according to a process description, which we will elaborate upon in Section 6.3.3. Below we describe the association of the user application and repository with other actors.

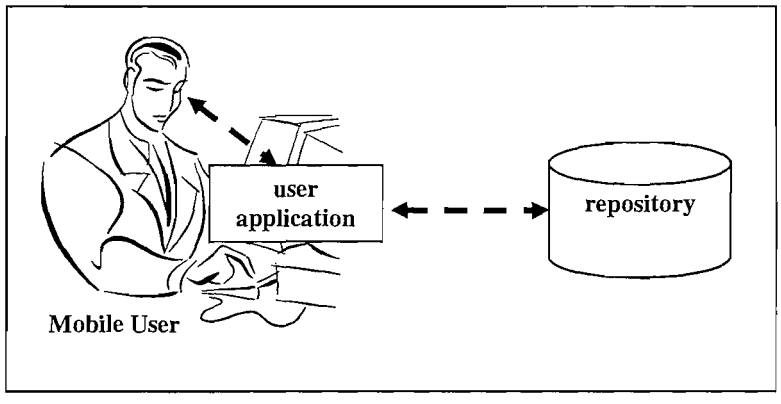


Figure 6.4: Actors in the collaboration protocol

One user application is associated with one mobile user. While on the fixed infrastructure, the user application is linked to the repositories. A user application is mainly associated with a locally available repository, which is treated as its main repository. At the same time, a user application may also interact with other remote repositories storing copies of documents for which the user is their master editor. This happens when those repositories are storing copies of a document, i.e. identified as $ID^{doc\text{-}n}$, which is being collaboratively edited by the mobile user and other collaborators. In this case, the document is associated with multiple repositories, in which case the mappings between $ID^{doc\text{-}n}$ and all the addresses of repositories ($\alpha^{rep}s$) are stored in the repository directory (see Section 6.2.1). Such an association is also illustrated in Figure 6.5.

At any one time, a user application can manage more than one document on behalf of the user. A mobile user may collaborate in editing multiple documents in parallel, in which case, for each document, the mobile user has an associated role. As an example, consider a scenario in which a mobile user collaborates with other users in editing three separate documents identified as $ID^{doc\text{-}n}$, $ID^{doc\text{-}m}$ and $ID^{doc\text{-}o}$. In such a scenario, the user may have the role of a master editor for $ID^{doc\text{-}n}$ while at the same time he or she may have the role of a regular editor for $ID^{doc\text{-}m}$ and $ID^{doc\text{-}o}$. Such

information is held by the document directory (cf. Section 6.2.1), which specifically stores details of documents including the roles of the users associated with each of them.

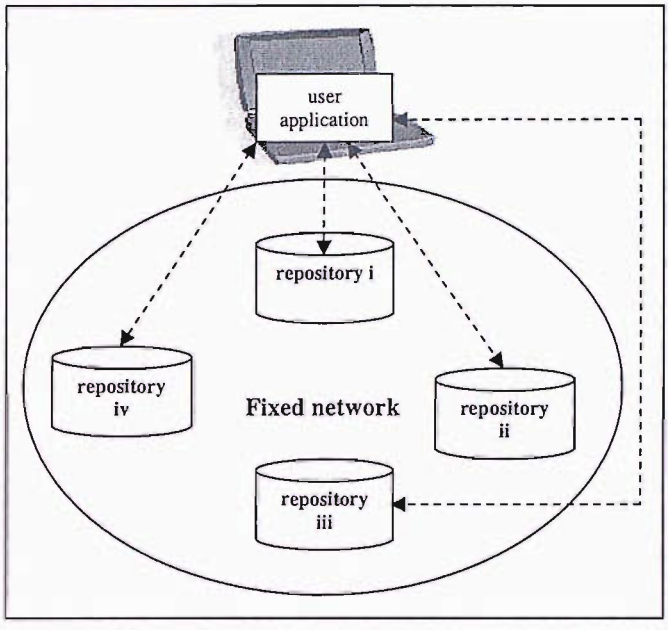


Figure 6.5: A user application associated with multiple repositories

In serving multiple user applications, a repository may handle one or more documents. A specific document can be shared and edited by multiple users and roles; thus, for a group of collaborators working on a document, the repository stores a list containing the collaborators' details. This includes the users' IDs, roles and addresses. Such associations are illustrated in Figure 6.6. A user ID is the distinguished name of the user, e.g. the user's name in a certificate, login name or email address.

In the figure, collaboratorList^{n} contains details of collaborators working on $ID^{doc\text{-}n}$, while collaboratorList^{m} and collaboratorList^{o} contain details of collaborators who work on $ID^{doc\text{-}m}$ and $ID^{doc\text{-}o}$ respectively. Peter, John and Mary are the user IDs of the collaborators, while α appended with a user ID refers to the address of the referred user. By default, α of a user refers to a list allocated to hold notifications for the user hosted by the repository. Alternatively, a user's α can be the user's email address, in which case the repository will have a mechanism to deliver the notification to the user's email address. In terms of roles, to differentiate between roles of a user, each role is appended with the document's id.

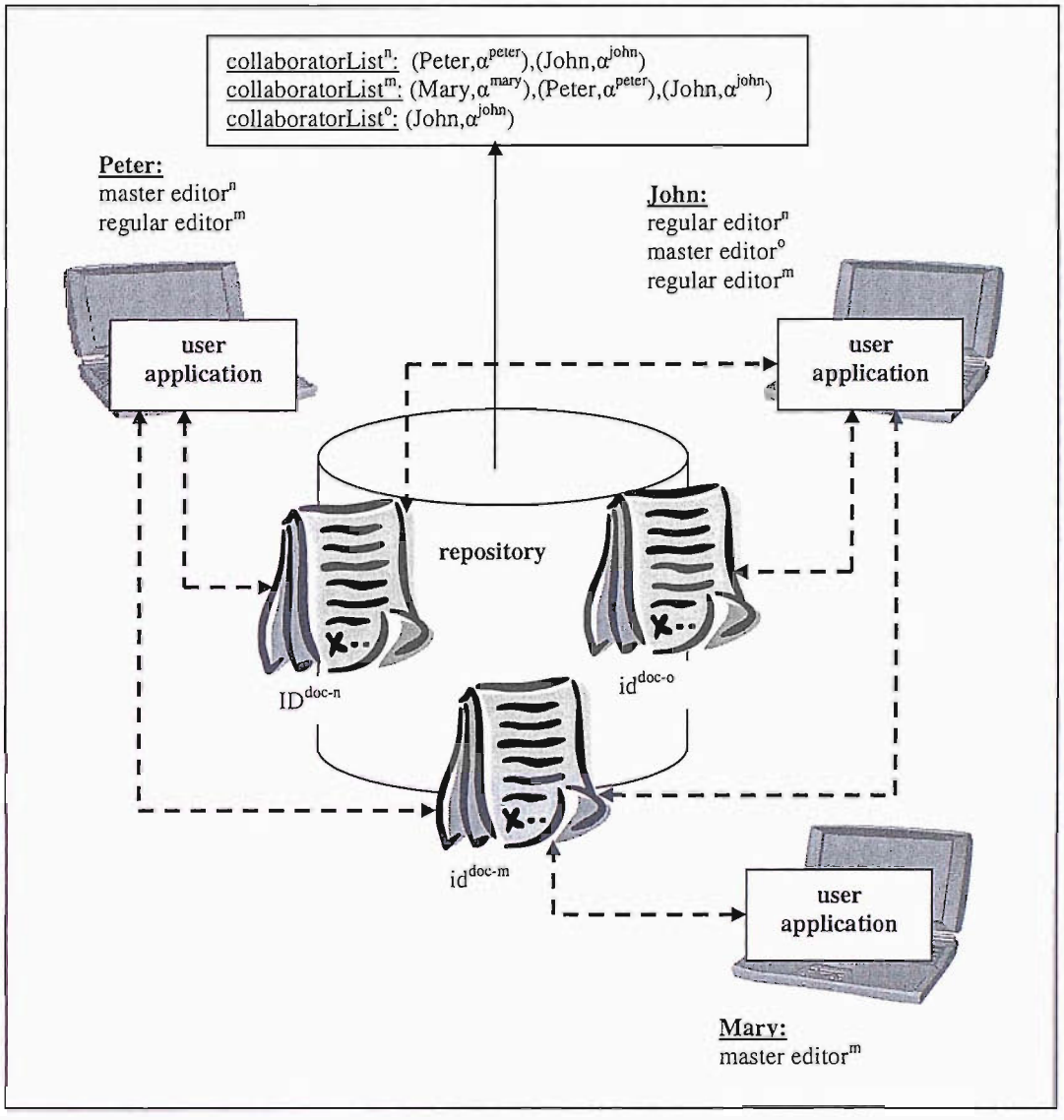


Figure 6.6: Associations between a repository and multiple documents and user roles.

6.5.2 Objects

An object is considered as a passive and non-reactive entity: it can be created, modified, passed or handled by actors. We define two kinds of objects, namely *documents* and *editing tokens*. On the one hand, documents stored in repositories may not be updated directly: they can be added, checked out and removed from repositories; such documents are therefore identified as read-only objects (cf. Figure 6.7). On the other hand, for a document to be editable it must be located on a

user's terminal. Hence, we adopt a terminology that distinguishes such documents. Documents that are hosted on mobile terminals are simply referred to as *documents*, whereas documents' replicas stored in repositories are referred to as *copies*. The document hosted on the master editor's terminal is known as the master document, while a document checked out by a regular editor is known as a regular document. Copies are also further categorized into two types, namely major versions and tentative versions. A major version refers to a copy that is committed by a master editor and available to other users, whereas a tentative version is committed by a regular editor and can only be viewed by the master editor.

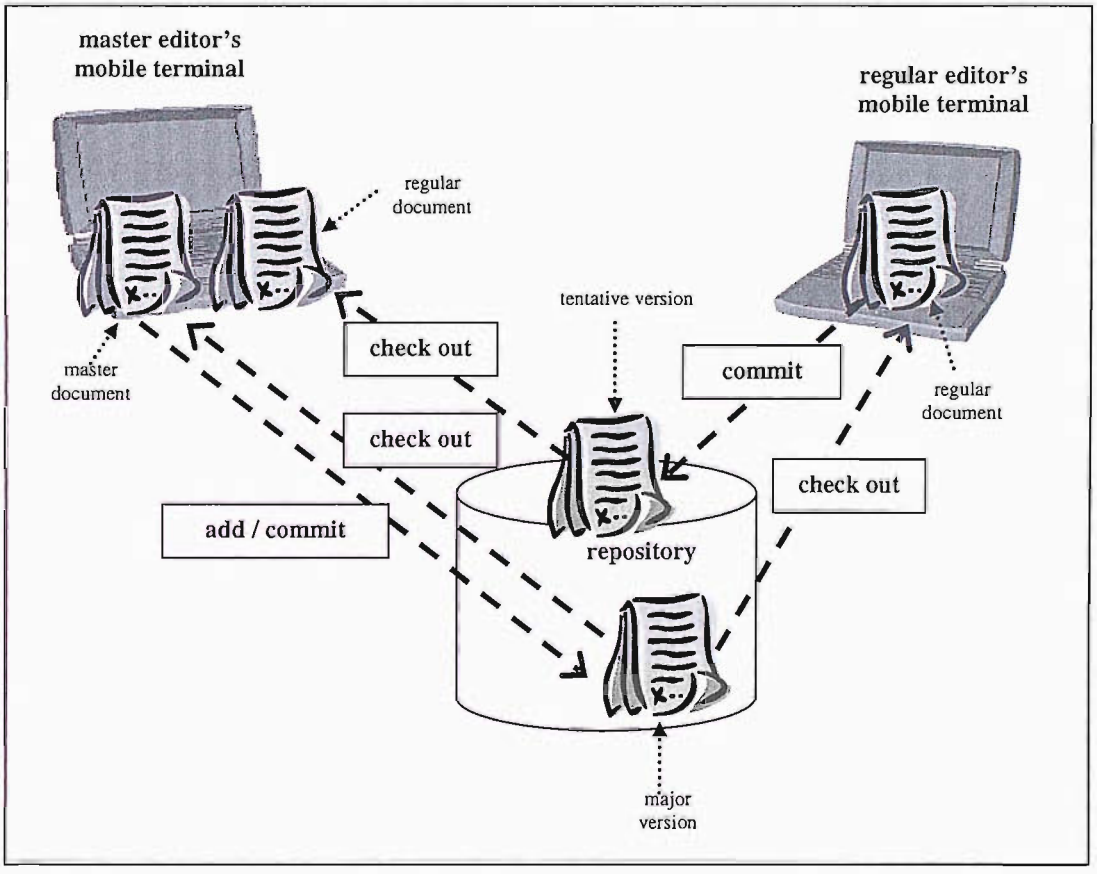


Figure 6.7: Document replication and life cycle.

Editing tokens are also objects created by repositories and are unique for each shared document. The editing token is passed between mobile collaborators, from a master editor to a regular editor; after transfer, the token's sender reverts to being a regular editor, whereas its recipient becomes the master editor. Thus, the holder of the editing token is known as the master editor. We use

cryptographic transformations of editing tokens so that repositories can prove the source and integrity of editing tokens and to protect them against forgery [90].

6.5.3 Process

A collaborative editing series involves a number of processes being initiated by collaborating mobile users. Each process has a goal associated with it, and to realize such a goal, a sequence of activities needs to be performed by the user application and the repository. For the purpose of illustration, we now describe a sequence of processes performed by a group of collaborators to obtain the final version of the document (cf. Figure 6.8). *Master editor*^{n} refers to a user assigned as the master editor for the document identified as $ID^{doc\text{-}n}$, while *regular editor*^{n} refers to a normal user of the document.

Roles	***Processes***
Master editor^{n}	o Add a document o Commit the $ID^{doc\text{-}n}$'s master document as a major version o Check out a major / tentative version of $ID^{doc\text{-}n}$ o Review an $ID^{doc\text{-}n}$'s tentative version o Remove all $ID^{doc\text{-}n}$'s copies from the fixed network o Pass editing token to another collaborator o Synchronise $ID^{doc\text{-}n}$'s versions
Regular editor^{n}	o Check out an $ID^{doc\text{-}n}$'s tentative version o Commit $ID^{doc\text{-}n}$'s regular document

Figure 6.8: List of authorized processes for each editing role

The *add document* process is initiated by a master editor in order to make a mobile document accessible to other users by adding a copy of the document to a repository on the fixed network. The *commit* process can be initiated by either master editor or regular editor; changes committed by the master editor result in a latest major version of the document being created, whereas changes committed by a regular editor result in a tentative version being created. The *check out* process can also be performed by both kinds of editors; the master editor may check out both

major and tentative versions of the document, whereas a regular editor may only check out the latest major version of the document. The *review tentative version* process is only permitted for the master editor, who can review a tentative version of the document and decide whether to approve or reject the changes made by a regular editor. Subsequently, the regular editor will receive a notification on the master editor's decision. The *pass editing token* process is initiated by a master editor who decides to delegate the master editor role to another collaborator. Finally, the *synchronise version* process is initiated by a master editor in order to request a local repository to obtain the latest version of the document from the mobile terminal or another repository. The master editor may have committed the latest version of a document to a repository at a previous location and when the master editor arrives at a new location, the master editor may choose to synchronise the document between a local and the remote repository. This way, the locally connected users at the master editor's new location can have access to the document.

In a collaboration, all collaborators are involved in constructing the document to some extent. As an illustration, Figure 6.9 shows that a master editor is involved in a sequence of processes: to *add a document* (1), to *review tentative changes* (5) and to *commit* (6) a document; a regular editor is following another sequence of processes: to *check-out* (2), to make changes to the checked out document and to *commit* (3) the changes. We distinguish the tentative version, i.e., $V_{1.1}$, resulting from a commit by the regular editor, from the latest major version (i.e., V_2) committed by the master editor. Changes in tentative versions have to be accepted by the master editor before they are merged to the latest version.

We envisage a system that allows parallel editing of a document among a group of users, but at the same time we want to give the sole editing right of the document to one user, i.e. the master editor. Thus, we distinguish the changes made by the master editor by regarding them as having higher priority than changes made by regular editors. For the latter, approval from the master editor is needed before the changes are applied to the master document. Thus in labeling these two classes of changes, we name the committed versions with master editor's modifications as major versions, while versions modified by regular editors are known as tentative versions.

Accomplishing a collaborative editing task involves a number of processes that are performed by a group of collaborating mobile users. Figure 6.8 shows a list of processes that can be initiated by a mobile user. Each process normally has some goals associated with it. To realize these goals, a sequence of activities needs to be performed by the user application and repository. These

activities will be further explained in Section 6.5.4. Here, we will start by categorizing processes into their authorized roles. This means that some processes are allowed to be performed only by users with specific roles. First, we list all the processes authorized to be performed by a master editor, followed by the processes authorized for a regular editor.

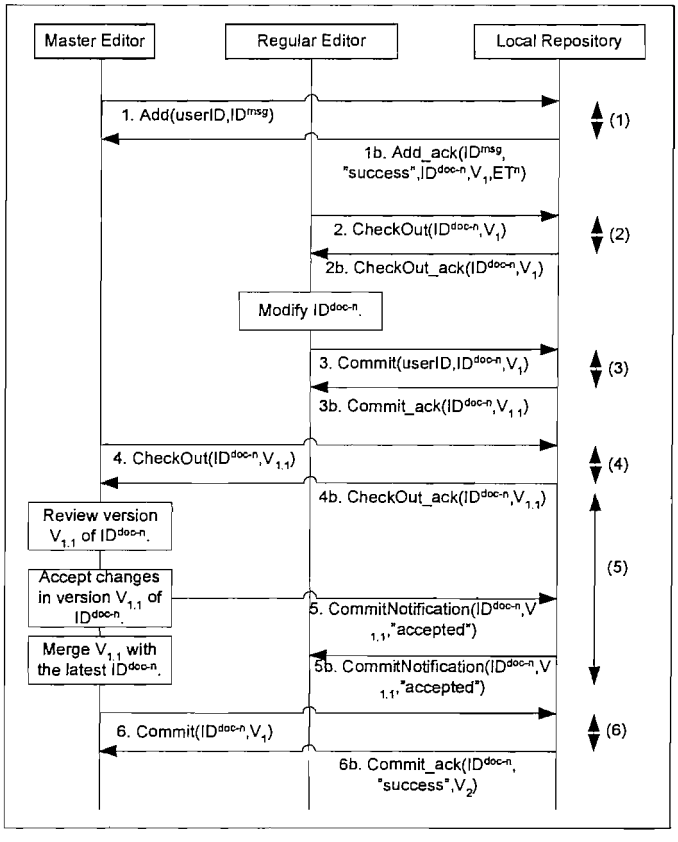


Figure 6.9: An example sequence of collaborative editing processes

6.5.4 Activities

An activity is a unit of work or action to be performed either by the user application or the repository. In general, some activities cannot be performed until other activities are completed. In describing the activities, we use the same combination of sequence and activity diagrams as described in Section 5.3, but without the location indicator.

6.5.4.1 Master editor adds a document

We now describe activities performed in the *add document* process (cf. Figure 6.10). The goal of this process is to make a document accessible to other mobile users by having its copy stored in a repository on the fixed network. First, a user selects the document to add; then, the user application sends an "add" request message that includes the user's identifier (ID^{user}), the message identifier ($ID^{message}$) and a copy of the document to the repository. The $ID^{message}$ is used as a session identifier, and therefore is also included in the reply from the repository. On receipt of the request, the repository stores the transferred document, marks it as a *master-copy* and assigns to it a unique identifier ($ID^{doc\text{-}n}$) and a version number (V_1). An editing token (ET^n) and a collaborator list (collaboratorList^n) (cf. Section 6.5.1), which are stored in the repository, are also created for the copy. The collaborator list enumerates the identifiers and addresses of the document's master editor and regular editors; the collaborator list is updated with user's details whenever the user has committed the modified version of the document. Such information is needed to enable asynchronous interactions between master and regular editors. The repository then returns an acknowledgement message to the user; it includes the session identifier ($ID^{message}$), the operation status (i.e. "success" or "failure"), the document identifier ($ID^{doc\text{-}n}$), the version number (V_1) and the editing token (ET^n). On receiving this message, the user application stores details of the successfully added document. Finally, the user application notifies the master editor of the successful addition of the document identified as $ID^{doc\text{-}n}$ to the repository.

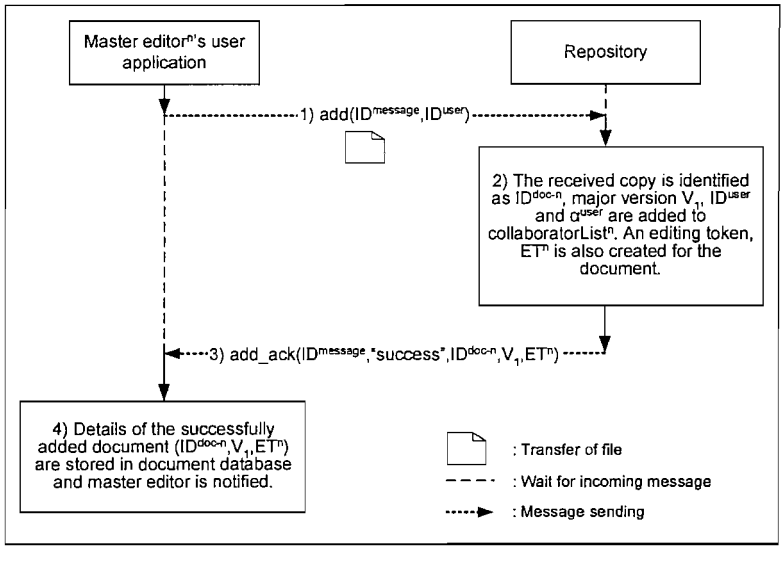


Figure 6.10: Master editor^n adds a document.

6.5.4.2 Master editor^n commits the master document as a major version

After performing some changes to the document, master editor^n may decide to commit the updated document to the repository. Once this request has been received by the user application, it sends a "commit" request to the repository (cf. Figure 6.11). Besides a copy of the updated document, the user application which keeps tracking and storing information about the latest state of the document, includes the document's id, $ID^{doc\text{-}n}$, the current version number of document, V_x , and document's editing token, ET^n, in the commit request. Upon receiving this request, first of all the repository checks whether ET^n is valid. If it is valid, the repository proceeds by committing the document as a new major version of the document. In our design, committing a modified version of a document involves the creation of a new copy, representing the latest version of the document in the repository without overwriting its previous version. This action is performed by the repository in order to prevent any inconsistencies between existing versions of the document. The repository assigns a new version number to this new copy, $V_{(x+1)}$, and then returns an acknowledgement message to the user application. This acknowledgement message contains $ID^{doc\text{-}n}$ and $V_{(x+1)}$. On receiving this message, the user application stores $ID^{doc\text{-}n}$, $V_{(x+1)}$ and ET^n. Finally, the user application notifies the user of the successful commit request.

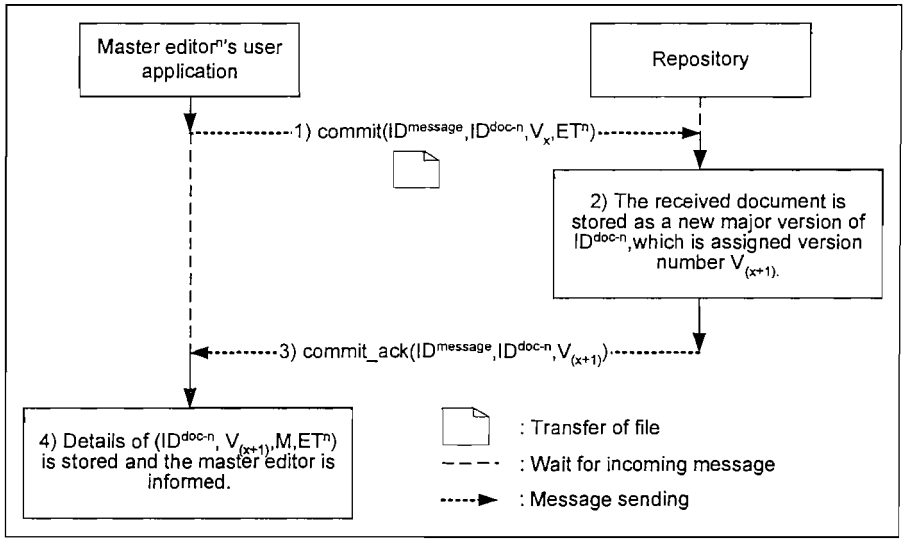


Figure 6.11: Master editor^n commits the master document.

6.5.4.3 Master editor^n checks out a copy of ID$^{doc\text{-}n}$

In order to check out the major versions of ID$^{doc\text{-}n}$ from a repository, a user application must first request details of major versions of ID$^{doc\text{-}n}$ from the repository (cf. Figure 6.12). To do this, the user application needs to send a "log" request to the repository, containing the document's ID (ID$^{doc\text{-}n}$), the document status (i.e. "major" for getting details of major versions; "tentative" for tentative version and "all" for both major and tentative versions of ID$^{doc\text{-}n}$), and lastly the document's editing token (ET^n). On receipt of this request, the repository extracts details of all major versions and places them in a list *list*^{n}, which is then included in an acknowledgement message, returned to the user application. On receipt of this message, the user application extracts information from list^n and displays it to master editor^n. From the displayed information, master editor^n can proceed by selecting a particular version of ID$^{doc\text{-}n}$ to be checked out from the repository.

Based on master editor^n's selection, the user application will send a "check_out" message to the repository. The message contains details of ID$^{doc\text{-}n}$'s version requested by master editor^n. In Figure 6.13, the check out message contains ID$^{doc\text{-}n}$, V_z, the version type, i.e. "major" or "tentaive" version, and ET^n. In this case, the message can be translated as a request received from master editor^n to check out a tentative version of ID$^{doc\text{-}n}$, which has the version number, V_z.

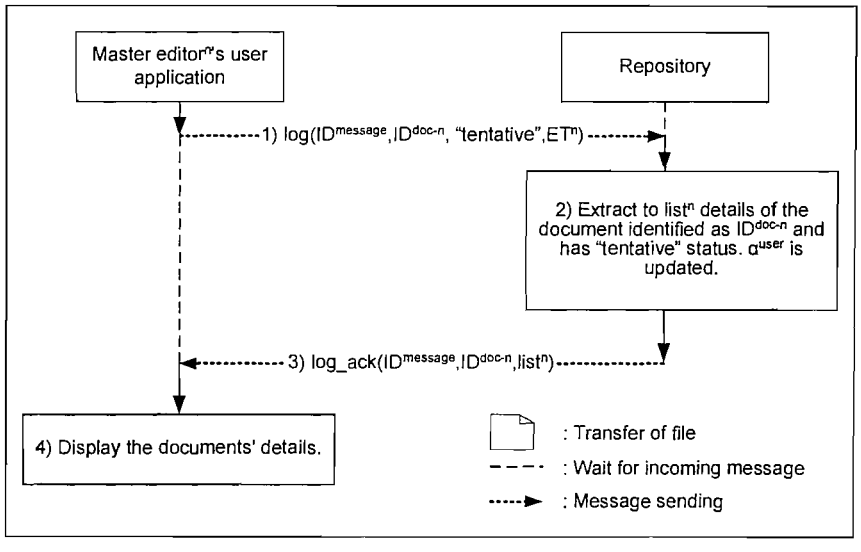


Figure 6.12: Master editor^n requests a log on ID$^{doc\text{-}n}$.

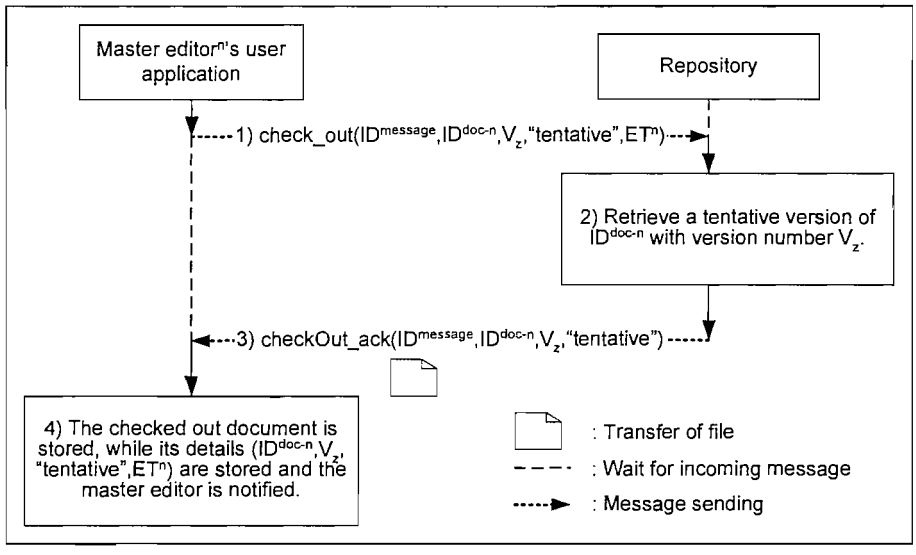


Figure 6.13: Master editor^n checks out a copy of document.

6.5.4.4 Master editor^n reviews a tentative version of ID$^{doc\text{-}n}$

A master editor^n who wants to review changes made by other users must first check out a particular tentative version of ID$^{doc\text{-}n}$ from the repository. The check out process was explained in Section 6.5.4.3. Once a required tentative version has been checked out, master editor^n may choose to perform "review document" from the user application's menu (cf. Figure 6.14). The user application then extracts details of the available ID$^{doc\text{-}n}$'s tentative versions and displays them to master editor^n (cf. Figure 6.15).

Figure 6.15 shows us an example of a list being displayed by the user application to master editor^n. The list contains details of ID$^{doc\text{-}n}$'s tentative versions, which include their version numbers and the user IDs of collaborators who made changes to them. From this list, master editor^n will choose a particular tentative version to be reviewed. As illustrated in Figure 6.15, we can see that master editor^n chooses to review a tentative version, $V_{8.3}$ of ID$^{doc\text{-}n}$, which was modified by another collaborator with the user id, John. The user application will then open the selected tentative version of ID$^{doc\text{-}n}$ and display it to the user.

Master editor^n can then start reviewing the document and may also make some changes to it. When master editor^n has saved the file, the user application asks master editor^n whether he or she

wants to merge the reviewed tentative version with the master document. Alternatively, the user may postpone the decision making to another time (cf. Figure 6.16).

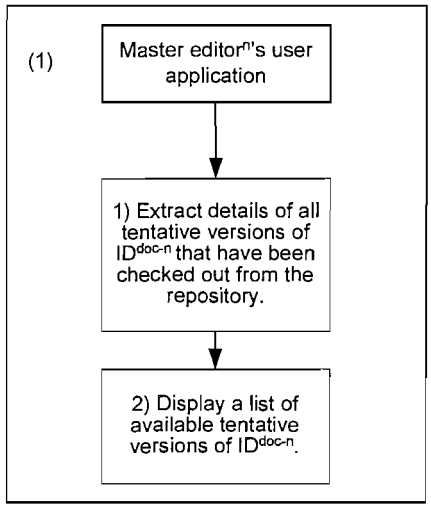


Figure 6.14: Master editor^n chooses to review changes made by other collaborators.

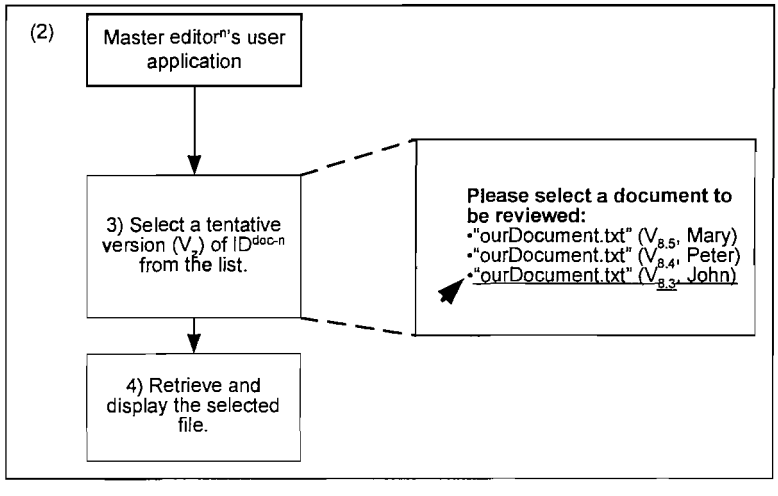


Figure 6.15: Master editor^n selects a tentative version to be reviewed from the list.

Then, master editor^n chooses one option from the request box displayed by the user application. In Figure 6.17, we can see that master editor^n chooses to merge the reviewed tentative version with the master document. This action causes the user application to merge both files into another copy of the document, which is considered as the latest major version of $ID^{doc\text{-}n}$. At any one time,

the version number of a master document is the one that was assigned by the repository, which was included in the acknowledgement message received from the repository after an "add" or "commit" request was successfully performed. In a case such as this, where the master document is merged with changes made by a regular user, the merged document is now known as the uncommitted version of the master document. Although this master document has been modified, it continues to carry the same version number until master editor^n commits this modified document to the repository and gets a new version number assigned by the repository.

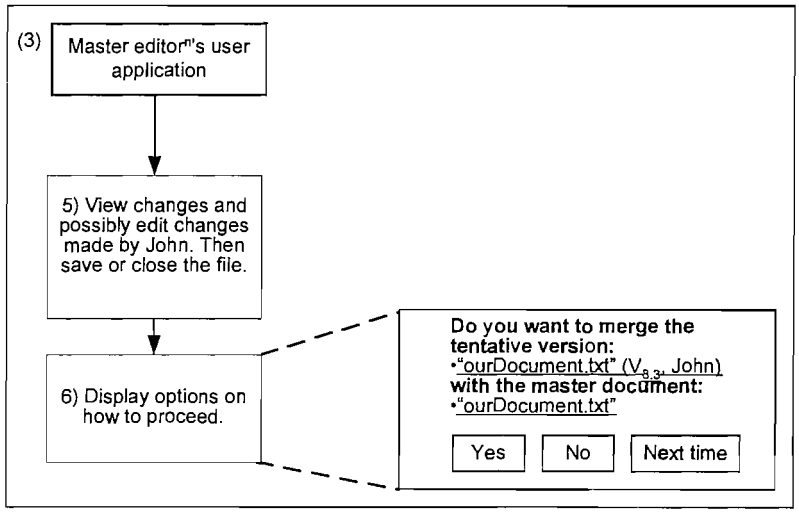


Figure 6.16: Master editor^n finishes reviewing the file and ready to merge changes in the reviewed tentative version with the master document.

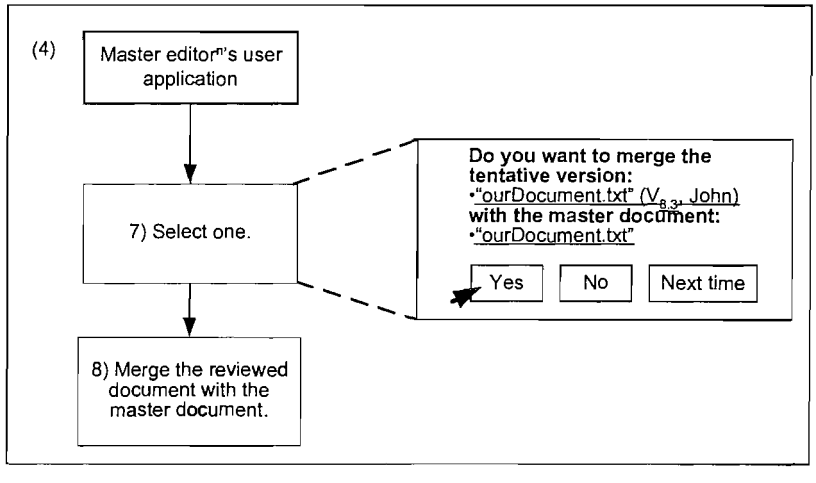


Figure 6.17: Master editor^n decides to merge the tentative version with the master document.

Referring back to Figure 6.16, if master editor^n chooses "No" or "Next Time", the user application will do no merging of the document. Besides interpreting the chosen option as either to merge or not to merge the documents, the user application also considers the selected "Yes" as the way that master editor^n formally accepts the changes made by another collaborator, while "No" rejects the changes and "Next Time" postpones the decision to accept or reject the changes. Thus, when a "Yes" or a "No" is selected, the user application sends a "commitNotification" message to the repository. This message contains details of the reviewed tentative version of $ID^{doc\text{-}n}$, as well as the "acceptance" or "rejection" status of the decision made by master editor^n about changes in a tentative version. This is shown in Figure 6.18.

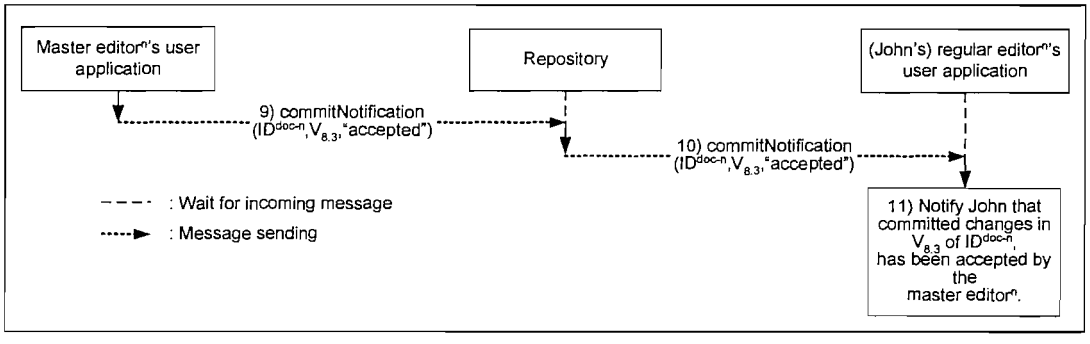


Figure 6.18: Regular editor^n receives notification of acceptance or rejection of changes being made to an $ID^{doc\text{-}n}$'s tentative version

On receipt of a "commitNotification" message that contains a tentative version's details and master editor^n's review decision, the repository extracts details of regular editor^n, who has made changes to the tentative version. The extracted details include regular editor^n's address, e.g. the user's email address, to which the repository will forward the received "commitNotification" message. An example is illustrated in Figure 6.18, where on receipt of a "commitNotification" message for a tentative version, with version number $V_{8.3}$, the repository will extract the details of John, who has made changes to $V_{8.3}$. Once this is done, the repository will then be able to forward the "commitNotification" message to John. On receipt of this message, John's user application will inform John about the notification.

6.5.4.5 Master editor^{n} removes ID$^{doc\text{-}n}$'s copies from the repository

Master editor^{n} may decide to remove ID$^{doc\text{-}n}$'s versions from the repository. When such a removal request is issued by master editor^{n}, the user application sends a "remove" request to all repositories that have copies of ID$^{doc\text{-}n}$ (cf. Figure 6.19). The user application does this by first extracting the details of all repositories from the repository directory, and then sending a "remove" request to each of these repositories. This request contains ID$^{doc\text{-}n}$ and ET^{n}. On receipt of this request, each repository first checks whether ET^{n} is actually the right editing token. If it is, the repository then removes all copies of ID$^{doc\text{-}n}$.

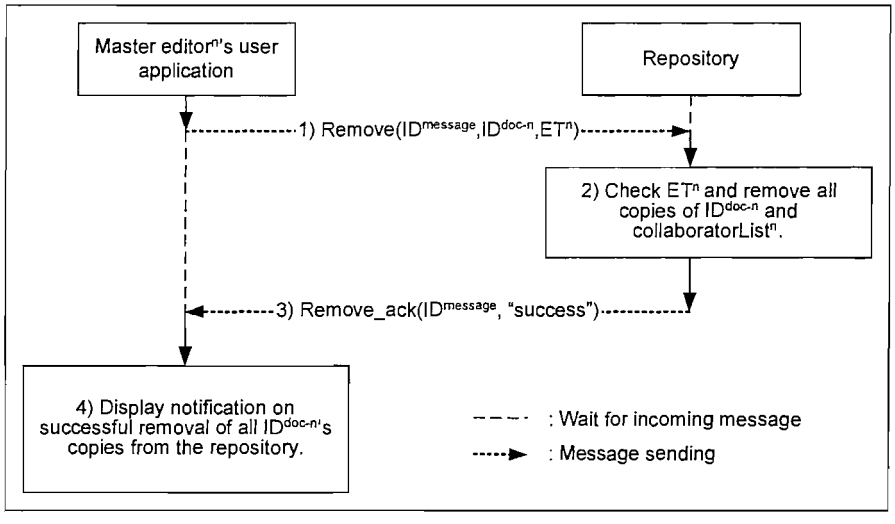


Figure 6.19: Master editor^{n} removes all ID$^{doc\text{-}n}$'s copies from the repository

6.5.4.6 Regular editor^{n} checks out a copy of ID$^{doc\text{-}n}$

Before a user can check out a copy from a local repository, the user application must first send a "log" request to the repository (cf. Figure 6.20). This request is sent in order to get details of all available copies in the repository, in which case an "all" status is set in the "log" request. For the same purpose, the request is not specified with a particular document identifier, ID$^{doc\text{-}n}$. On receipt of this request, the repository extracts information on the latest major version of all available copies. The information includes copies' IDs and their version numbers, which are included in a list called List^{x}. This list is then included in an acknowledgement message that is returned to the requesting user.

As illustrated in Figure 6.20, once a user application receives a "log_ack" message from the repository, it displays the extracted information from the received list. From the displayed list, the user may then choose which copies he or she wants to check out.

The listed copies are publicly available and can be checked out by any user. But a master editor^n may decide to make $ID^{doc\text{-}n}$'s copies available only to certain users. This is done by having selected user IDs in the collaborator list, so that the repository will only allow these users to check out and modify the document. Thus, options provided to master editor^n include:

- To make $ID^{doc\text{-}n}$'s copies available to other users (read and modify / read only).
- To make $ID^{doc\text{-}n}$'s copies available only to a group of users, in which case the user IDs of the accepted collaborators are added to the collaborator list (read and modify / read only).

Once a document has been selected by the user, the user application sends a "check_out" request to the repository. This request contains $ID^{doc\text{-}n}$, which is the identifier of the selected document. Once this request is received by the repository, it will return the latest major version copy of $ID^{doc\text{-}n}$ to the user together with an acknowledgement message that contains $ID^{doc\text{-}n}$ and its version number, e.g. V_8 (cf. Figure 6.22). On receipt of this reply, the user application saves the received copy, marks it as a regular document and stores the document's details including its ID and version number. The user is now identified as a regular editor^n.

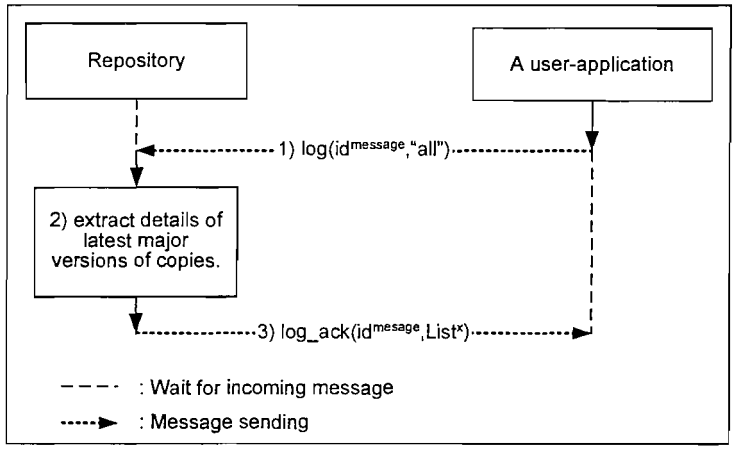


Figure 6.20: A user requests for a log on all available documents.

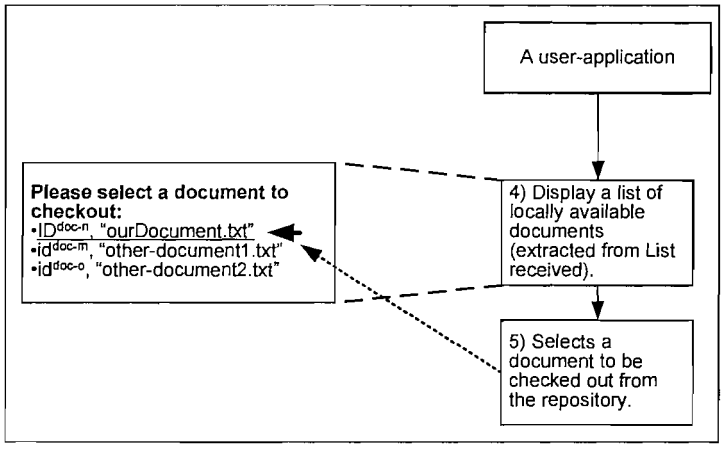


Figure 6.21: User application displays available documents that can be checked out.

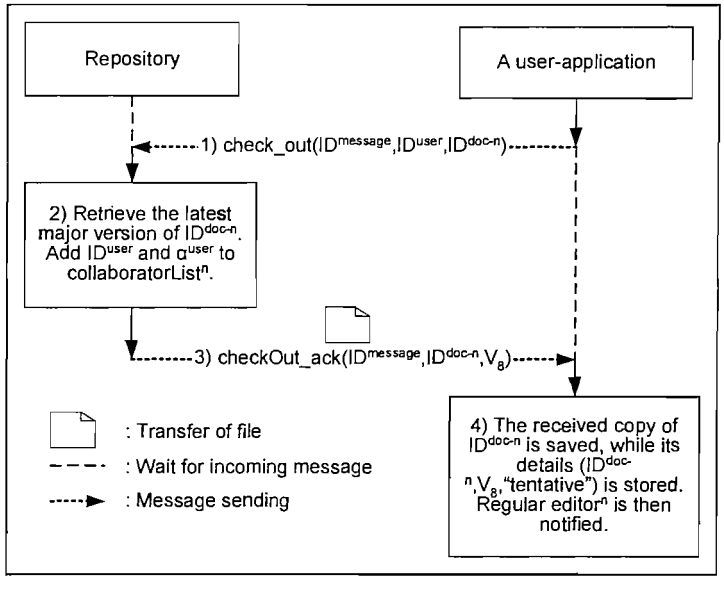


Figure 6.22: Regular editor^n checks out the latest major version of ID$^{doc\text{-}n}$

6.5.4.7. Regular editor^n commits a tentative version of ID$^{doc\text{-}n}$

Following a check out process, a regular editor^n may modify the checked out document and later decide to commit it. Once regular editor^n has selected the "commit" option from the user application's main menu, the user application sends a "commit" request to the repository (cf. Figure 6.23). This request contains the ID (ID$^{doc\text{-}n}$) and version number (V_8). On receipt of this request, the repository knows that the requesting user is just a regular editor^n, causing it to

commit the document as a tentative version of $ID^{doc\text{-}n}$, e.g. in this case the version assigned is $V_{8.1}$. Afterwards, the repository sends an acknowledgement message containing $ID^{doc\text{-}n}$ and new version number $V_{8.1}$ to the regular editor^n.

By committing the document, the regular editor^n knows that changes he has made will be merged with the same major version that he has previously checked out, i.e. V_8, although the merged document can then be modified again by master editor^n. To explain this further, a sequence of related activities is explained below.

The regular editor^n, i.e. John, checks out a major version of $ID^{doc\text{-}n}$, which has the version number V_8. John makes some changes to the checked out document. John commits the modified document and receives an acknowledgment from the repository that includes details of the committed document. The repository identifies the committed document as a tentative version of $ID^{doc\text{-}n}$ and assigns to it a new version number, i.e. $V_{8.3}$. This tentative version ($V_{8.3}$) is then reviewed by the master editor. The version is acceptable for review by the master editor because at this point the latest version number of the master document is still V_8. The regular copy ($V_{8.3}$) is still considered not to be outdated since the changes were made by John on V_8 of the master document. If the master document had a newer version, i.e. V_9, then this regular-copy ($V_{8.3}$) would no longer be acceptable, as it would be considered to be outdated. John would be notified that he must first update the checked out document to the latest version of the master document before re-committing the changes. Master editor^n commits the new master document to the repository and at this point the repository assigns V_9 as the version number of the master document (cf. Figure 6.24). By receiving a "CommitNotification" message with an "accepted" status, John acknowledges that the committed tentative version ($V_{8.3}$) will be merged with version V_8 of the master document, which may have been modified by the master editor^n.

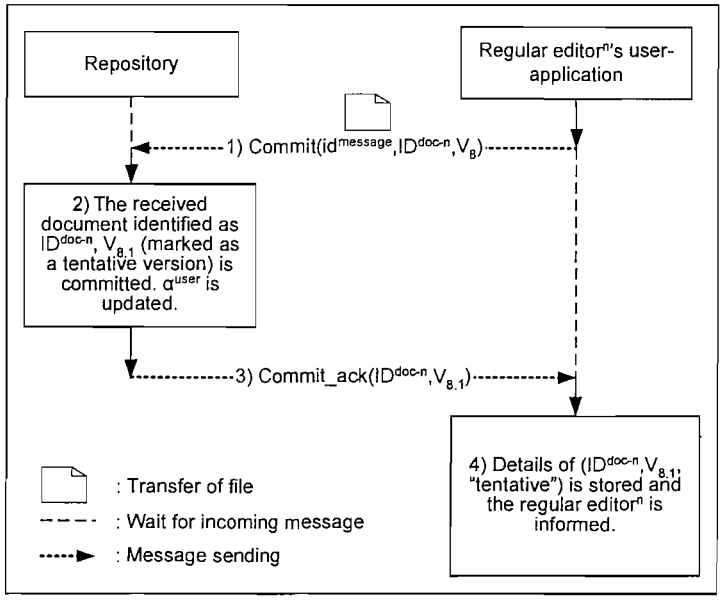


Figure 6.23: Regular editor^n commits the updated regular-document as a tentative version of $ID^{doc\text{-}n}$

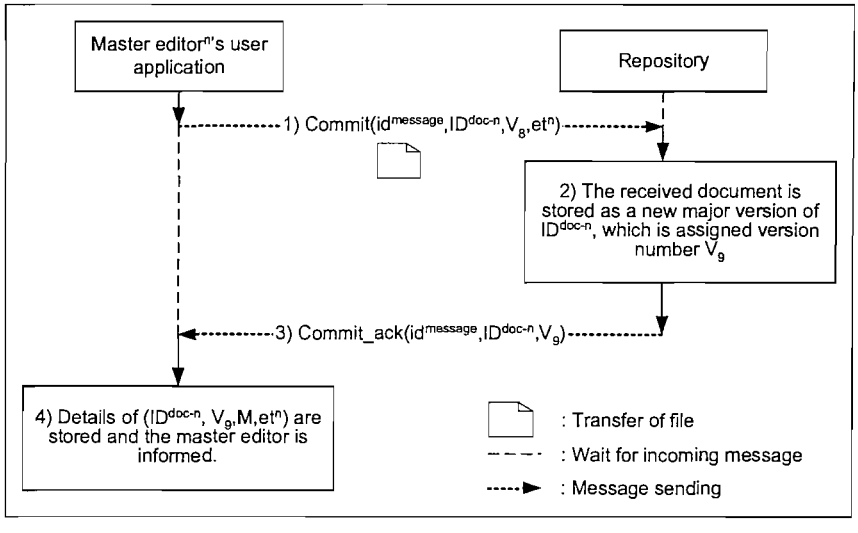


Figure 6.24: Master editor^n commits the master document as a new major version

6.5.4.9. Master editor^n passes editing token to a newly appointed master editor

At any time during the collaboration, master editor^n may decide to stop being the master editor for $ID^{doc\text{-}n}$ and pass ET^n to another collaborator. To do this, once requested, master editor^n's user

application sends a "getCollaboratorList" message to the repository (cf. Figure 6.25). In return the repository sends an acknowledgement message containing a list of collaborators' IDs and addresses, e.g. user's email address.

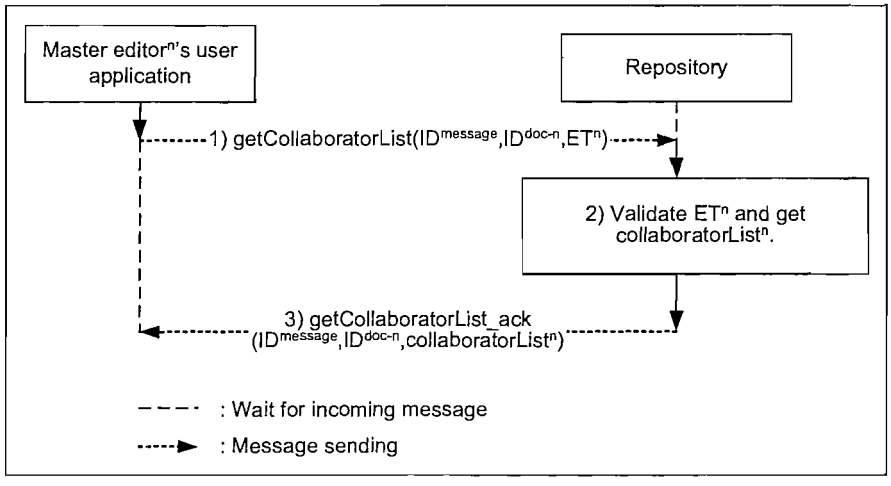


Figure 6.25: Master editor^n requests $ID^{doc\text{-}n}$'s collaborator list.

On receipt of the acknowledgement message, the user application extracts the IDs of the collaborators and displays them to master editor^n. From this list, master editor^n chooses a collaborator to be the next master editor^n. For example, Figure 6.26 illustrates that master editor^n chooses John as the next master editor^n. To α^{john}, the user application sends a "passEditingToken" message, which contains $ID^{doc\text{-}n}$ and ET^n.

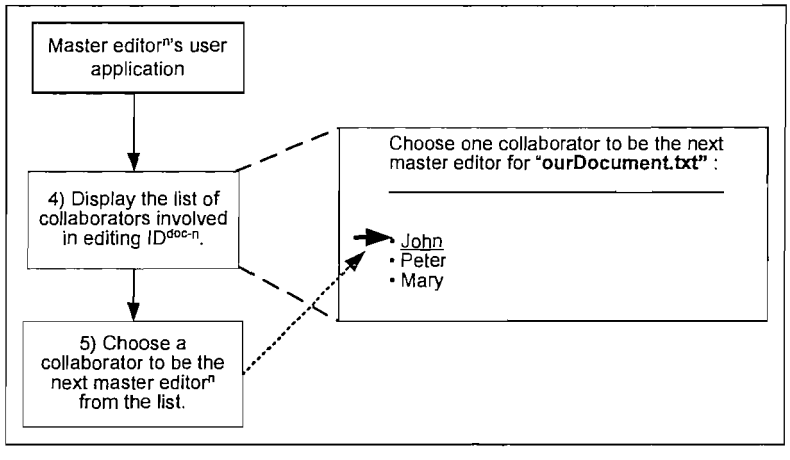


Figure 6.26: Master editor^n selects a collaborator to be the next master editor^n

On receipt of the "passEditingToken" request, John's user application asks John whether he want to accept the offer to be the next master editor for $ID^{doc\text{-}n}$, i.e. a file called "ourDocument.txt". Figure 6.27 illustrates that John accepts the offer to be the new master editor^n. An acknowledgement message is then sent to the previous master editor^n, which contains $ID^{doc\text{-}n}$ and an "accept" status. On receipt of this message, the user application of the previous master editor^n removes ET^n from all $ID^{doc\text{-}n}$'s details and marks all $ID^{doc\text{-}n}$'s documents as regular documents (cf. Figure 6.28). This is done through a change in the status of the user, who is no longer master editor^n. At the same time, on John's mobile terminal, the user application checks out the latest major version of $ID^{doc\text{-}n}$ from the repository. Once this has been done, John's user application marks the newly checked out document as the master document of $ID^{doc\text{-}n}$. This document can always be merged with John's previous regular document of $ID^{doc\text{-}n}$. Besides this, all the details of $ID^{doc\text{-}n}$ are also added with the editing token, ET^n.

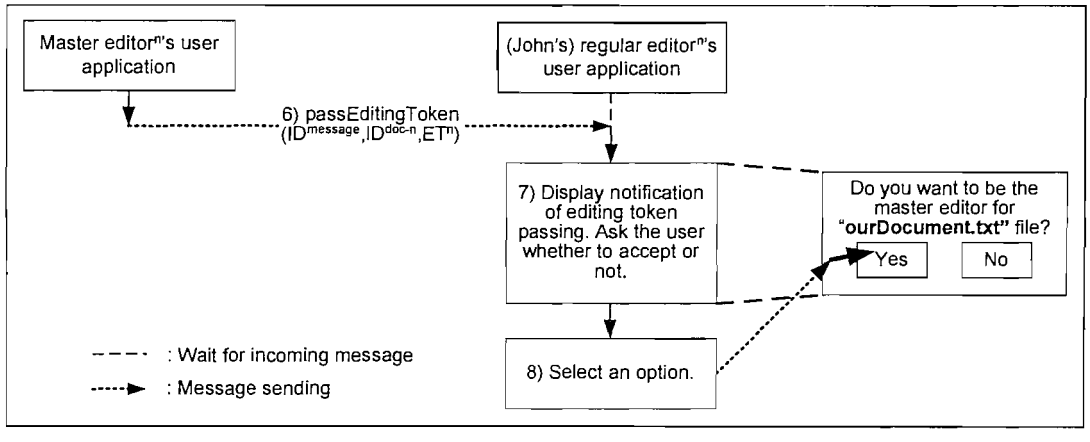


Figure 6.27: The chosen collaborator decides on whether to accept or reject the offer to be the next master editor for $ID^{doc\text{-}n}$

Referring back to Figure 6.27, in a case where John rejects the offer to be the new master editor^n, the "passEditingToken_ack" message returned to the previous master editor^n will contain a "reject" status. On receipt of such message, the previous master editor repeats all the steps in Section 6.5.4.9 but selects a different user (besides John) to be the new master editor^n.

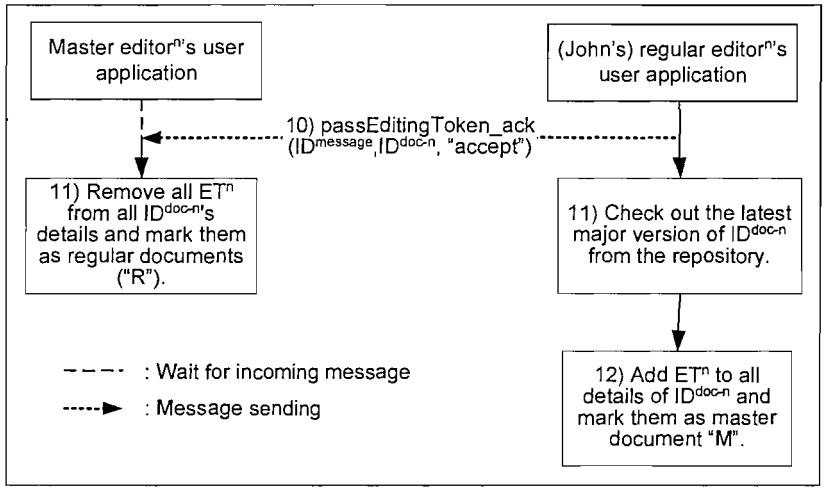


Figure 6.28: Updating status on both previous and new master editor^n's terminals

6.5.4.10. Version synchronising between repositories

Master editor^n may move to another location and at the new location, the user application will be connected to a different repository, i.e. repository2, which is the current locally available repository (cf. Figure 6.29). If this is the first time master editor^n has come to this location, no copies of ID$^{doc\text{-}n}$ could be stored in repository2. At this new location, master editor^n may decide to collaborate with locally connected users and want to make ID$^{doc\text{-}n}$'s copies available to them. In this case, master editor^n does not need to add ID$^{doc\text{-}n}$ again, as was done previously. In this case, the master editor just chooses the "synchroniseVersion" option from the main user application's menu and user application requests repository2 to get the latest major version of ID$^{doc\text{-}n}$ from another repository, i.e. repository1, which has the required version.

Figure 6.30 shows another way in which versions can be synchronised between two repositories, but this can be applied only by repositories that have ID$^{doc\text{-}n}$*'s* copies. In this case, synchronisation is automatically initiated by the master editor's user application. Once a newer major version of ID$^{doc\text{-}n}$ has been committed to a local repository, i.e. repository2, the user application retrieves from the repository directory the addresses of all the other repositories that store ID$^{doc\text{-}n}$'s copies. To each of these repositories, the user application sends a "synchroniseVersion" request that contains ID$^{doc\text{-}n}$, α^{rep2} and ET^n. The variable α^{rep2} represents the address of repository2 that has the latest major version of ID$^{doc\text{-}n}$. Figure 6.30 illustrates a remote repository, i.e. repository1, which

receives such a request. First of all, repository1 checks the validity of ET^n, after which it sends a "synchroniseVersion" message to repository2. Besides containing $ID^{doc\text{-}n}$ and ET^n, this message also includes V_x, which is the latest major version number of $ID^{doc\text{-}n}$ in repository1. On receipt of this message, repository2 first checks V_x against its latest major version of $ID^{doc\text{-}n}$, V_y. This is done to prevent repository1 from getting an outdated version from repository2, i.e. when $V_x < V_y$. If V_y is a later version than V_x ($V_y > V_x$), then repository2 returns a copy of V_y to repository1 together with an acknowledgement message. On receipt of this message and the document's copy, repository1 stores the received copy and its details.

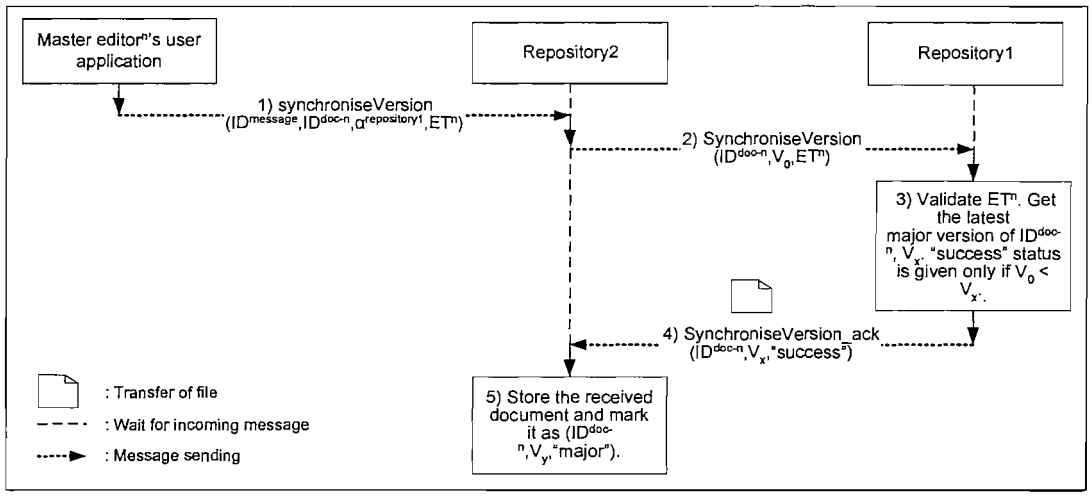


Figure 6.29: Add and synchronise major versions of $ID^{doc\text{-}n}$ on the new local repository.

New version numbers can be assigned by different repositories to newly committed versions, but each version will be unique across repositories. This is ensured since every time a master editor commits the modified master document, its current version number, i.e. V_x, is included in the commit request, thus the receiving repository (no matter which one), assigns the next version number of the committed copy as V_{x+1}. This also applies to a repository that does not have any previous version of the master document, where instead of adding the document as a new major version (V_1), the document is assigned as V_{x+1}.

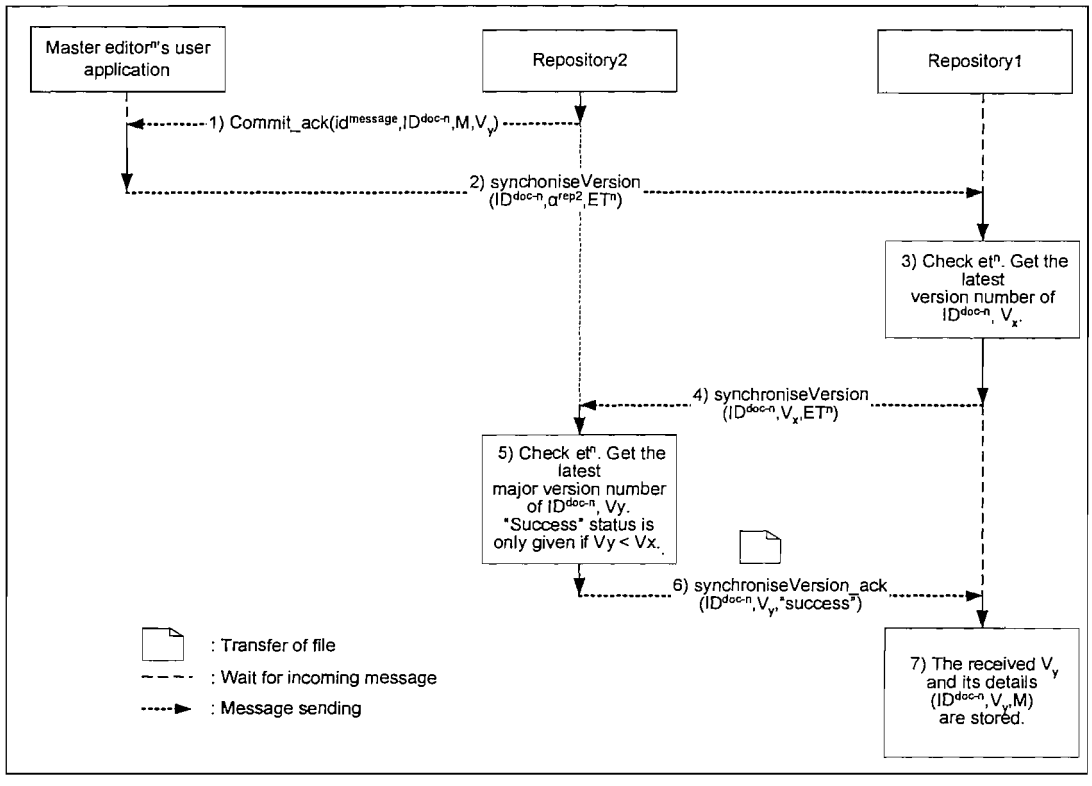


Figure 6.30: New major version of $ID^{doc\text{-}n}$ is propagated to a remote repository

6.6 Implementation

In this section we describe how we implement the collaborative editing application. The application can be developed to operate on its own, even though we are in favour of employing MAMiMoU to support the interactions between the application components. For the sake of clarity, we describe two approaches to implementing the application, i.e. with and without MAMiMoU's support. We will first describe the implementation of an application not supported by the middleware, followed by a description of the application's integration with MAMiMoU.

We have developed our application using the Southampton Framework For Agent Research (SoFAR) [72]. The collaboration protocol of the application is implemented by two agents, namely a user-application agent and a repository agent (cf. Figure 6.31). We host the user-application agent on the mobile terminal, while the repository agent is hosted on stationary terminals. As there is no mechanism used by the application to support the asynchronous delivery

of notifications, the user-application supports an additional function for mobile users: to check for incoming notifications about a particular document from the repository. In this case, notifications for users are held by the repositories until they are collected by the recipients. Such repositories are ones that have the versions being modified by the users awaiting review from the master editor. The disadvantage of implementing such an approach is that the notifications will be received by the users only when requested, and the repository that is being requested can be remotely located and may not be accessible.

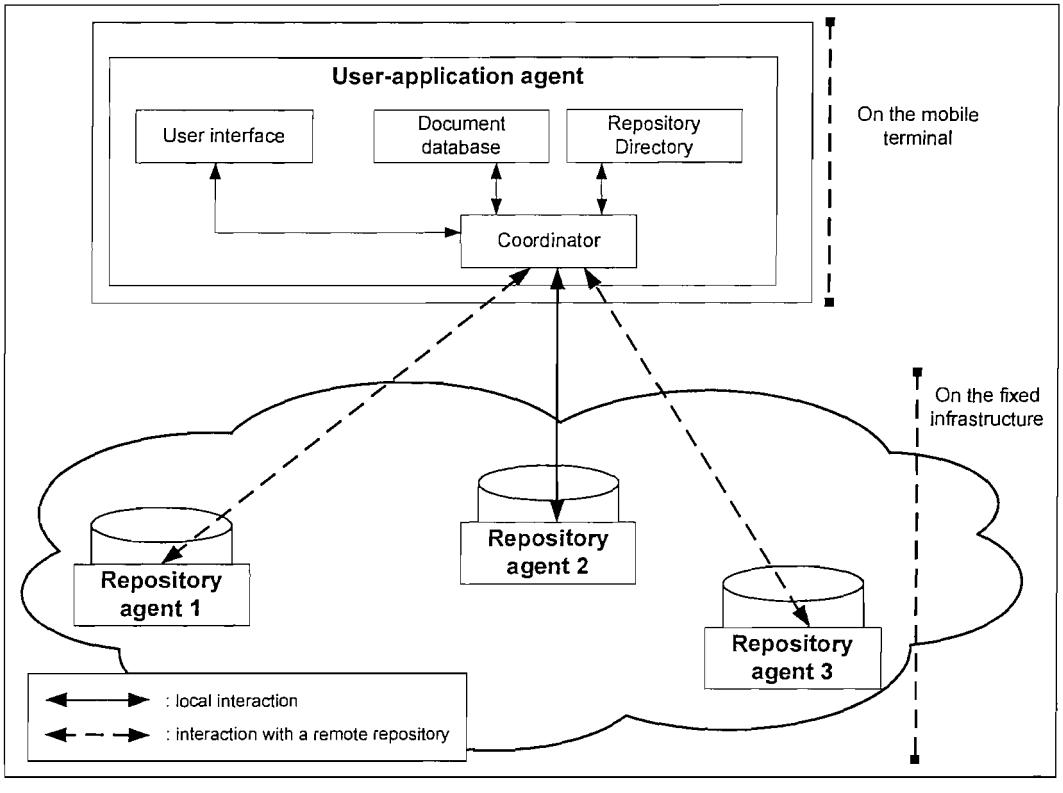


Figure 6.31: Collaborative editing application

To take full advantage of MAMiMoU, we offloaded some of the collaborative editing application's components onto MAMiMoU. This is illustrated in Figure 6.32. The coordinator is now divided into two components, namely coordinator1 and coordinator2. Coordinator2 and the repository directory are the two application components offloaded onto the fixed network as part of the shadow. When a request is received from a user, coordinator1 creates a request message by querying the document database. The encoded message is then forwarded to coordinator2, which forwards it to a local repository after making a query for the repository's address from the

repository directory. An acknowledgement message from the repository is then returned to coordinator1, which forwards the message to the user interface. By offloading these components to the shadow, less computation is performed on the mobile terminal. Besides this, some activities can now be performed when the mobile terminal is disconnected. For example, coordinator2 can still query repository directory and send an unsent request to the repository, and receive an acknowledgement.

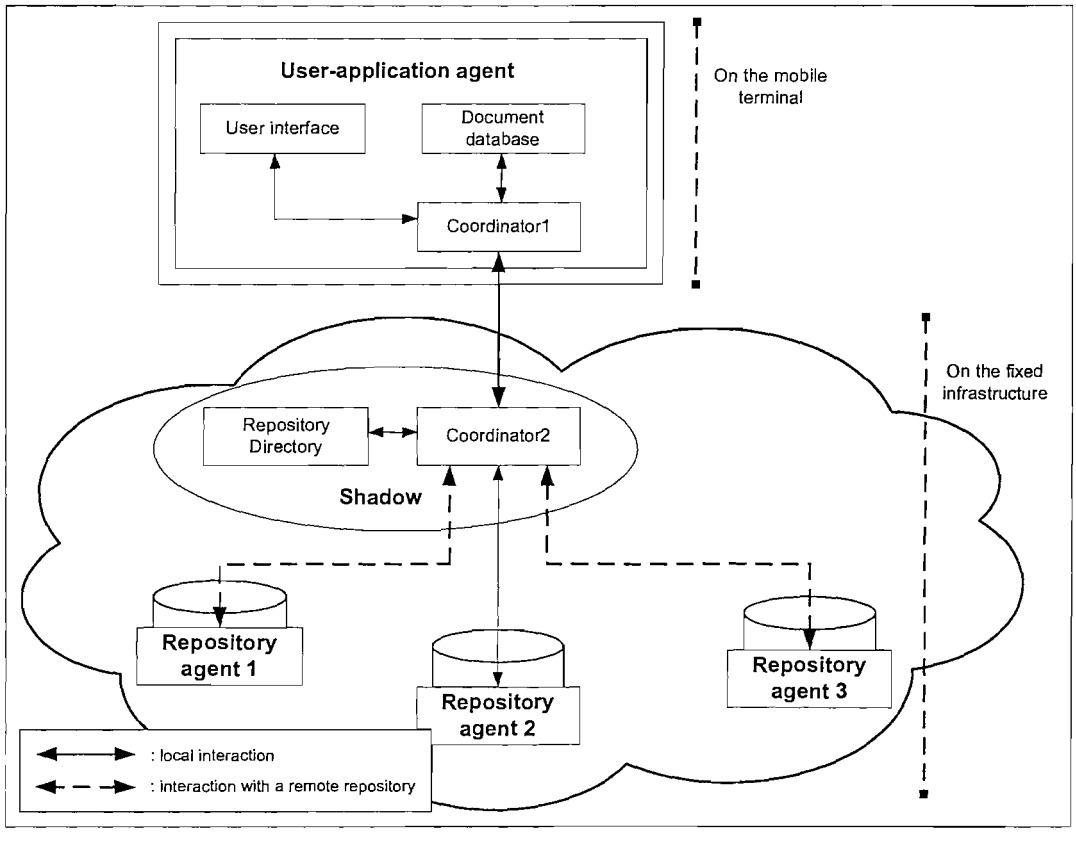


Figure 6.32: Collaborative editing application supported by MAMiMoU.

The system now consists of three agents: a user-application agent; a shadow; and a repository agent (cf. Figure 6.32). The shadow implements the coordinator2 and repository-directory components of the application, while the user-application agent implements the user-interface, document directory and coordinator1. The shadow is now used to support asynchronous delivery of notifications, where it will always be available on the network to receive notifications from the repositories on behalf of the user. By relying on the shadow to do this, notifications will be forwarded to the user once the user re-connects to the network. Additionally, having coordinator2

and repository-directory offloaded onto the fixed network allows them to continue their tasks, e.g. to serve pending requests from the user while the mobile terminal is disconnected. This way, time can be saved since no pending requests are lost during the disconnection of users, which may require the user to re-send the requests again. Having said this, we will analyze further the benefits gained from this integration by means of an empirical evaluation presented in Chapter 7.

6.7 Security

To address the issue of security, cryptography [30], which is known as a major enabling technology for network security, can be used to establish secured channels between the application's components, both in terms of authenticity and integrity, as well as confidentiality. To do this, an additional layer dealing with security can be implemented on top of the collaboration protocol layer. This layer will implement a cryptographic algorithm, i.e. an algorithm defined by a sequence of steps precisely specifying the actions required to achieve a specific security objective, and a cryptographic protocol, i.e. a distributed algorithm defined by a sequence of steps precisely specifying the actions required from two or more entities to achieve a specific security objective [83].

One potential cryptosystem[3] that can be used is the public key cryptography, in which a user has a pair of mathematically-related keys — a public and private key, where it is computationally infeasible for an outsider to derive one from the other. Public key cryptography can be used to protect the confidentiality of data transferred on the network. Besides protecting the confidentiality of messages exchanged between components, a public key cryptosystem can also be used to protect the authenticity and integrity of a message. This way, the authenticity and integrity of objects included in the message, e.g. an editing token or a file, can indirectly be protected. For example, a sender A can protect the authenticity and integrity of a message M by computing a digital signature S for M. Digital signatures provide an electronic analog of handwritten signatures for electronic documents, where the signature is not forgeable, can be verified by the recipient [83].

[3] A cryptosystem is the package of all procedures, protocols, cryptographic algorithms and instructions used for encoding and decoding messages using cryptography.

Another way of protecting editing tokens against unauthorized reading and undetected mutilation is by having editing tokens stored in an encrypted form in the repository. User applications and the repository both use the same encryption and decryption algorithm, i.e. [90]. This way both user application and repository are able to encrypt or decrypt editing tokens with any secret cryptographic function. In supporting this technique, an approach presented in [107] can be used, where the editing token is encrypted and decrypted as one partition of the shadow's state. This way, the symmetric key used in the encryption can be made available to platforms being visited by the shadow. To control such propagation of keys, mobile code called a keylet is employed.

6.8 Discussion

In this section we summarize the advantages offered by our application. First of all, our application's architecture promotes high accessibility of mobile documents to users. The application allows other users to check out a document with authorization given by the document's owner. This is possible even if the owner is not online. To achieve this, we have taken the approach proposed in [111], which suggests that one way to maximize the availability of mobile documents is by having their replicas stored on the fixed network.

Document replicas will be kept on repositories located on the fixed infrastructure. A repository is a storage system, in which an aggregation of documents are kept and maintained in an organized way. It allows operations on documents such as add, commit, check out, update and remove to be performed [12]. A local repository is a repository that runs in the user's vicinity: one that runs on the same local network as the user. It advertises its services on a local lookup table, e.g. [81], from which a mobile user discovers it. To take full advantage of distributed repositories available on the fixed network, a document replica may be stored not only on one repository but also on as many repositories as necessary. A replica of the same document can be added to another repository every time a user changes their location. The more replicas available on the network means the easier it is for other users to access the document.

The main advantage offered by our collaborative editing application in supporting collaboration between mobile users is its collaboration protocol. This protocol can be regarded as a service promoting collaborative activities between users, allowing it to be re-used in other types of

settings, e.g. supporting an application with a different architecture or implemented with a different technology.

Having users simultaneously manipulate a document's replicas raises the issue of consistency, between the original document and its replicas. To resolve this, we have adopted the metaphor of an editing token. The user holding an editing token has the sole permission to edit a specific document. We call such user the master editor, while the original document is referred to as the master document. We promote flexibility by allowing other users to work on their copy of the document in parallel. We call these users regular editors as changes they made to their copy of document need to be approved by the master editor before those changes are merged with the master document.

Changes committed by a regular editor to the repository may not be reviewed immediately by the master editor. This may due to the unavailability of the master editor on the network, or the master editor is not ready to review the changes. With the support from the collaboration protocol, such requests will be queued on the network and can be viewed by the master editor once requested. This application will support weak consistency of documents, where document copies on the network are allowed to diverge temporarily, but they will eventually come to an agreement by means of the master editor's approval. Document copies that have the same version as the master document are considered up to date and consistent with the master document. Other versions are either waiting to be reviewed by the master editor, or already outdated. Our application helps a master editor to have easy access to all versions committed by other users. Even though user-committed versions of a document are stored by different repositories across the network, our application is able to pull information about these versions from the repositories. Once obtained, this information will be presented to the master editor.

The protocol also promotes eager notification of a document's new version across repositories that have a copy of the document. When a master editor successfully commits a new version of a document to a local repository, our application is able to extend information on this new document version to other repositories that currently have only the outdated versions of the document. This information also includes details of a repository keeping the latest version of the document. With such information, other repositories could synchronise their outdated versions with the latest version from the specified repository.

A version control system such as Concurrent Version System (CVS) [12] can be used to support repository and document versioning. CVS is a version control system aimed at keeping document history and has all versions of a document by storing differences between them. In this case, an application, e.g. a repository agent can be employed to act as a wrapper to the CVS repository, while implementing our protocol described for a repository. This way the repository agent will translate all messages received from user applications to CVS commands executed on the repository. Results acquired from the repository are returned to the requesting user applications.

In comparison to other related works [135][94][95], our work is unique in the sense that we use mobile elements, i.e. shadows of MAMiMoU, to support the document sharing and collaborative application for mobile users. Our approach is similar to others in the sense of having a proxy or an application do tasks on behalf of a user on the infrastructure, although our approach offers more flexibility with a shadow: a state associated with a user that is mobile and capable of migrating to the user's vicinity. Repositories that are exploited by the shadow in this application are those that are also local to the user's device. The short-distance communication between the mobile device, shadow and repositories on the fixed infrastructure helps reduce the bandwidth required for the application and improve its performance [34]. To summarize from the implementation's viewpoint, relying on MAMiMoU to handle complex interactions between mobile device and repositories has made the application's development straightforward. It allows us to concentrate on the application-specific details of the application.

6.9 Conclusion

We have presented a collaborative editing application supporting mobile document sharing and editing between mobile users by describing its architecture and collaboration protocol. Even though the application is able to operate on its own, we relied on MAMiMoU to store and forward messages transparently to mobile users. This is to improve the application's performance, as some collaborative activities, such as delivery of notifications and propagation of knowledge between repositories, can still be continued while users are offline. To prove this hypothesis, we will present our evaluation in the next chapter, which aims to compare the performance of the collaborative editing application when it is supported by MAMiMoU and when it is not.

Chapter 7

Evaluation

In the previous chapter, we introduced a collaborative editing application, with an architecture that promotes high accessibility of mobile documents among mobile users, supported by a collaboration protocol designed to support collaborative editing between mobile users. Based on services offered by MAMiMoU, which include supporting asynchronous messaging and hosting an offloaded application's components, we believe that MAMiMoU's support can improve the application's performance. To prove this claim, we perform an evaluation to compare the performance of the application both when it is supported and when it is not supported by MAMiMoU.

7.1 Evaluation Overview

In this chapter we present an empirical evaluation to analyse the benefits gained from the integration of the collaborative editing application with MAMiMoU. Our aim is to establish whether MAMiMoU significantly improves the collaborative editing application's performance. In the evaluation, we test the real implementation of MAMiMoU and the collaborative editing application in simulated environments. To this end, we compare the application's behaviour in two different settings, specifically with and without MAMiMoU's support. In both settings, the application is programmed to perform the same sequence of collaborative editing processes, which we refer to in the rest of the chapter as a *collaborative editing task*.

We have designed an environment that simulates users performing typical collaborative editing tasks. During collaboration, the simulated users are made to experience unavailability of connection, being connected via different bandwidth, or being made to move to other locations, resulting in delays before they can continue with their processes. Even though the evaluation is performed in simulated environments without real users, the analysis is sufficient since the aim of

the evaluation is not to evaluate interactions between users, but to evaluate the system being used to support collaboration between mobile users in such environments.

7.1.1 Simulation Methodology

In this section we present the approach we adopt to conduct the simulation. The simulated environment can be described in terms of four different dimensions, each consisting of different levels (cf. Figure 7.1). These dimensions and their categorizations are described next.

Discrete Dimensions	Simulations
i) MAMiMoU's support	o With MAMiMoU's support (SoFAR and MAMiMoU) o Without MAMiMoU's support (SoFAR)
ii) Collaboration patterns	o Serialized editing o Simultaneous editing o Random editing
iii) Users' connectivity	o Always connected users (ACU) o Moderately connected users (MCU) o Rarely connected users (RCU)
iv) Different features in mobile users' environments	o Users' connection speed o Users' mobility o Collaboration group size o Shared file size

Figure 7.1: Discrete dimensions

7.1.1.1 Dimension 1: MAMiMoU's support

At any one time, the first dimension represents one of two possible settings: the collaborative editing application is either supported by MAMiMoU or it is not. The application's implementation in both settings is described in Section 7.7.

7.1.1.2 Dimension 2: collaboration patterns

There are infinite possibilities for how a group of people can collaborate in editing a document. There are different ways of performing collaborative editing, or *collaboration patterns,* which can

be considered as an aspect of dependencies between the work of the collaborators [82]. The collaboration patterns influence the time required to complete a collaborative editing task, since different collaboration patterns may involve a different number of activities and different order of performing the activities. Thus, some collaboration patterns may save time, in which case the collaborative task among a group of mobile collaborators can be performed faster. As a result, we include this feature as the second dimension that parameterizes the evaluation environment. As we cannot possibly simulate an infinite number of different possible patterns of collaboration, we narrow them down to three types of collaboration patterns, which we have named *serialized*, *simultaneous* and *random editing*. Serialized editing collaboration requires most activities to be performed in series; simultaneous editing collaboration promotes higher degree of parallelism; while random editing collaboration allows the mixture of the other two patterns. We choose these patterns due to the difference in number and order of activities they may offer. Thus, with these variations, we expect to see a variation in the time needed to complete the collaborative tasks in the simulation.

To give a general view of the collaboration pattern simulation, here we describe the general idea of what will happen during the simulated collaboration. In the simulation, a group of between four and sixteen users collaborate in editing a document. Each user is responsible for editing a section of the document, and thus no update conflict is expected to occur. The sections written by regular editors will be committed as tentative versions in the repository. Accordingly, the master editor will have to merge these sections, i.e. the accepted tentative versions, into the latest version of the master document.

In the *serialized editing* collaboration pattern (cf. Figure 7.2), the editing token is passed between the editors sequentially. In this case, each editor has a turn to become the master editor. An editor who receives the editing token becomes the new master editor, who is entitled to merge changes made to the latest version of the master document and commit it to a repository as a newer version of the master document.

In the *simultaneous editing* collaboration (cf. Figure 7.3), only one editor is responsible for being the master editor all the time; this editor is responsible for reviewing and approving the changes committed by other editors as tentative versions. This collaboration is simultaneous in the sense that all regular editors can concurrently make changes to the checked-out document, commit

changes as tentative versions and then leave the decision to review and accept or reject the changes to the master editor.

Finally, in the *random editing collaboration* (cf. Figure 7.4), the master editor will either randomly pass the editing token to a potential candidate, or review a tentative version committed by another user. This type of collaboration is really flexible; in which case it can mix both serialized and simultaneous editing to various degree, e.g. simultaneous editing is adopted most of the time compared to serialized editing or vice versa. Thus, in the collaboration, only some editors may get the opportunity to become master editor, unlike serialized editing. On the other hand, there may be more than one master editor throughout the collaboration process, unlike simultaneous editing. In simulating this pattern we consider an optimistic approach, which is based on the following rules:

1) When an editor (editor-A) is assigned as a master editor:
    a. editor-A is not allowed to be a master editor again once he / she passes the token to another user
    b. the previously submitted tentative version by editor-A will not be reviewed again, as the section updated by editor-A is already merged with the master document
2) When the tentative version committed by a regular-editor (editor-B) has been reviewed by the master editor:
    a. editor-B will not be a master editor
3) When all sections have been reviewed, the final version of the master document can be committed to the repository, allowing other regular editors to check out the latest version.

In Figure 7.4 we denote a list of editors entitled to be the next master editor as potentialME-list, while review-list is a list of tentative versions that need to be reviewed in order to complete the collaborative editing task.

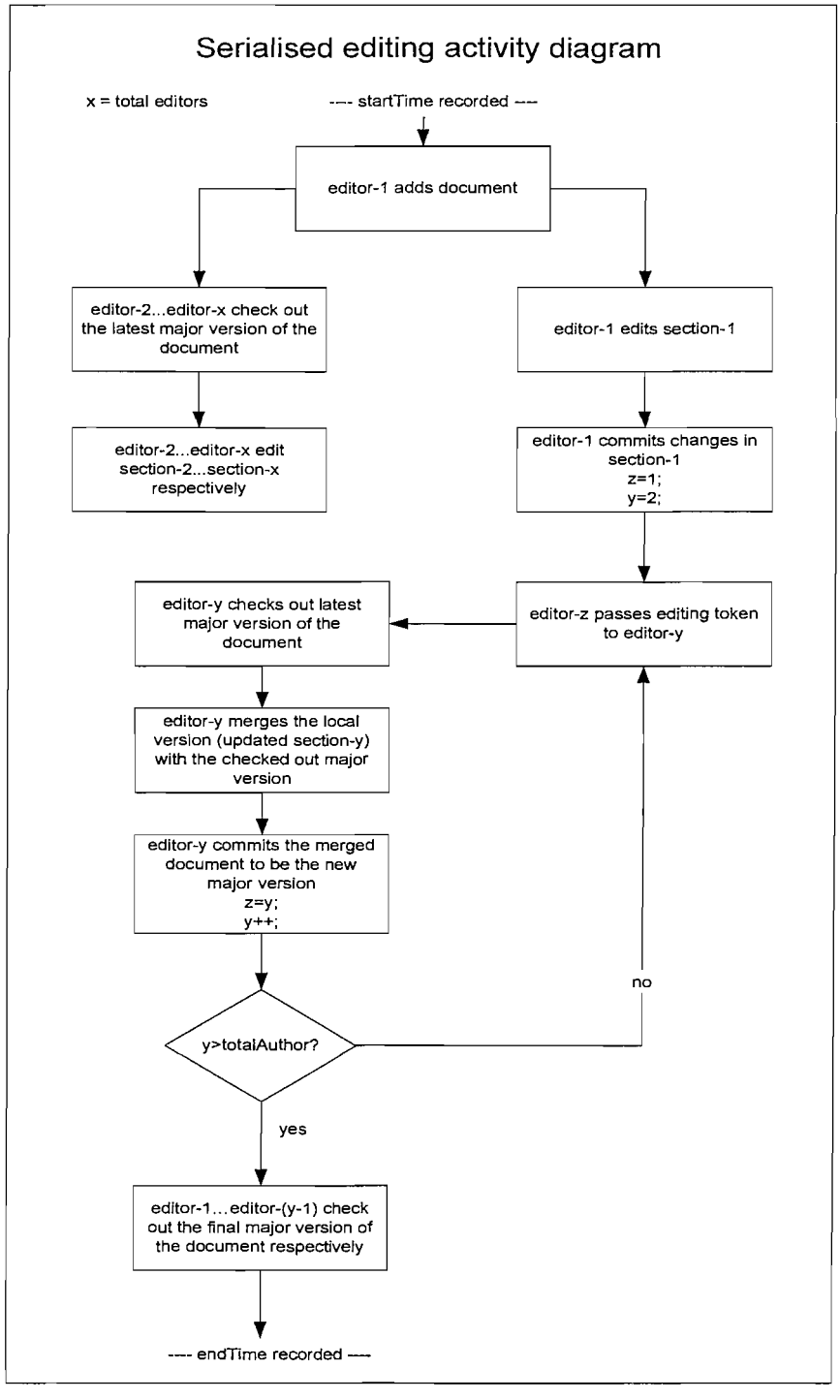


Figure 7.2: Serialized editing collaboration

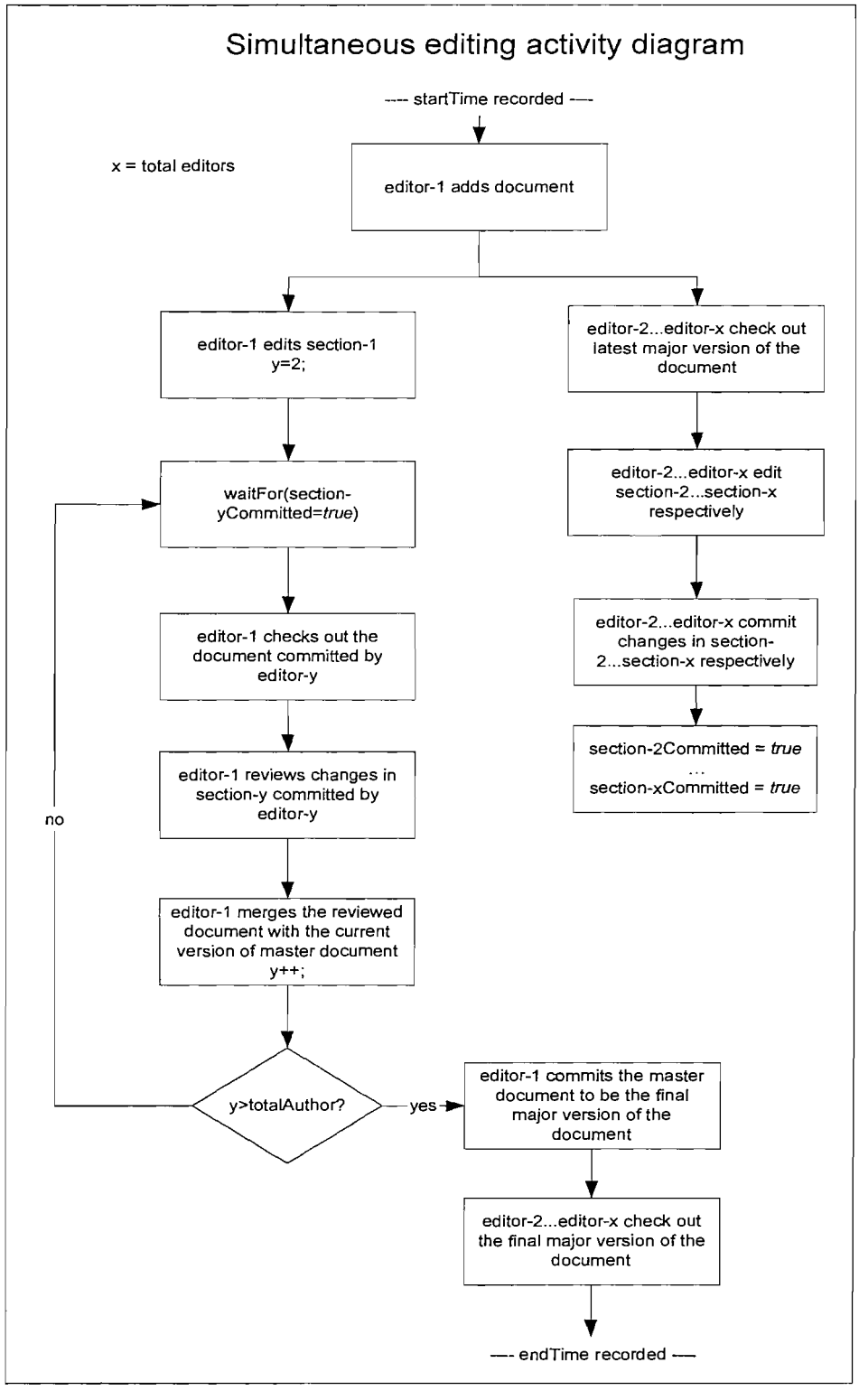


Figure 7.3: Simultaneous editing collaboration

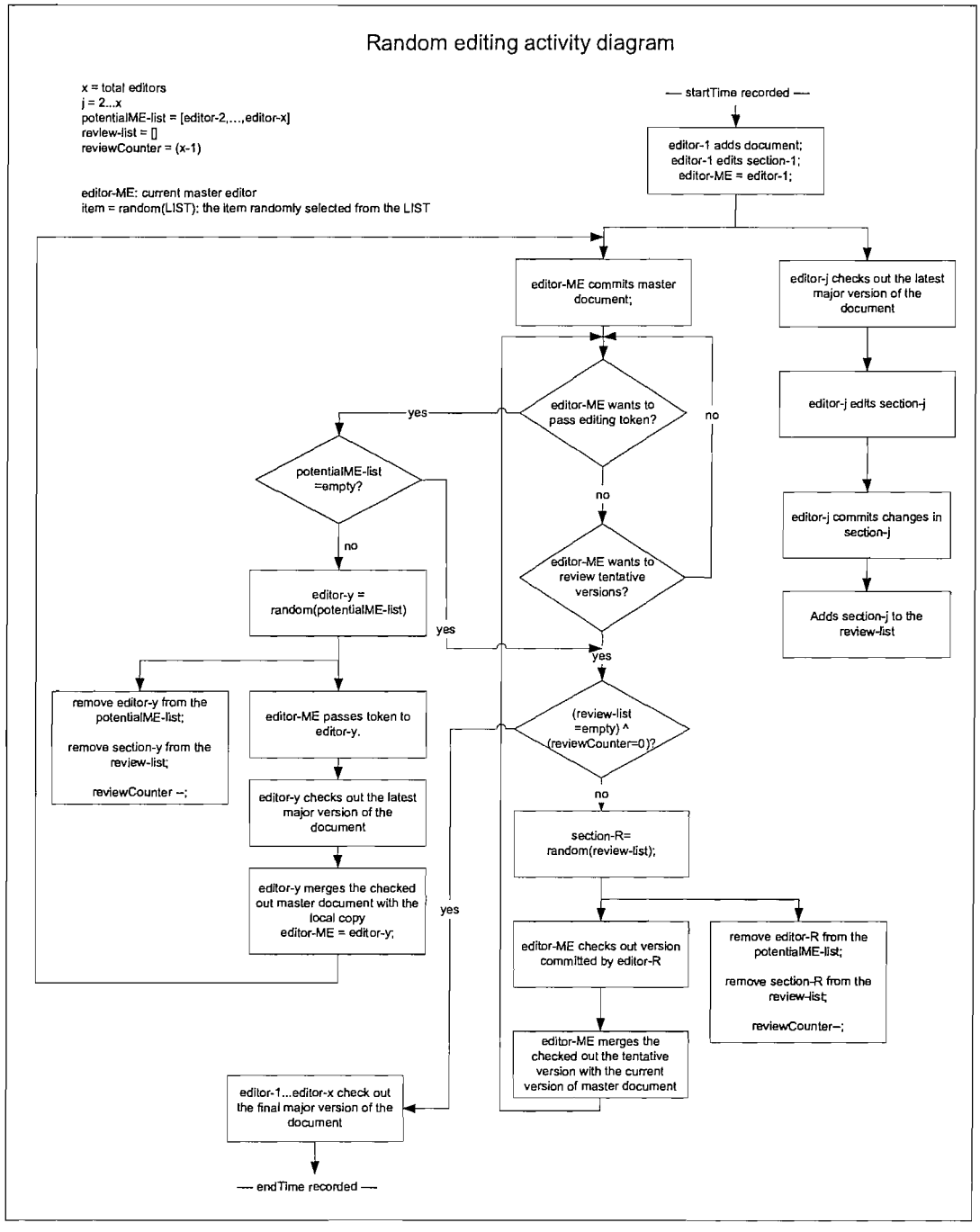


Figure 7.4: Random editing collaboration

In order to present our hypothesis based on this dimension, we have analysed the expected behaviour of each collaboration pattern. First, we calculate the number of activities needed by

each collaboration pattern to accomplish an editing task, followed by the required number of connectivity.

Collaboration patterns / activity	**Serialized editing**	**Simultaneous editing**	**Random editing**	
Add document	1	1	1	
Check out document	3*(totalEditor - 1)	3*(totalEditor - 1)	3*(totalEditor -1)	
Commit document	totalEditor	totalEditor	totalEditor	
Pass editing token	totalEditor - 1	0	totalEditor - 1 (max)	0 (min)
Merge document	totalEditor – 1	totalEditor – 1	totalEditor – 1	totalEditor – 1
Review document	0	totalEditor – 1	0	totalEditor – 1

Figure 7.5: Formulas to obtain the number of every activity performed in each collaboration pattern (derived from Figure 7.2 – 7.4)

With these formulas (cf. Figure 7.5), we can obtain the number of each type of activity and estimate which of these collaboration patterns will spend more time accomplishing the editing task. This is done by obtaining the total number of activities involved when a pattern is adopted. If we assume that each occurrence of an activity requires the same amount of time as others, we may estimate that the collaboration pattern with the most number of activities will require more time than if other patterns are adopted.

Based on this figure, we estimate that the collaborative editing task will be completed about the same length of time for any collaboration pattern being adopted. In performing the task, some activities may be carried out in parallel by different users, which we believe will affect the task completion time. However, this analysis is made just to get a rough idea of the outcome of the simulations.

7.1.1.3. Dimension 3: users' connectivity

The main difference between a collaborative editing application with and without MAMiMoU's support is the total time needed by the application to be connected in order to proceed with the editing task. When supported by MAMiMoU, the mobile collaborators do not have to be continuously connected to the fixed infrastructure while collaborating. The only time connectivity

is needed is when the application needs to deliver messages or requests to the repository, and to receive the results. From this perspective, we can say that the use of MAMiMoU can minimise the connectivity required for performing tasks. Thus we expect that the application with its support would have the advantage in an environment where connectivity to the network is infrequent.

We consider three different environments in which users may operate:

i. Always connected users (ACU) environment
ii. Moderately connected users (MCU) environment
iii. Rarely connected users (RCU) environment

In the ACU environment, all users are always connected to the network, while in the MCU environment each user is connected only 50% of the time. In this environment, users' connections and disconnections are uniformly distributed throughout the simulations. We simulate this by activating and deactivating the online status of the device, where each activation and deactivation is performed 50% of the time. In the RCU environment, each user is connected to the network for only 10% of the time. As in the MCU environment, the activation and deactivation of online status is generated randomly, summing up to the ratio of 10% online activation time and 90% online deactivation time.

We expect that a group of always connected users will always complete tasks faster than moderately and rarely connected users. This is due to the continuous connectivity that allows users to perform an uninterrupted collaborative editing task. A collaborative editing task requires messages to be exchanged between collaborators' mobile terminals and the repository. Once a mobile terminal is disconnected, no messages can be received or sent by the mobile terminal. This slows down the completion of the collaborative task. While in contrast, if all users are always connected, the collaboration can be carried out continuously, as messages can always be exchanged between the repository and the mobile terminals. This speeds up the completion of the task.

Connectivity to the fixed network is needed when the device is sending a message to the repository or expecting a reply from the repository. At this time, the absence of connectivity will result in a delay in proceeding with the task since the mobile terminal needs to wait for reconnection. Apart from such a situation, the absence of connectivity does not have much effect.

Based on this hypothesis, we first estimate the respective ratio of connectivity required by each collaboration pattern to complete a collaborative editing task.

Specifically, most activities require connectivity when the request is first sent by the mobile terminal to the repository, and later when the reply is returned by the repository. If the mobile terminal is disconnected between these two times, the result will be lost and not be returned to the user application. But with the support provided by MAMiMoU, the shadow will always be available to receive the result. Thus, any disconnections will not cause a significant effect. For activities like "pass editing token" and "check out document", connectivity is required more than twice. For each activity, we estimate the number of messages being exchanged between different components, reflecting the need for connectivity. With this summary, we may estimate the significance of connection presence for each activity. To evaluate the effect of connectivity on the application's performance, we simulate three degrees of users' connectivity.

As mentioned earlier, different patterns of collaboration involve different number of activities. At the same time, different activities will have different sensitivities to the level of connectivity. Thus, from the number of each type of activity, we may estimate the relative ratios of connectivity required by the different collaboration patterns.

7.1.1.4. Dimension 4: features in mobile users' environments

The fourth dimension is used to express miscellaneous features of mobile users' environments, such as connection speed, users' mobility and different size of collaborating group. These features and their different discrete values are listed in Figure 7.6. In the user's connection speed simulation, users are made to experience different connection bandwidths; in the user's mobility simulation, users experience different degrees of mobility; in the collaboration group size simulation, different numbers of users are simulated to perform collaborative tasks; and finally in the document size simulation, users are made to edit different sizes of shared document. These features will be explained further in Sections 7.2.1 to 7.2.4. In each section, we explain the hypothesis, followed by the results obtained and the analysis we draw from the results.

Features in mobile users' environments	Simulations
Users' connection speed (cf. Section 7.2.1)	• Slow (14.4 Kbps) • Medium (56.6 Kbps) • High speed (100 Mbps)
Users' mobility (cf. Section 7.2.2)	• Stationary users • Moderately mobile users (users change location every 10 minutes) • Highly mobile users (users change location every 5 minutes)
Collaboration group size (cf. Section 7.2.3)	• Small group (4 editors) • Medium size group (8 editors) • Large group (16 editors)
Document size (cf. Section 7.2.4)	• Small file (4KB) • Medium size file (16KB) • Large file (77KB)

Figure 7.6: Dimension 4: features in mobile users' environments

7.1.2 Computing Facilities

In order to run the evaluation, we use Southampton University's Iridis cluster, consisting of 204 computational nodes. The service is primarily designed as a batch service for users who need to run distributed memory parallel jobs. Nodes are accessed via the Easy scheduler, to which jobs are submitted. Most Iridis nodes have CPU type of Intel PIII or P4 and have at least 256MB available memory (RAM). With this facility, we can run the evaluation on 7-19 nodes at one time. Seven nodes are required to run a simulation involving four mobile collaborators; eleven nodes are required to run a simulation involving eight mobile collaborators and nineteen nodes are required to run a simulation involving sixteen mobile collaborators. We will describe the evaluation settings next.

7.1.3 Evaluation Environment Settings

We set up the evaluation environment by running a repository on one node and user applications on another 4 to 16 nodes, each of which represents a user's mobile terminal (cf. Figure 7.7). In

order to simulate user migration (which affects the location of shadows in MAMiMoU), a shadow handler is deployed on another node to create and migrate shadows as if a user had changed location. We also run a recording agent on one stationary node, which records the events that are logged by the application.

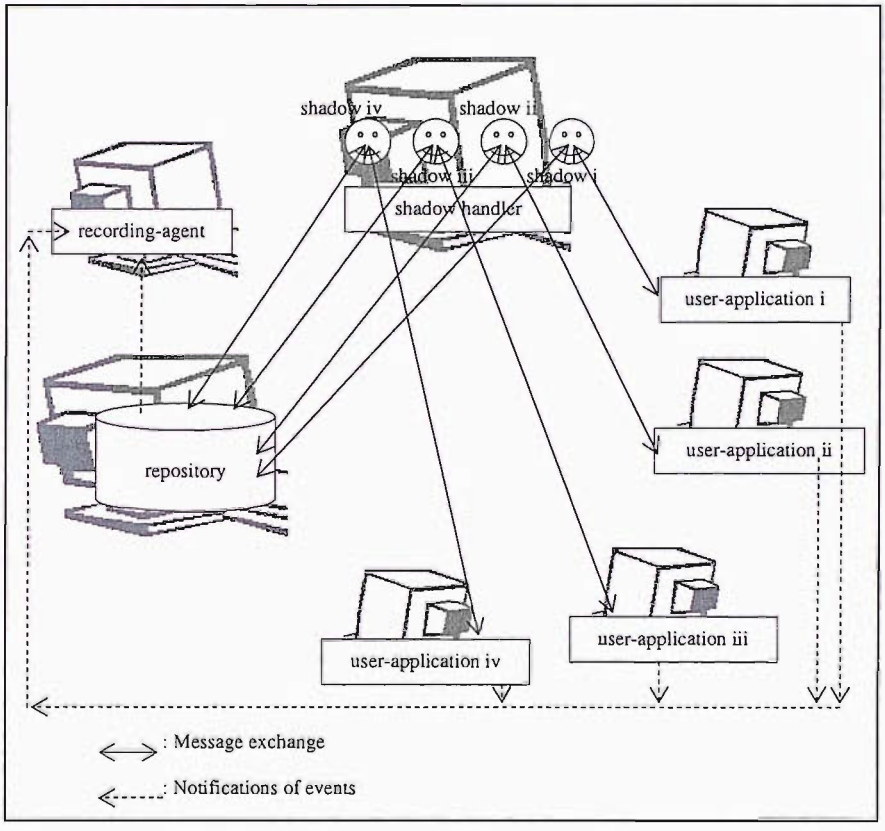


Figure 7.7: Evaluation environment settings

The metric we use to assess application performance is task completion time, which is the total time spent by the application to complete a simulated collaborative editing task among a group of virtual users. The less time required by an application to complete the collaborative editing task, the higher performance we claim for the application. Task completion time is actually measured from the log recorded by the recording agent. In the simulation, the application logs a *start-time* and an *end-time*. Start-time is the time when Editor1, who is the initial master editor, sends the "add" request to the repository, while end-time is the time when the last editor, i.e. Editor4, has checked out the final version of the document (cf. Figure 7.8). Both start-time and end-time are

recorded by the recording agent when notifications of such events are received from the respective users' user applications.

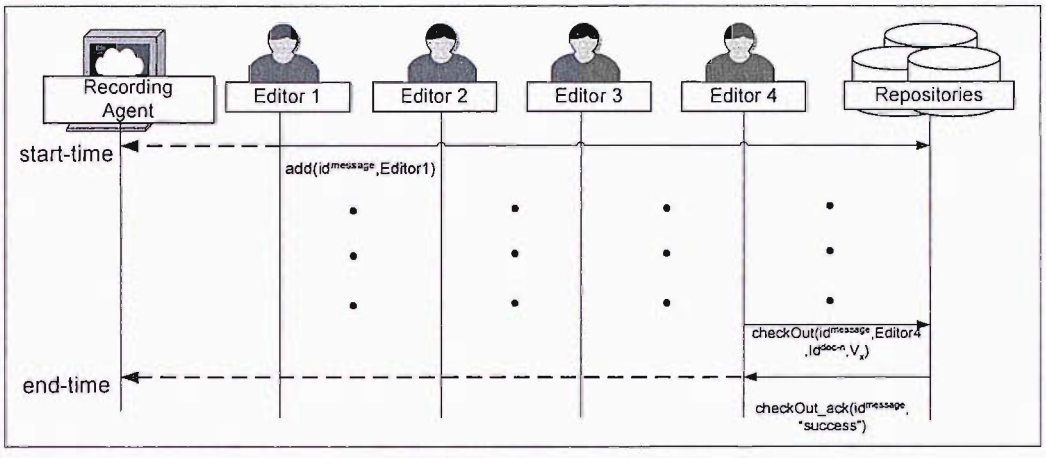


Figure 7.8: Simulation of a collaborative editing task between mobile collaborators

7.2 The evaluation

In this section, we describe the four evaluations we perform to analyse the application's performance with and without MAMiMoU's support.

7.2.1 Evaluation based on users' connectivity

The first evaluation is performed to observe the performance of the collaborative editing application in different settings of users' connectivity. Our aim is to establish whether a collaborative editing task is completed faster by a collaborative editing application that is supported by MAMiMoU, than one without it, in the RCU environment. Based on Figure 7.6 and 7.9, we expect that a group of mobile collaborators adopting the serialized editing collaboration will take the longest time in completing the collaborative editing task in comparison with the other two patterns of collaboration. The reason for this is that it has the highest number of messages passed, i.e. 70, in comparison to the total of 46 and 18 messages required by the simultaneous and random editing.

Activity	Number of connection needed
Add document	• Sending add message: 1 • Receiving add_ack: 1 • **Total: 2**
Check out document	• Sending log message: 1 • Receiving log_ack message: 1 • Sending check out message: 1 • Receiving checkout_ack message: 1 • **Total: 4**
Commit document	• Sending commit message: 1 • Receiving commit_ack message: 1 • **Total: 2**
Pass editing token	Current master editor: • Sending get collaborator list message: 1 • Receiving get collaborator list_ack message: 1 • Sending pass editing token message: 1 • Receiving pass editing token _ack message: 1 • **Total: 4** New master editor: • Receiving pass editing token message: 1 • Sending pass editing token ack message: 1 • Sending change master editor : 1 • Receiving change master editor _ack : 1 • **Total: 4**
Merge document	• **Total: 0**
Review document	• **Total: 0**

Figure 7.9: Number of message transfer needed for each activity

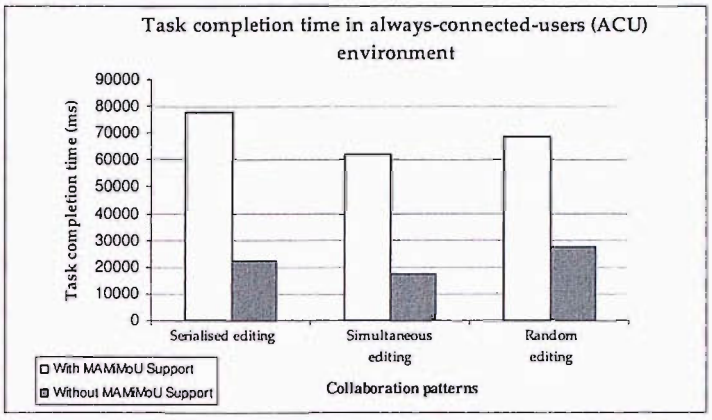


Figure 7.10: Task completion time in ACU environment

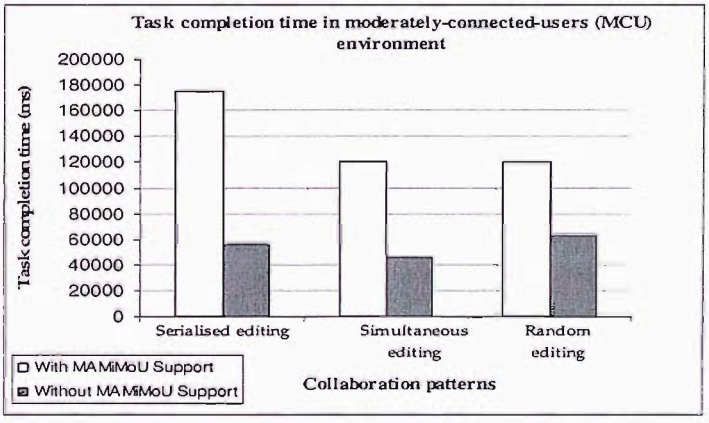


Figure 7.11: Task completion time in MCU environment

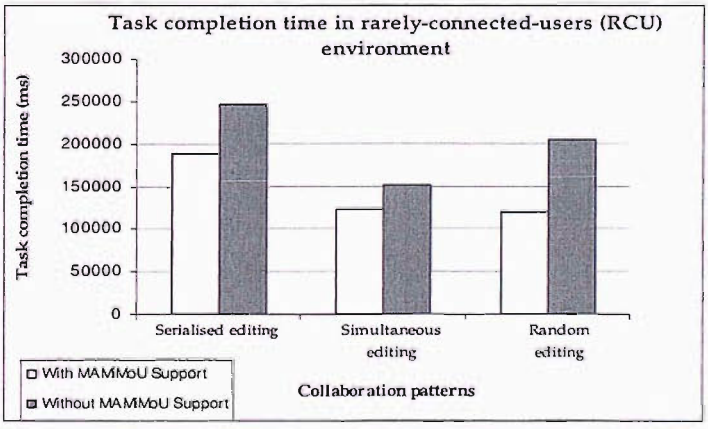


Figure 7.12: Task completion time in RCU environment

We have performed a total of 18 experiments for this evaluation. For each of them, we run 10 simulations of the collaborative editing task, which we average and plot in Figures 7.10 to 7.12. The most noticeable observation is that in the ACU and MCU environments, the application without MAMiMoU's support performs the task faster than the application supported by MAMiMoU, but in the RCU environment, an application supported by MAMiMoU outperforms the one without MAMiMoU.

Comparing results from MCU and RCU, on average, the performance of the application without MAMiMoU's support degraded as much as 266% in RCU, in comparison to only 4% performance degradation of the application supported by MAMiMoU. From this, we may say that the application supported by MAMiMoU performs more steadily in MCU and RCU, which means that the application is not affected much by additional users' disconnections. The calculation of these percentages is shown in Figure 7.14.a.

Referring to Figure 7.12, we calculate completion time difference by applications with and without MAMiMoU's support, expressed as percentages. These calculations are presented in Figure 7.13. On average, 27.42% of time is saved when the application is supported by MAMiMoU and specifically, the highest percentage of time is saved (41.03%) when the random editing collaboration is adopted.

To learn more about the reliability of an application's performance when it is supported and not supported by MAMiMoU, we produce graphs in Figure 7.14. Graphs (1) and (2) show performance deterioration, expressed as percentages, in the MCU and RCU environments respectively. The percentage is calculated based on task completion time recorded in the ACU environment (cf. Figure 7.14.a). In both graphs, we can see that the performance of an application without MAMiMoU's support deteriorates more than that of an application supported by MAMiMoU.

Percentage of difference	***Serialized editing***	***Simultaneous editing***	***Random editing***	***Average***
Rarely connected users (RCU)	23.15%	18.09%	41.03%	27.42%

Figure 7.13: Percentage of time saved by MAMiMoU in RCU environment

We can see that the application without middleware shows a slightly higher percentage of performance deterioration in the MCU environment in comparison with the application supported by middleware, while this percentage is much higher in the RCU environment (cf. Figure 7.14.b-c). From this observation, we conclude that performance of the application without MAMiMoU's support deteriorates faster with additional disconnections in the environment.

To prove the statistical significance of these experimental results, we perform some hypothesis testing on the measurements obtained. Our main objective is to show that the results we obtained are significant for the claim that in the RCU environment the application supported by MAMiMoU performs better than the application without MAMiMoU's support.

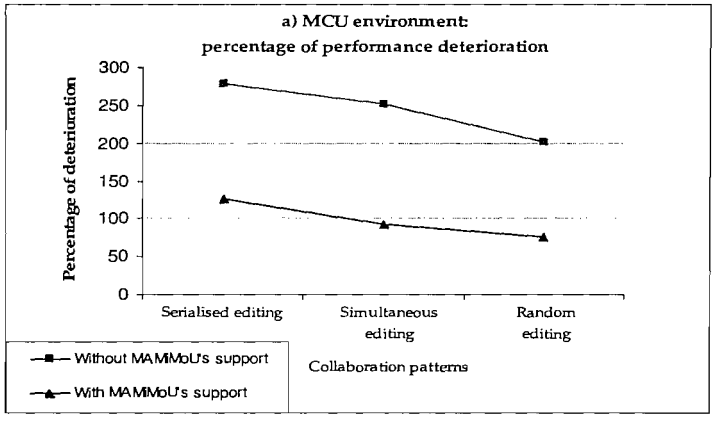
The error bar plot as shown in Figure 7.15 is based on Figure 7.12. From this plot, we can see that the task completion time range for the application both with and without middleware support overlap with each other except when random editing collaboration is adopted. With this observation, we can conclude that the two sets of results for random editing are significantly different from each other, but we could reach no conclusion in cases where serialized and simultaneous collaboration patterns are adopted. We do further hypothesis testing on task completion time recorded from these two simulated environments to prove that the measurements recorded from the application supported by MAMiMoU are significantly less than the measurements recorded when the application without MAMiMoU's support is performing the task.

For graph 1.
Percentage of deterioration = $(tct^{MCU} / tct^{ACU}) * 100$

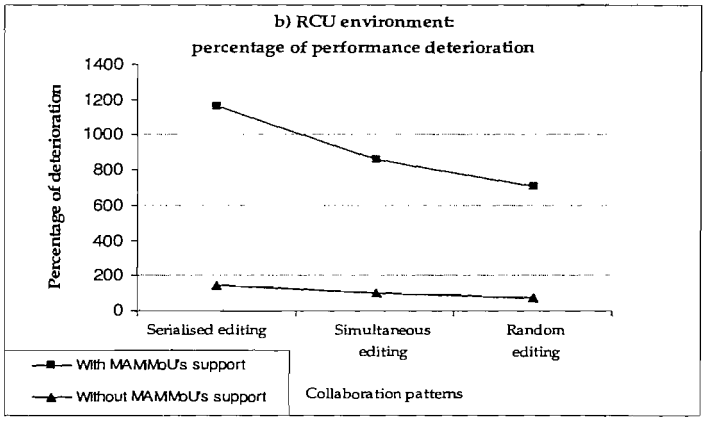
For graph 2.
Percentage of deterioration = $(tct^{RCU} / tct^{ACU}) * 100$

tct^{X} : task completion time in environment X.

a) Calculation of application's performance deterioration



b) Percentage of performance deterioration in MCU environment



c) Percentage of performance deterioration in RCU environment

Figure 7.14: Deterioration of application's performance

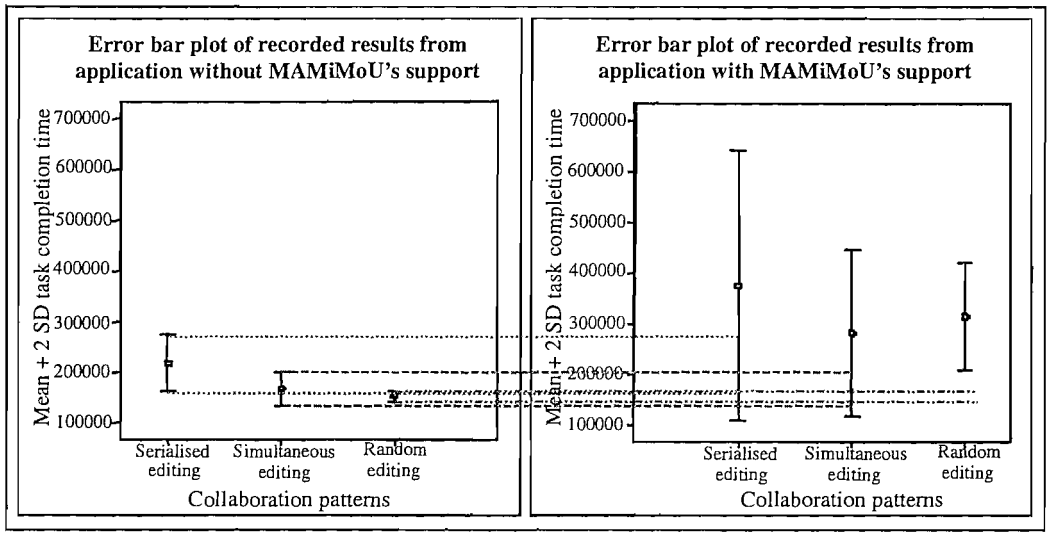


Figure 7.15: Error bar plot

We use SPSS software [14] to generate the hypothesis testing analysis in order to find the significance value of the results we have obtained. Since the recorded task completion times do not constitute a random sample from a larger population and they are not based on assumption that they come from a parametric family of distributions, we use non-parametric testing to calculate the significance value of the recorded results [18]. We use the Mann-Whitney U Test, which is a two-independent-samples non-parametric test. We define task completion time performed by an application supported by MAMiMoU and an application without MAMiMoU's support as the two independent samples to be tested.

Figure 7.16.a shows the results generated by SPSS software based on task completion time recorded when the serialized editing collaboration is adopted in the RCU environment. Specifically, it presents the summary statistics from a Mann-Whitney U Test. The table lists the average of ranks for the recorded task completion time grouped by application with and without MAMiMoU's support; values are *14.9* and *7.1* respectively. These two values are different, which means that their groups' spans differ. To further confirm that their spans really differ, the Mann-Whitney U Test or the Wilcoxon W Test is used. Figure 7.16.b contains statistics associated with a Mann-Whitney U Test for two independent measurements recorded by the application with and without MAMiMoU's support when serialized editing collaboration is adopted in the RCU environment.

The small significance value (<0.05) (cf. * in Figure 7.16.b) indicates that the two compared groups of values have different locations (value range) and are thus significantly different. This allows us to draw correct conclusions from the results obtained. We also performed the same test for the simultaneous and random editing collaborations. Thus, we can similarly conclude that differences in task completion time collected from the two different settings, namely an application with and without MAMiMoU's support, are significant. Based on this conclusion, we can correctly assume that the application supported by MAMiMoU performs the collaborative editing task faster in the RCU environment, which is deduced from the fact that the task completion time recorded by an application in this setting is significantly lower than the same measurements recorded from the same application when it is not supported by MAMiMoU. One reason why such results are observed only in the RCU environment is due to the application not supported by MAMiMoU failing to operate normally, e.g. frequent timeouts occuring on the mobile terminal when submitting requests to the repository. Whereas for an application supported by MAMiMoU, frequent disconnections of the mobile terminal are addressed by MAMiMoU, in which case the shadow stores and forwards messages while the mobile terminal is disconnected.

a) Mean rank for serialized editing collaboration

Ranks	*Application*	*N*	*Mean rank*	*Sum of Ranks*
Task completion time	Without MAMiMoU's support	10	14.9	149
	With MAMiMoU's support	10	7.1	61
	Total	20		

b) Test statistics for serialized editing collaboration

Test Statistics (b)	*Task completion time*
Mann-Whitney U	6
Wilcoxon W	61.
Z	-3.326
Asymp. Sig. (2-tailed)	0.001
Exact Sig. 2*(1-tailed Sig.)]	**0** (< 0.05 *)

a: Not corrected for ties
b: Grouping variable: with or without MAMiMoU's support

Figure 7.16: Mean rank and test statistics for serialized editing collaboration

Since we have proved that the results we obtained are significant, we can now further prove that our next hypothesis, which claims that a group of users adopting the serialized editing collaboration will suffer the most in the RCU environment (cf. Figure 7.7). This is because the serialized editing collaboration requires a high number of connectivity during the collaboration. This can be seen in Figures 7.12 to 7.14, where the serialized editing collaboration takes the longest time to complete the task, followed by random and simultaneous editing collaboration. With this observation, we can draw a conclusion that the serialized editing collaboration is not suitable to be adopted in the mobile users' environments, and simultaneous editing is the recommended collaboration pattern. We further analyse these results based on different collaboration patterns by comparing the task completion time of an application with and without MAMiMoU's support in the RCU environment. From Figure 7.13 we can see that total time being saved by MAMiMoU in the serialized editing collaboration is higher than in the simultaneous editing collaboration but lower than in the random editing collaboration.

For the following evaluations, we do the same hypothesis testing as explained in this section, but to simplify the presentation we will omit the explanations of such testing. Besides this, no graphs from the ACU and MCU environments will be described since we are specifically interested in observing what happens in the RCU environment.

7.2.2 Evaluation based on connection speed

In this evaluation, four users are simulated as collaborating over the same connection speed. We expect that the collaborative editing task can be completed faster with high-speed communication links (100Mbps) than with slower ones (56.6Kbps or 14.4Kbps), because of the reduced time required to exchange messages and transfer files between components. Our aim is to show that in all these three connection speed settings, MAMiMoU helps to speed up the collaborative task. We simulate the overall latency of a particular speed rate; for instance, in simulating a communication channel with a speed of 14.4Kbps, we use the formula shown in Figure 7.17 to calculate the delay. The delay will be simulated at the recipient every time a message is received. The actual_sending_time is the actual time needed to deliver the message via the connection being used. For example, to deliver a message or a file with the size of 100kB on a 100Mbps network, approximately 7.8125 milliseconds are needed.

```
simulateSendingTime := messageSize_in_Kbits / 14.4;
delay := simulateSendingTime – actual_sending_time;
```

messageSize_in_Kbits: size of message or file in Kbits
actual_sending_time: total time spent sending the message or file
delay: total delay to be simulated in seconds

Figure 7.17: Delay calculation

The results obtained in this evaluation are illustrated in Figures 7.18 to 7.20, where we can see that for all connection settings an application with MAMiMoU outperforms the one without it.

From Figure 7.21, we can see that in comparison to the application's performance with a 100Mbps connection, the performance of the application with MAMiMoU support deteriorates more than the application without MAMiMoU support with 56.6kbps and 14.4kbps connections. This may be due to the delay when messages are being transferred between the mobile terminal and shadow resulting in the messages failing to be delivered once a disconnection occurs. Thus the time needed to resend the messages prolongs the time taken to complete the collaborative editing task. However, even with this delay, MAMiMoU still saves time in all the three connection-speed settings (cf. Figure 7.22). Due to high deterioration in the application's performance, a lower percentage of time is saved with lower connection speeds. From this observation we can conclude that applications supported by MAMiMoU are also affected by slow connections, but not as much as applications without MAMiMoU support.

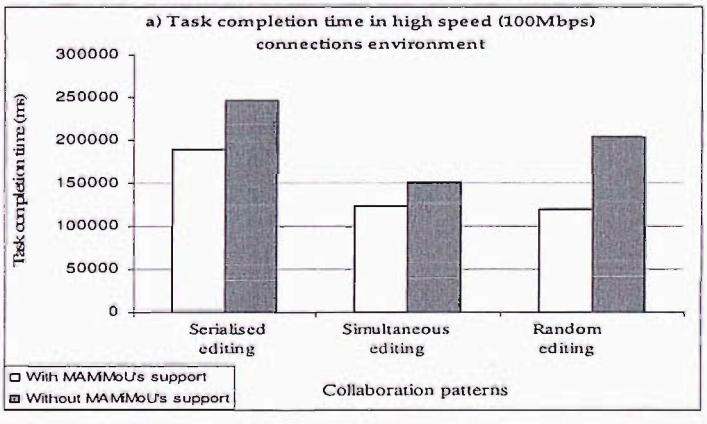


Figure 7.18: Effect of different users' connection speed in RCU environment: high-speed connections

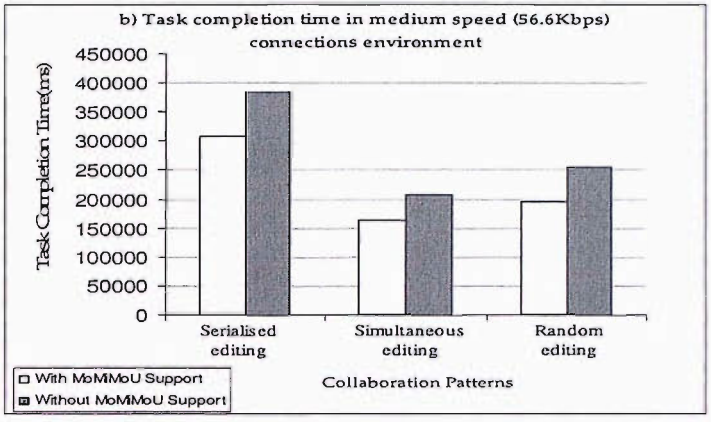


Figure 7.19: Effect of different users' connection speed in RCU environment: medium speed connections

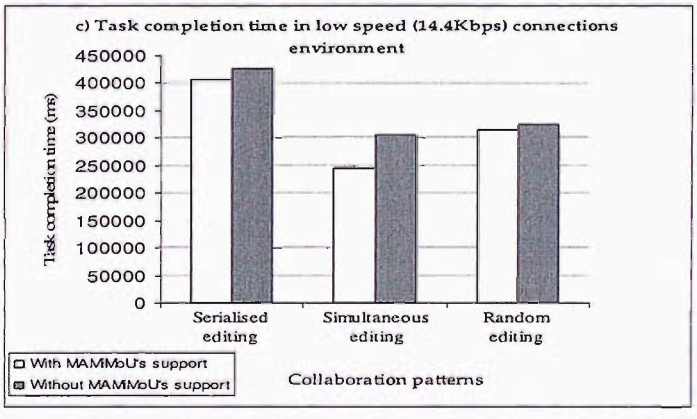


Figure 7.20: Effect of different users' connection speed in RCU environment: slow connections

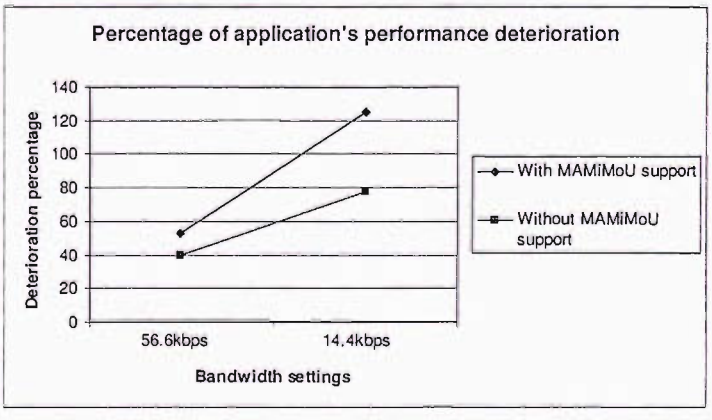


Figure 7.21: Percentage of application's performance deterioration

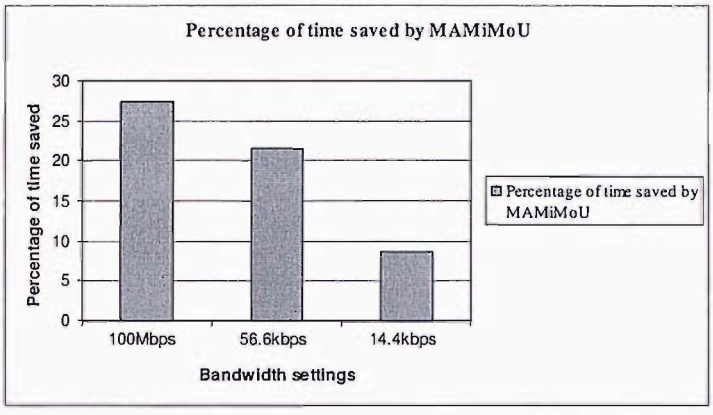


Figure 7.22: Percentage of time saved by MAMiMoU

7.2.3 Evaluation based on users' mobility

We expect that a group of highly mobile users tends to slow down the collaborative task due to the extra work the users' applications need to perform every time they reconnect to the network from a different location. In other words, users' mobility overhead is proportional to the number of users' migrations. Since MAMiMoU can perform tasks on behalf of the users while they are moving and reconnecting to their new location, MAMiMoU can help save time in completing the collaboration. Thus, we perform an evaluation to prove that MAMiMoU not only saves time in a collaboration between stationary users, but also in environments where users are moderately mobile (moving every 10 minutes) and highly mobile (moving every 5 minutes).

Figures 7.25 to 7.27 show that the application completes the task faster with MAMiMoU than without it, in all three settings of users' mobility. We can also see that more time is saved by MAMiMoU in environments where users are moderately and highly mobile.

From Figure 7.26, we can see that performance of the application without MAMiMoU support deteriorates significantly more than the application supported by MAMiMoU. This shows that MAMiMoU allows an application to perform reliably in different settings of users' mobility. While from Figure 7.27, we can see that MAMiMoU saves time in all the three users' mobility settings. The most time is saved when users are relocating every 10 minutes. The time being saved by MAMiMoU becomes less when the users relocate every 5 minutes, probably due to the

higher overhead time needed for the shadow to migrate and re-establish connection with the mobile terminal.

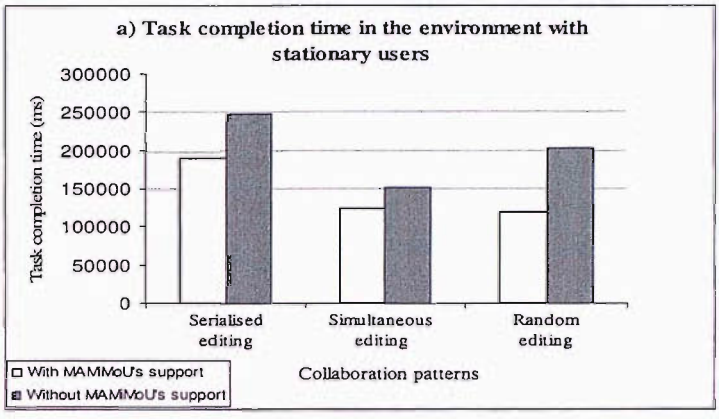


Figure 7.23: Effect of different users' mobility number in RCU environment: stationary users

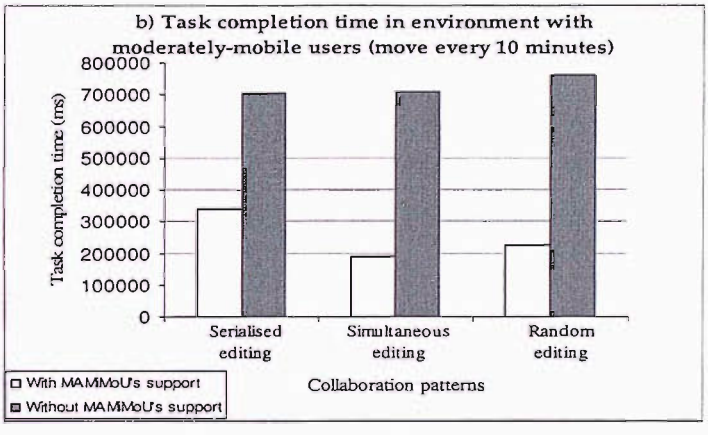


Figure 7.24: Effect of different users' mobility number in RCU environment: moderately mobile users

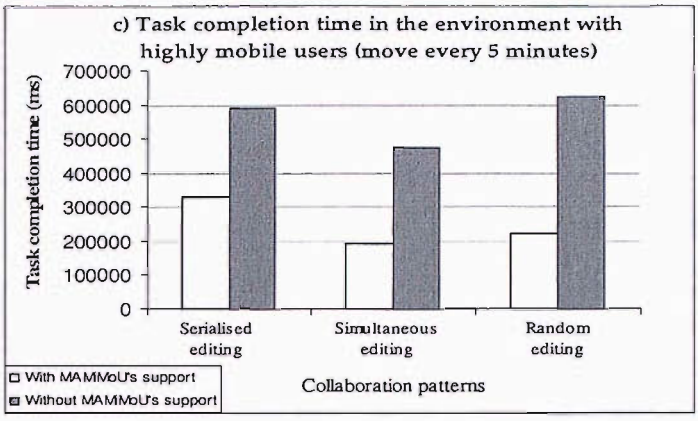


Figure 7.25: Effect of different users' mobility number in RCU environment: highly mobile users

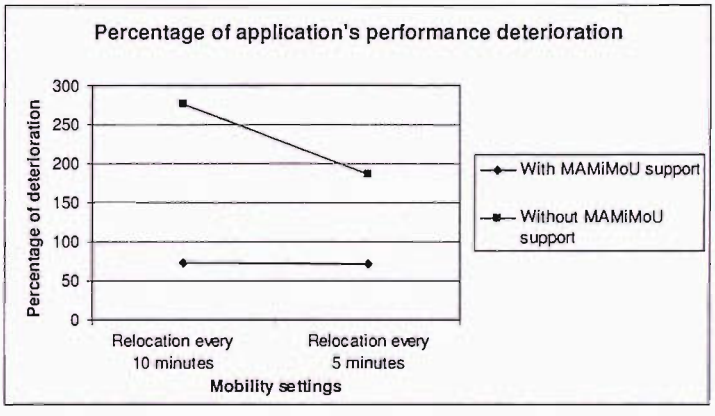


Figure 7. 26: Percentage of application's performance deterioration

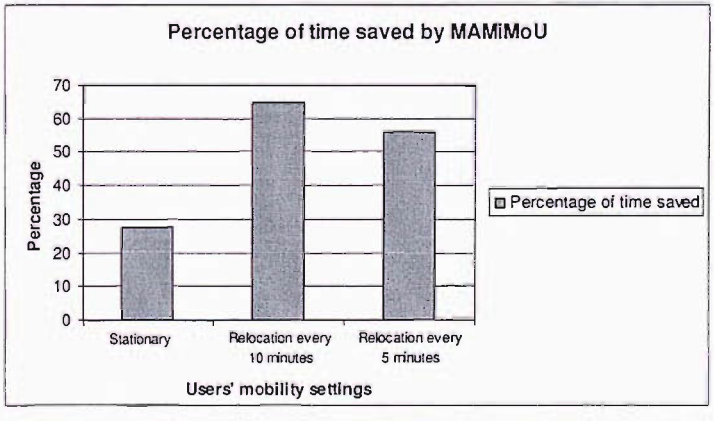


Figure 7. 27: Percentage of time saved by MAMiMoU

7.2.4 Evaluation based on collaboration group size

The next evaluation we performed was based on the size of the collaboration group. Our initial hypothesis is based on Figure 7.5 and Figure 7.9, from which we obtain the number of message exchange that reflects connectivity ratio required by a group of users' applications according to different group sizes, namely groups consisting of 4, 8 and 16 editors. From the estimation, we expect that a large number of editors will slow down the collaborative task, because of the increasing number of processes that need to be performed, i.e. more regular editors being involved in the collaboration means that more tentative versions need to be reviewed by the master editor. Thus, we expect such a delay to increase in proportion to the number of editors involved in the collaboration. Since MAMiMoU may perform some tasks while users are disconnected, some time can be saved in the collaboration process. Thus, in this evaluation, we want to prove that the bigger the size of the collaboration group, the more time can be saved by MAMiMoU in the RCU environment.

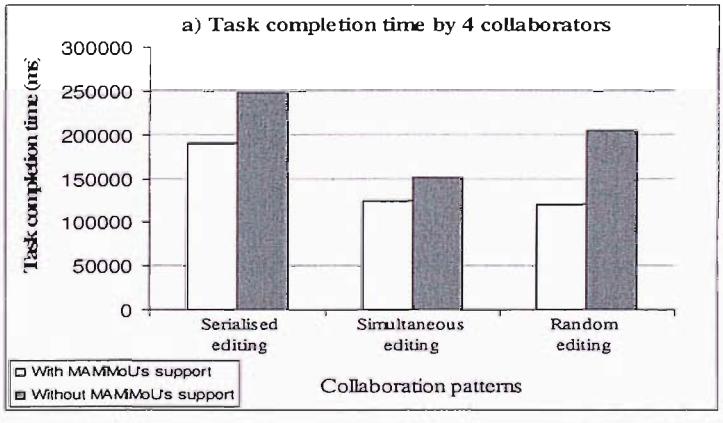


Figure 7.28: Effect of different collaboration-group size: small group

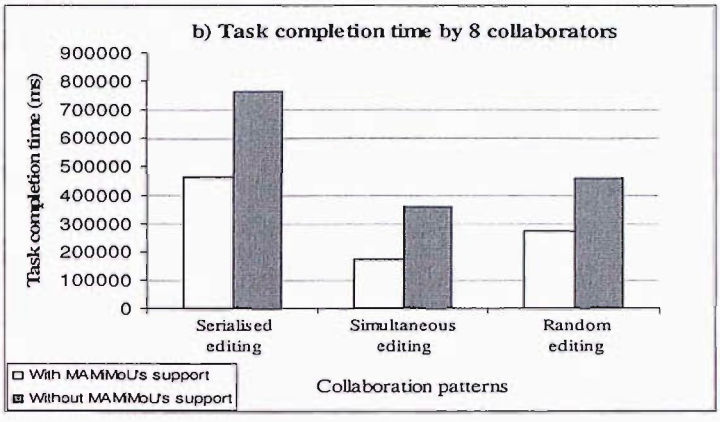


Figure 7.29: Effect of different collaboration group size: medium sized group

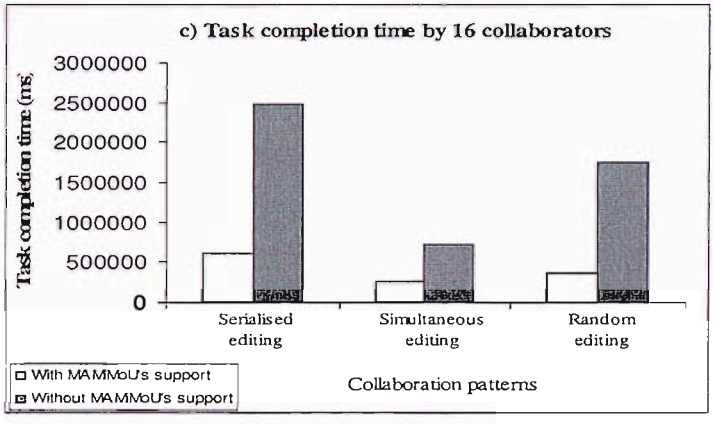


Figure 7.30: Effect of different collaboration group size: large group

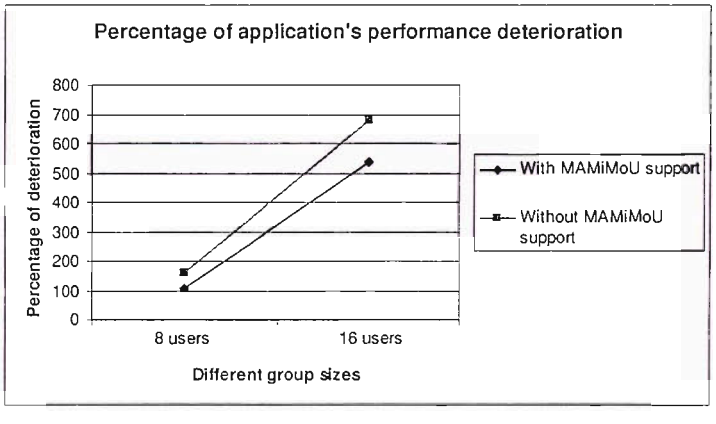


Figure 7.31: Average percentage of application's performance deterioration

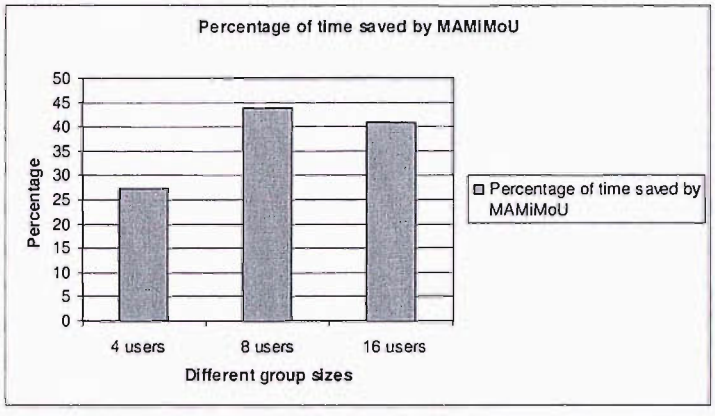


Figure 7.32: Percentage of time saved by MAMiMoU based on different collaboration group size

Figures 7.29 to 7.31 show the results we obtained from this evaluation. We can see that the application that is supported by MAMiMoU outperforms the application without MAMiMoU's support in all three different settings of collaboration group size. From Figure 7.31, we can see that performance of the application without MAMiMoU deteriorates more than the application supported by MAMiMoU when the group size is increased to 8 and 16 users. This shows that MAMiMoU allows the application to perform reliably when the number of collaborators using the system increases. Studying the results further, we calculate the percentage of time being saved by MAMiMoU in each setting (cf. Figure 7.32). From this graph, we can conclude that MAMiMoU saves time in all three settings of collaboration group size. However, the time being saved when the collaboration consists of 16 users is less than when the group size is 8 users and thus we could not prove that the bigger the size of the collaboration group, the more time can be saved by MAMiMoU. However, such results may be due to the higher number of shadows running on a machine resulting in a longer time taken for a shadow to process requests from the mobile terminal. Another possibility is due to the slow response from the repository that has to serve more users.

7.2.5 Evaluation based on file size

For our final evaluation, we simulate collaborative editing tasks by involving different file sizes. Our aim is to establish that the size of a document being edited by a group of collaborators would also affect the completion time of a collaborative editing task. Basically, this is due to the extra time being used for long file transfers when a large sized file is involved. Activities such as

"commit" and "check out" document involve such file transfers. Thus, these particular activities are expected to be lengthy when dealing with a large file.

From this evaluation, we believe that we will be able to prove that the support given by MAMiMoU to the application will help to shorten the task completion time. This is based on our prediction that the application without MAMiMoU's support will not be able to cope well in the RCU environment especially when the task involves a large sized file. This is basically due to the application's lack of support for handling file transfer when a disconnection occurs. The only failure handler is to repetitively attempt to transfer the file until it is successfully delivered. We assume that such repetitive attempts at a file transfer, especially involving a large file, further lengthen the task completion time. In the implementation, we transfer the file separately from the message. We use data buffering in order to transfer a single file in smaller parts (maximum of 300 bits a time). The file is transferred through MAMiMoU just like messages are stored and forward between the mobile terminal and network-based applications. This way the file is temporarily cached by the shadow on the platform it is operating before they are being forwarded to the mobile terminal or network-based application.

Here we can estimate which collaboration pattern may be greatly affected by the size of the file. The high number of activities that involve file transfer leads to a high probability that file size affects total task completion time. From Figure 7.7 we deduce the number of file transfers occurring for each collaboration pattern. From the calculated number of file transfers, we can see that each collaboration pattern involves the same number of file transfers. From this observation, we may assume that different file size may not have any effect for different collaboration patterns. Even though we could not reach a conclusion based on the collaboration patterns, from this evaluation we want to prove that the collaborative editing application supported by MAMiMoU will perform better in the RCU environment. Besides this, our aim is also to prove that when a larger file is involved, more time is being saved when the application is supported by MAMiMoU in comparison with when the application is without MAMiMoU's support.

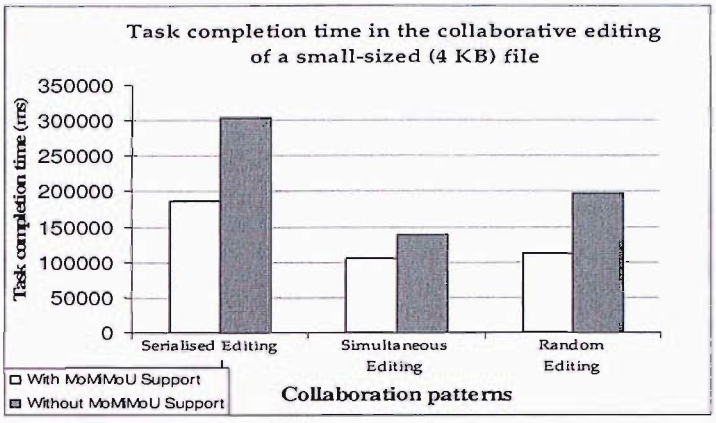


Figure 7.33: Collaborative editing of a small file

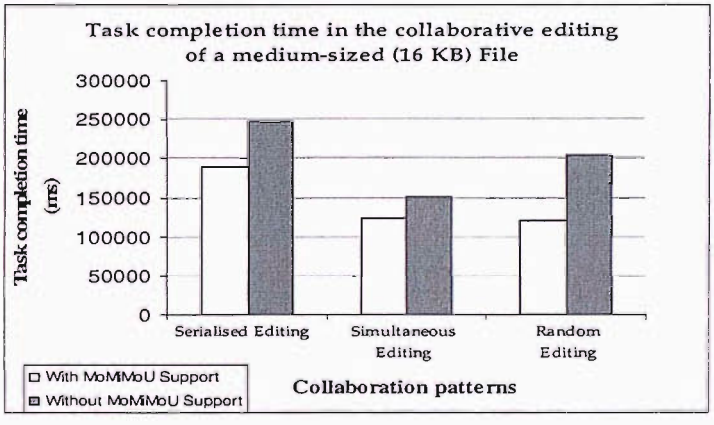


Figure 7.34: Collaborative editing of a medium-sized file

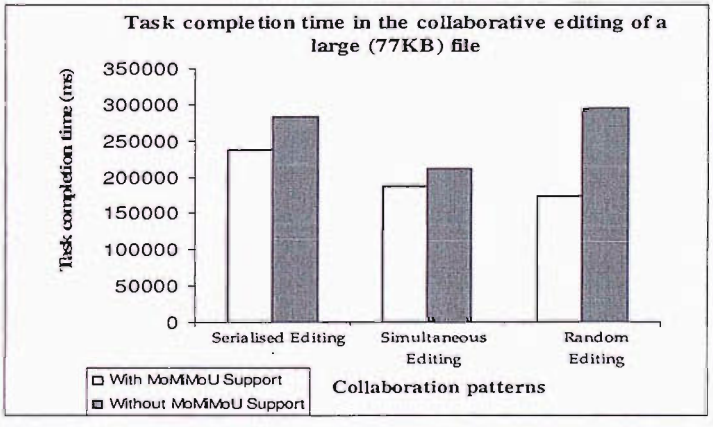


Figure 7.35: Collaborative editing of a large file

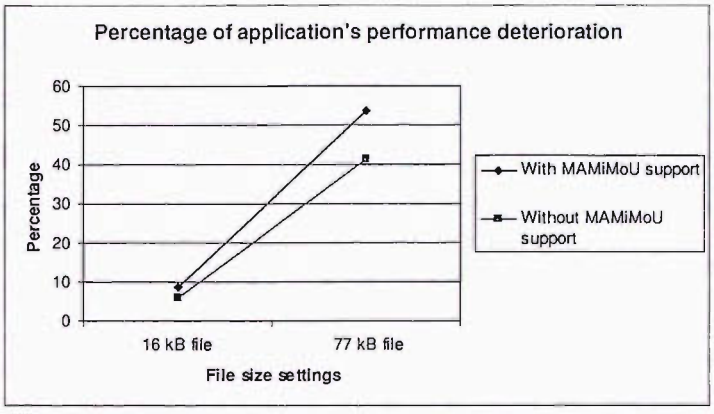


Figure 7.36: Average percentage of application's performance deterioration

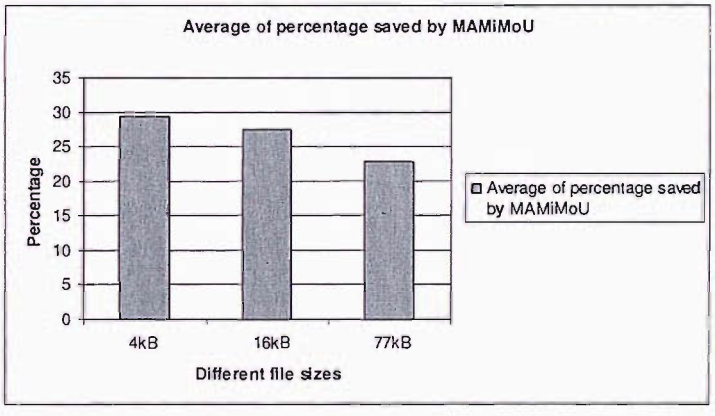


Figure 7.37: Time saved by MAMiMoU in RCU environment based on different file size

From graphs in Figures 7.34 to 7.36, we can see that collaborative editing tasks are completed faster by applications supported by MAMiMoU in comparison with applications without MAMiMoU's support. This applies to all simulations involving different file sizes. In Figure 7.36, we can see that performance of the application supported by MAMiMoU deteriorates more than the application without MAMiMoU support. Even though this is the case, MAMiMoU still reduces the amount of time needed to complete the collaborative editing task in all three file size settings (cf. Figure 7.37). The percentage of time saved is reduced when the size of the shared file increases and thus we could not prove that when a larger file is involved, the more time is being saved by MAMiMoU. However, such a result may be due to the need to transfer the file twice when MAMiMoU is used, i.e. a file transferred from a mobile terminal will be transferred to the shadow first, and then to the repository, and vice versa.

7.3 Analysis

First of all, we have proven that MAMiMoU improves performance of a collaborative editing application in RCU, which is a challenging environment for a mobile terminal's application to operate, where most of the time the application relies on a network-based application, e.g. a repository. Secondly, we have proven that in many different challenging features of RCU, (different bandwidths, users' mobility, collaboration group size and file size settings), MAMiMoU was able to maintain its ability to support the application and improves the application's performance. This is shown in the graphs showing the time saved by MAMiMoU in those different settings (cf. Section 7.2.2 – Section 7.2.5). The most time saved by MAMiMoU in the collaborative work is when mobile users are simulated to relocate every 10 and 5 minutes (cf. Section 7.2.3), in which case MAMiMoU saves more than 50% of time. The least time is saved by MAMiMoU is when the users have a connection speed of 14.4kbps, in which case less than 10% of time is saved (cf. Section 7.2.2).

In terms of collaboration patterns, we observed that the serialized editing collaboration always results in the longest time needed to complete a collaborative editing task, while simultaneous editing collaboration always results in the least time needed. From this, we can draw a conclusion that simultaneous editing is better for mobile users' environments, while serialized editing should not be adopted. Simultaneous editing may offer this advantage due to the parallelism it promotes in performing the collaborative activities.

7.4 Discussion

The evaluation we have presented in this chapter is unique in the sense that it combines the experiment on a collaborative application with a middleware system. Evaluations and analysis performed on CSCW systems [24][25][19] involve the study of interactions and technologies used in the collaborative work, while system evaluations such as presented in [22][34] focus only on evaluating the performance of mobile agent-based systems. The simulation is designed in such a way since the application is expected to operate in mobile users' environments. The features forming simulated dimensions in the experiment reflect the real challenging features of the environments of mobile users.

The results we obtained in Section 7.2.1 may raise a question of whether MAMiMoU is suitable to support the applications, since in the ACU and MCU environments, the application without MAMiMoU support performed better. This is due to the overhead introduced by MAMiMoU in migrating shadows and by having to store and forward messages between the mobile terminal and repository. We can conclude that in ACU and MCU environments, having direct interactions with the repository are more advantageous, while in RCU, MAMiMoU is needed to support the application and improves its performance. To support an application in all the three different environments, the application can be provided with two modes; the first mode is for ACU and MCU environments that do not require the support from MAMiMoU, while the second mode is for RCU environment, in which case the MAMiMoU service will be enabled. The switch between the two modes can be automated by having a mechanism monitoring the mobile terminal's connectivity to the network. If the mechanism detects that mobile terminal is connected intermittently to the network within the ratio of 10% or less connectivity over a period of time, the MAMiMoU service will be enabled. Otherwise, normal operation without MAMiMoU's support will be performed.

7.5 Conclusion

In this chapter, we have presented an empirical evaluation analysing the performance of the collaborative editing application when it is supported and not supported by MAMiMoU. From this evaluation, we have proven that MAMiMoU is able to improve the application's performance in RCU environment, which has other challenging and dynamic features, i.e. different bandwidth, different users' mobility, different collaboration group sizes and different shared file sizes settings. Since RCU is a typical environment in mobile users' environments, we can draw a conclusion that MAMiMoU is a suitable approach and solution to address problems in mobile users' environments. We now proceed to the final chapter, in which we present our conclusion for the whole thesis and describe the future work.

Chapter 8

Conclusion and Future Work

In this thesis we have described the environment of mobile users and the challenges in developing suitable applications for mobile users. Our first aim was to address the common limitations in mobile users' environments, which are caused by the quality of users' connections, users' mobility and also the limitations of users' mobile terminals. We envisioned an abstraction layer that hides users' mobility from the applications and the complexity of the communications between mobile terminal and network-based applications. To realize this, we developed a middleware called Mobile-Agent Based Middleware for Mobile Users (MAMiMoU), which employs mobile agents as its main component. We use mobile agents as they can autonomously perform tasks for mobile users and migrate to users' vicinities. Such close proximity between the user's mobile terminal and the shadow can help reduce the bandwidth required and thus improve the middleware's performance [34].

Our other interest was to support the sharing between mobile users of documents that are hosted on the users' mobile terminals. We call such documents mobile documents; the sharing of these documents between users is common in many types of collaborations. A challenge in supporting the sharing of mobile documents is to make them accessible to other users, as they may be hardly available to the network due to frequent disconnections of mobile terminals from the fixed network. Besides this, supporting collaborations between mobile users is also difficult, as a group of collaborating users is hardly ever online at the same time. Complex applications to support both mobile document sharing and collaborative editing are impossible to deploy on mobile terminals that have limited capability. Thus, our approach has been to design a collaboration protocol to support a collaborative editing application by taking into account the characteristics of

both mobile documents and mobile users' environments. To improve the application's performance in mobile users' environments, we integrated the application with MAMiMoU.

We performed an evaluation involving a test of a real implementation by running it in a simulated mobile users' environment, where users have intermittent connectivity and mobility. In such an environment, we monitored the performance of the collaborative editing application used by a group of mobile collaborators to complete collaborative editing tasks. In one setting the application was supported by MAMiMoU, while in the other it was not. The performance of the application in both settings was compared and discussed. From the results obtained, we proved that the application's performance was significantly improved. This proves our thesis, which states that MAMiMoU is a practical solution to support mobile users' applications in mobile users' environments, in which case MAMiMoU improves their performance. MAMiMoU was formalized based on the ideas we obtained from the study we conducted on existing mobile-agent based systems.

8.1 Conclusion

We have successfully accomplished our aims as stated in Chapter 1. We have addressed the common limitations in mobile users' environments by having designed and implemented MAMiMoU. MAMiMoU's architecture and coordination algorithm are two of our main contributions in this thesis. MAMiMoU allows users to seamlessly exploit resources on the fixed network while on the move. It acts as an abstraction layer that hides complexity in communications from the applications, allowing mobile terminal applications and network-based applications to transparently interact with each other. MAMiMoU's coordination algorithm describes the interaction protocol between its components, the reconciliation protocol between multiple shadows and failure handlers. MAMiMoU offers a substrate for building distributed applications across mobile terminals and fixed infrastructures, allowing the large scale deployment of advanced services to mobile users.

Another contribution presented in this thesis is the support for collaborative editing of mobile documents between mobile users. We presented a collaborative editing application by describing its architecture and the supporting collaboration protocol, which is the major part of the application. The collaboration protocol specifies the components' activities, used to maintain

consistency between shared documents, and coordinates the collaborative editing activities between mobile users. The application allows mobile documents to be shared between mobile users by having documents' copies hosted on the network. Such documents' copies can also be cached on users' mobile terminals, allowing the users to access the document while disconnected from the network. Tailored to the characteristics of mobile users, the collaboration protocol was also designed to support asynchronous collaboration between mobile users.

Our final contribution is the design of an evaluation methodology, which is used to measure the effectiveness of MAMiMoU as a middleware supporting an application that operates in mobile users' environments. In this evaluation, the common characteristics of mobile users' environments were simulated, in which MAMiMoU is put to the test by supporting the collaborative editing application. Different patterns of collaboration between mobile users were simulated and the application's performance was measured based on the total time used by the users to complete a collaborative editing task. Other aspects considered in the evaluation include the effects of different users' connectivity, users' connection speed, users' mobility, collaboration group size and shared document size. From this evaluation, we concluded that MAMiMoU can improve the performance of an application operating in an environment that closely resembles real mobile users' environments. With this result, we proved that MAMiMoU is able to support applications in the challenging environments of mobile users by improving the applications' performance. As a contribution to the field of mobile collaboration, our evaluation methodology can be re-used to measure the performance of other collaborative applications in mobile users' environment.

8.2 Future Work

In the future, we plan on deploying MAMiMoU in real mobile users' environment, and extending MAMiMoU in order to have more systematic mobility management. We divide the future work into two major plans, each addressing some research questions and the work involved. The first plan focuses on the deployment of MAMiMoU, while the second plan focuses on the users' mobility prediction mechanism that can be added to the middleware. Both plans are elaborated on below.

8.2.1 MAMiMoU deployment

The current implementation of MAMiMoU is in the form of a demonstrator. In the future, we wish to deploy MAMiMoU as a concrete middleware system, able to support real mobile users and applications. In general we need to implement all of the assumptions that we made, i.e. each local area environment has at least a shadow handler and a platform that is able to run mobile agents. The first step in achieving this is to develop clear concepts and methods for the deployment that will hide the details and complexity related to the implementation. The concepts may include the description of functionalities offered by MAMiMoU to the environment in which it will be deployed. These are required in order to bridge the gap between component specifications and their actual configuration and distribution across the real fixed network and mobile terminals.

The interfaces and protocols between MAMiMoU's components and the API offered by MAMiMoU to applications needs to be well-defined and standardized. This is essential to support the deployment of MAMiMoU by multiple network operators and on heterogeneous platforms. The shadow handler and mobile agent's platform has to be installed and initialized on the user's network, while an MT-agent has to be installed on the mobile terminal. In performing this, specific requirements must be considered, e.g. a mobile agent platform has to be installed and run on the same node as the shadow-handler.

To deploy the middleware, two types of packages can be prepared for the network operators and mobile applications providers: a package consisting of components deployable on the fixed network; and another package that consists of MT-agents deployable on the mobile terminal. Each package should include the component descriptors, which provide a method of abstracting the source code of the application from any reference to software or hardware configuration [2]. Each package may also include additional information on the components' requirements, dependencies and their constraints, which might not be part of the component specification itself. This way, MAMiMoU's components can be easily installed on the target terminals and establish their initial configuration. The deployment process may include designing a methodology to design applications using MAMiMoU's support, e.g. to decide whether to offload the application's components to fixed network or not. Additionally, design metrics can be adopted to offer guidance and feedback regarding the quality of the designs.

8.2.2 Mobility Prediction

Currently, the shadow in MAMiMoU passively migrates to the location of the mobile terminal when it connects to a new location, i.e. it moves only when requested. This creates a delay in providing the service to users. A solution to this problem involves having the shadow migrate just before the user reattaches to the new location. One way to allow this is by having the user always inform the system well beforehand of where he / she is going to go. But we may want to minimize reliance on user feedback. Another way to achieve this solution would be to use a mechanism that predicts the next location of a mobile user. When a prediction is made, the shadow, possibly with all of its applications, can migrate in advance. Then, on arrival at its new location, the mobile terminal could immediately become connected with the shadow, and continue to interact with the applications through the shadow without encountering a delay in connecting to the shadow. Such a capability would be especially important when a large number of shadows and applications needed to migrate closer to the mobile terminal. We believe such an approach can improve the performance of the middleware since it would be instantly ready to serve the user without any migration delay. By having multiple possible predictions for the user's next location, we may consider cloning the user's shadow at each of those locations. With cloned shadows, we would need to address the coordination and interaction between them. This will be regarded as a research question in this work.

One example of users' mobility prediction is presented in [61], which is called the predictive mobility management (PMM). It is based on the user's movement history, i.e. their previous movement patterns. Based on these movement patterns, a pattern-matching/recognition-based Mobile Motion Prediction algorithm (MMP) is proposed, which can be used to estimate the future location of the mobile user. Any movement that cannot be classified by the simple mobility patterns is classified as random movement. The random parts of the movement are modeled using Stochastic Processes and Markov Chain, which can provide detailed calculations of the probability distributions for the next possible states with a given confidence level. Users' mobility prediction mechanisms such as this are normally employed by mobile service providers, which could be used by MAMiMoU. Alternatively, the same type of services may be offered by some applications on the network.

References

[1] Bagrodia R., Phan T. and Guy R. (2003). *A Scalable, Distributed Middleware Service Architecture to Support Mobile Internet Applications*. Wireless Networks, 9(4): 311-320, 2003. 10 pages.

[2] Baude F., Caromel D., Huet F., Mestre L., and Vayssiµere J. (2002). *Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications*. In 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11, 2002.

[3] Baumgarten H., Borrmann L., Köhler T., Pink S., Lacoste G. and Reichert F. (1996). *Middleware for a New Generation of Mobile Networks: The ACTS OnTheMove Project.* INET'96 Conference of the Internet Society. Montreal, Canada. 6/1996.

[4] Bellifemine F., Poggi A. and Rimassa G. (2001). *Developing multi agent systems with a FIPA-compliant agent framework. in Software - Practice & Experience*. John Wiley & Sons, Ltd. vol no. 31, 2001, page 103-128.

[5] Bhattacharjee S., Calvert K.L., and Zegura E.W. (1997). *An Architecture for Active Networking*. High Performance Networking (HPN'97), Apr. 1997.

[6] Broos R., Dillenseger B., Dini P., Hong T., Leichsenring A., Leith M., Malville E., Nietfeld M. and Sadi K. (2000). *Mobile Mobile Agent Platform Assessment Report.* Contribution to the EU Advanced Communications Technology and Services (ACTS) Programme, 2000. Available at http://www.fokus.gmd de/research/cc/ecco/climate/ap-documents/miami-agplatf.pdf.

[7] Cabri G. Leonardi L. and Zambonelli F. (2000). *Weak and Strong Mobility in Mobile Agent Applications*. In Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java (PA JAVA 2000), Manchester (UK).

[8] Caporuscio M. and Inverardi P. (2003). *Yet Another Framework for Supporting Mobile and Collaborative Work*. International Workshop on Distributed and Mobile Collaboration, Linz, Austria, June 2003.

[9] Caporuscio M., Carzaniga A. and A. L. Wolf. *Design and evaluation of a support service for mobile, wireless publish/subscribe applications*. Technical Report CU-CS-944-03, Department of Computer Science, University of Colorado, Jan. 2003. Available at http://www.di.univaq.it/.

[10] Carzaniga A., Picco G. P. and Vigna G. (1997). *Designing Distributed Applications with Mobile Code Paradigms*. In Proceedings of the 19th International Conference on Software Engineering (ICSE'97), pages 22-32. ACM Press, 1997.

[11] Carzaniga A., Rosenblum D. S. and Wolf A. L. (2001). *Design and Evaluation of a Wide-Area Event Notification Service*. ACM Transactions on Computer Systems, 19(3):332-383, August 2001.

[12] Cederqvist P. *Version Management with CVS*. Available at (http://www.cvshome.org/docs/manual/).

[13] Chaki S., Fenkam P., Gall H., Jha S., Kirda E. and Veith H. (2003). *Integrating Publish/Subscribe into a Mobile Teamwork Support Platform*. In Prooceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE) 2003.

[14] Chicago IL SPSS Inc. *SPSS base 12.0 for Windows user's guide*, 2000.

[15] Chyi D. (2000). *An Infrastructure for a Mobile-Agent System that Provides Personalized Services to Mobile Devices*. Dartmouth College Computer Science Technical Report TR2000-370, 2000.

[16] Clements P.E., Papaioannou T. and Edwards J. (1997). *Aglets: Enabling the virtual enterprise*. Published and Presented at International Conference Managing Enterprises-Stakeholders, Engineering, Logistics and Achievement (ME-SELA'97) Loughborough, UK. July 1997 (ME-SELA '97).

[17] Czerwinski S.E., Zhao B.Y., Hodes T.D., Joseph A.D., and Katz R.H. (1999). *An Architecture for a Secure Service Discovery Service*. In Proceedings of the Fifth Annual International Conference on Mobile Computing and Networks (MobiCom '99), Seattle, WA, August 1999, pp. 24-35.

[18] Dallal G.E. (2003). *The Little Handbook of Statistical Practice*. Chapter Nonparametric Statistics. http://www.statisticalpractice.com/, 2003.

[19] David B., Delotte O., Chalon R., Tarpin-Bernard F. and Saikali K (2003). *Patterns in Collaborative System Design*. Development and Use IFIP Working Group 13.2 workshop "Methodologies for user centered systems design", 2nd Workshop on Software and Usability Cross-pollination : The Role of Usability Patterns, INTERACT 2003, Sept 2003, Zurich.

[20] de Lara E., Kumar R., Wallach D. S. and Zwaenepoel W. (2001). *Collaboration and Document Editing on Bandwidth-Limited Devices*. In Proceedings of the Workshop on Application Models and Programming Tools for Ubiquitous Computing (UbiTools).

Atlanta, Georgia. September, 2001.

[21] Dejan S. M. (1999). *Trend Wars: Mobile agent applications.* IEEE Concurrency 7(3): 80-90, 1999.

[22] Dikaiakos M., Kyriakaou M. and Samaras G. (2001). *Performance Evaluation of Mobile-Agent Middleware: A Hierarchical Approach.* In Proceedings of the 5th IEEE International Conference on Mobile Agents, J.P. Picco (ed.), Lecture Notes of Computer Science series, vol. 2240, pages 244-259, Springer, Atlanta, USA, December 2001.

[23] Dobridge, T., Lin J., Rajan D., Roscoe T., Brandenburg J., and Byerly B. (1998). *Artefact: A Framework for Low-Overhead Web-Based Collaborative Systems.* In Proceedings of the ACM CSCW'98, November 1998, Page 189 - 196.

[24] Ellis C. (1999). *An evaluation framework for collaborative Systems.* Report to NTT Tech. Group. University of Colorado CS Report CU-CS-109-99, 1999.

[25] Ellis, C. A.and Nutt G.J. (1992). *The Modelling and analysis of collaborative Systems.* In CSCW Tools and Technologies Workshop October 1992.

[26] Farooq U., Parsons E. W. and Majumdar S. (2004). *Performance of publish/subscribe middleware in mobile wireless networks.* In Proceedings of the fourth international workshop on Software and performance, Redwood Shores, California, 278 - 289.

[27] Forman G. H. and Zahorjan J. (1994). *The Challenges of Mobile Computing.* IEEE Computer, V 27, N 4, (April 1994), pp. 38-47.

[28] Fuggetta A., Picco G.P. and Vigna G. (1998). *Understanding Code Mobility.* IEEE Transactions on Software Engineering, Volume 24, Issue 5, page 342-361. May 1998.

[29] Gadah A. and Kunz T. (2003). *A Survey of Middleware Paradigms for Mobile Computing.* Carleton University, Systems and Computer Engineering, Technical Report SCE-03-16, July 2003.

[30] Ganguli H. (2002). *Java Security.* Chapter 12 - An Introduction to Cryptography. Premier Press ©. 2002.

[31] Geihs K. (2001). *Middleware challenges ahead.* IEEE Computer, 34(6):24-31, June 2001.

[32] Gong L. (1998). *Java Security Architecture (JDK1.2).* Technical report, Sun Microsystems, March 1998.

[33] *Grasshopper A Platform for Mobile Software Agents.* Available at http://www.grasshopper.de/download/doc/GrasshopperIntroduction.pdf

[34] Gray R., Kotz D., Peterson R. A., Barton J., Chacon D., Gerken P., Hofmann M., Bradshaw J., Breedy M., Jeffers R and Suri N (2001). *Mobile-Agent versus Client/Server Performance: Scalability in an Information-Retrieval Task.* In Proceedings of Mobile

Agents (229-243). 2001.

[35] Groove Networks (2002). *Groove: Good for Bandwidth.* Available at http://www.groove.net/pdf/Groove_and_Bandwidth.pdf

[36] Gudgin M., Hadley M., Moreau J.-J. and Nielsen H. F. (2001). *Soap version 1.2. Technical report.* World Wide Web Consortium.

[37] Harrison C., Chess D. and Kershenbaum A. (1995). *Mobile Agents: Are they a good idea?.* IBM T.J. Watson Research Center, Yorktown Heights, New York, March 1995.

[38] Helin, H., Laamanen, H., and Raatikainen, K. (1999). *Mobile Agent Communication in Wireless Networks.* In Proceedings of the European Wireless'99, pages 211-216, October 1999.

[39] Hofmann M. O., McGovern A. and Whitebread K. R. (2002). *Mobile agents on the digital battlefield.* In Proceedings of the second international conference on Autonomous agents, (page 219 - 225). Minneapolis, Minnesota, United States, April 2002.

[40] HohL F. (1998). *Time limited blackbox security: Protecting mobile agents from malicious hosts.* In Mobile Agents Qnd Security, number 1419 in LNCS. Springer-Verlag, 1998.

[41] Hurst L. (1998). *MCK: Wireless Communication with Mobile Agents.* Technical Report (TCD-CS-1998-05). Trinity College Dublin, Dublin 1998.

[42] IBM developerWorks (2004). *Publish-Subscribe Notification for Web services.* White Paper. http://www-106.ibm.com/developerworks/library/ws-pubsub/WS-PubSub.pdf WS-Notification:IBM developerWorks.

[43] IBM, Inc. (1998). *IBM Aglets Documentation.* Available at URL http://aglets.trl.ibm.co.jp, 1998.

[44] Intergraph Corporation (2003). *Wireless Technology Overview*, White Paper 2003. http://solutions.intergraph.com/core/white_papers/WirelessTech20043513A.pdf

[45] Jacobsen K. and Johansen D. (1997). *Mobile Software on Mobile Hardware: Experiences with TACOMA on PDAs.* Technical Report 97-32, Department of Computer Science,University of Troms, Norway, 1997.

[46] JADE . A White Paper. http://jade.tilab.com/papers/WhitePaperJADEEXP.pdf.

[47] Jennings N. R., Sycara K. and Wooldridge M. (1998). *A Roadmap of Agent Research and Development.* Autonomous Agents and Multi-Agent Systems, Volume 1, Issue 1 1998. (7 - 38).

[48] Jul E., Levy H., Hutchinson N. and Black. A. (1988). *Fine-grained mobility in the Emerald system.* ACM Transactions on Computer Systems 6, 1. (Feb. 1988), 109-133.

[49] Kistler J. J. and Satyanarayanan M. (1992). *Disconnected Operation in the Coda File*

System. ACM Transactions on Computer Systems, 10(1), February 1992.

[50] Kleinrock L. (1995). *Nomadic computing: An opportunity.* Computer Communications Review (Jan. 1995).

[51] Kock M (1995). *The Collaborative Multi-User Editor Project IRIS.* Technical Report TUM-I9524, University of Munich, Aug. 1995.

[52] Koskimies O. and Raatikainen K. (2000). *Partitioning Applications with Agents.* In Proceedings of the Second International Workshop on Mobile Agents for Telecommunication Applications (MATA2000). 2000.

[53] Kotz D., Gray R. S. and Rus D. (2002). *Future Directions for Mobile-Agent Research.* Technical Report TR2002-415. Hanover, NH. 2002.

[54] Kun Y., Xin G. and Dayou L. (2000). *Security in mobile agent system: problems and approaches.* ACM SIGOPS Operating Systems Review, Volume 34, Issue 1 (January 2000). (21 - 28).

[55] Kurkovsky S., Bhagyavati and Ray A. (2004). *A collaborative problem-solving framework for mobile devices.* ACM Southeast Regional Conference 2004: 5-10

[56] La Corte A., A. Puliafito A. and Tomarchio O. (1999). *An Agent-based framework for Mobile Users.* In 3rd European Research Seminar On Advances In Distributed Systems (ERSADS'99), Madeira (Portugal), April 1999.

[57] La Porta T.F., Sabnani K.K. and Gitlin R.D. (1996). *Challenges for nomadic computing.* Mobility management and wireless communications 1996.

[58] Lange D. B. (1998). *Mobile Objects and Mobile Agents: The Future of Distributed Computing.* In Proceedings of The European Conference on Object-Oriented Programming '98, 1998.

[59] Lange D. B. and Ishima M. (1998). *Program and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.

[60] Lauzac S.W., Chrysanthis P.K. (2002). *Personalizing information gathering for mobile database clients.* Proceedings of the 17th symposium on Proceedings of the 2002 ACM symposium on applied computing, March 2002, pp. 49-56.

[61] Liu G. and Maguire G. (1995). *A Predictive Mobility Management Scheme for Supporting Wireless Mobile Computing.* Technical Report, TRITA-IT R 95:04, Royal Institute of Technology (KTH), Feb 1995.

[62] Maffeis S. (2000). *Communication Middleware for Mobile Applications – A Comparison.* White Paper, SoftWired AG, August 2000.

[63] Mahmoud Q.H. (2001). *MobiAgent: An Agent-based Approach to Wireless Information*

Systems. In the Proceedings of the 3rd International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2001).

[64] Mallat N., Rossi M. and Tuunainen V. K. (2004). *Mobile banking services.* Communications of the ACM, Volume 47, Issue 5 (May 2004), (42 - 46).

[65] Marques P., Simões P., Silva L., Boavida F. and Silva J. (2001). *Providing Applications with Mobile Agent Technology*. In the Proceedings of the 4th International Conference on Open Architectures and Network Programming (OPENARCH). 2001.

[66] Mattern F., Sturm P (2003). *From Distributed Systems to Ubiquitous Computing - The State of the Art, Trends, and Prospects of Future Networked Systems*. In Klaus Irmscher, Klaus-Peter Fähnrich (Eds.) Proc. KIVS 2003, pp. 3-25, Springer-Verlag, February 2003.

[67] Mihailescu P. and Binder W. (2001). *A Mobile Agent Framework for M-Commerce*. Agents in E-Business (AgEB-2001), Workshop of the Informatik 2001, Vienna, Austria, September 2001.

[68] Mitsubishi Electric ITA Horizon Systems Laboratory. Technology at a glance: Concordia, Java Mobile Agent Technology. Available at http://www.meitca.com/HSL/Projects/Concordia/Concordia-at-a-glance.html

[69] Moreau L. (2001). *Distributed Directory Service and Message Router for Mobile Agents*. In Journal of Science of Computer Programming, Volume 39 (2-3), Page 249-272. 2001.

[70] Moreau L. (2002). *A Fault-Tolerant Directory Service for Mobile Agents based on Forwarding Pointers*. In The 17th ACM Symposium on Applied Computing (SAC'2002) , Track on Agents, Interactions, Mobility and Systems, Madrid, March 2002.

[71] Moreau L., De Roure D., Hall W. and Jennings N. (2000). *Case for Support: MAGNITUDE Mobile AGents Negotiating for ITinerant Users in the Distributed Enterprise*. http://www.ecs.soton.ac.uk/~lavm/magnitude/.

[72] Moreau L., Gibbins N., DeRoure D., El-Beltagy S., Hall W., Hughes G., Joyce D., Kim S. and Michaelides D. (2000). *SoFAR with DIM Agents: An Agent Framework for Distributed Information Management.* In Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 2000), pages 369-388, 2000.

[73] Moreau L., Tan V. and Gibbins N. (2001). *Transparent migration of mobile agents*. In IEE Seminar: Mobile Agents - Where are They Going?, pages 2/1-2/11, Savoy Place, London, April 2001. IEE.

[74] Moreau L., Zaini N., Cruickshank D. and De Roure D. (2003). *SoFAR: An Agent Framework for Distributed Information Management*. Chapter in Intelligent Agent

Software Engineering, pages 49-67. Idea Group Publishing, 2003.

[75] Moreau L., Zaini N., Zhou J., Jennings N. R., Wei Y. Z., Hall W., De Roure D., Gilchrist I., O'Dell M., Reich S., Berka T., and Di Napoli C. (2002). *A Market-Based Recommender System.* In Proceedings of the Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems at AAMAS 2002 (AOIS'02), Bologna, Italy, July 2002. http://CEUR-WS.org/Vol-59/.

[76] Morton S. and Bukhres O. (1997). *Utilizing mobile computing in the Wishard Memorial Hospital ambulatory service.* In Proceedings of the 1997 ACM symposium on Applied computing, San Jose, California, United States, (287 - 294).

[77] Nalla A., Sumi A. and Renganarayanan V. (2002). *aZIMAs: Almost Zero Infrastructure Mobile Agents System.* In Proceedings of the IEEE Wireless Communications and Networking Conference, March 2002.

[78] Necula G. and Lee P. (1996). *Safe kernel extensions without run-time checking.* In Proceedings of the "2nd Symposium on Operating System Design and Implementation (OSDI '96), Washington, October" 1996.

[79] Noble B. D. and Satyanarayanan M. (1999). *Experience with adaptive mobile applications in Odyssey.* Mobile Networks and Applications archive, Volume 4, Issue 4 , page 245-354. December 1999.

[80] Nwana H. S., Rosenschein J., Sandholm T., Sierra C., Maes P., and Guttman R. (1998). *Agent-mediated electronic commerce: issues, challenges and some viewpoints.* In Proceedings of the third International Conference on Autonomous Agents, 1998.

[81] Oaks S. and Wong H. (2000). *Jini In a Nutshell.* O'Reilly 2000.

[82] Olson J. S and Teasly S (1996). *Groupware in the Wild: Lessons Learned from a Year of Virtual Collocation.* CSCW '96. Proceedings of the ACM 1996 conference on Computer supported cooperative work, pages 419-42.

[83] Oppliger R. (2002). Internet and Intranet Security, Second Edition. Artech House © 2002, Chapter 5 - Cryptographic Techniques.

[84] Page J., Zaslavsky A. and Indrawan M. *A buddy model of security for mobile agent communities operating in pervasive scenarios.* In Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation - Volume 32 January 2004.

[85] Peine H. and Stolpmann T. (1997). *The Architecture of the Ara Platform for Mobile Agents.* First International Workshop on Mobile Agents, Berlin, Germany, April 7-8, 1997. Mobile Agents (MA'97).

[86] Perkins C.E. (1998). *Mobile networking through mobile IP*. IEEE Internet Computing 2 (January-February 1998) 58-69.

[87] Perry M., Agarwal D. (2003). *Collaborative Editing within the Pervasive Collaborative Computing Environment*. In Proceedings of the 5th International Workshop on Collaborative Editing, ECSCW 2003, Helsinki, Finland, September 15, 2003. LBNL-53769.

[88] Picco G., Murphy A. and Roman G.-C. (1999). *LIME: Linda Meets Mobility*. In Proceedings of the 21st International Conference on Software Engineering, pages 368–377, May 1999.

[89] Pittura E. and Samaras G. (1997). *Data Management for Mobile Computing*. Kluwer Academic Publishers, 1997.

[90] Postma A., Boer W. de, Helme A., and Smit G. (1996). *Distributed Encryption and Decryption Algorithms*. Memoranda Informatica 96-20, University of Twente, Enschede, December 1996.

[91] Powell M., and Miller B. (1983). *Process Migration in DEMOS/MO*. In Proceedings of the Ninth ACM Symposium on Operating Systems Principles (Bretton Woods, N.H., Oct. 11-13), ACM/SIGOPS, New York, 1983, pp. 110-119.

[92] Reichert F. (1996). *Middleware for a New Generation of Mobile Networks: The ACTS OnTheMove Project*. Inet 96 conference contribution, June 1996.

[93] Roman G.-C., Picco G. P. and Murphy A. M. (2000). *Software Engineering for Mobility: a Roadmap*. In Proceedings of the conference on the future of Software engineering, Limerick, Ireland. Pages 241-258. 2000.

[94] Roth J. (2001). *Information sharing with handheld appliances*. In the Proceedings of the 8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI'01), Toronto, Canada, May, 2001, (263-279).

[95] Roth J. (2003). *The Resource Framework for Mobile Applications*. In the Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS 2003), Angers, France, April 2003, 87-94, volume 4.

[96] Salz R., Leach P. and Mealling M. (2004). *A uuid urn namespace*. Network Working Group, Internet-Draft, January 2004. http://www.ietf.org/internet-drafts/draft-mealling-uuid-urn-03.txt.

[97] Samaras G. and Panayiotou C. *Personalized portals for the wireless user based on mobile agents*. In Proceedings of the 2nd international workshop on Mobile commerce. Atlanta, Georgia, USA. (70 - 74).

[98] Samaras G., Pitoura E. and Evripidou P. (1999). *Software Models for Wireless and Mobile Computing: Survey and Case Study*. Technical Report TR-99-5, University of Cyprus, March 1999.

[99] Sander T. and Tschudin C. F. (1998). *Protecting mobile agents against malicious hosts*. In Mobile Agents and Security, number 1419 in LNCS. Springer-Verlag, 1998.

[100] Satyanarayanan M. (1996). *Fundamental challenges in mobile computing*. In Proceedings of Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, ACM Press (1996).

[101] Schiller J. (2000). *Mobile Communications*. Addison-Wesley, Reading, Mass. and London, 2000.

[102] Stamos J. and Gifford D. (1990). *Remote evaluation*. ACM Trans. Computer System 12, 4 (Oct. 1990), 537-565.

[103] Sun Microsystems (1994). *The Java Language: An Overview*. Technical Report, Sun Microsystems, 1994.

[104] Sun Microsystems (1996). *Java Remote Method Invocation Specification.*

[105] Suri N., Bradshaw J. M., Breedy M. R., Groth P. T., Hill G. A., Jeffers R., and Mitrovich T.S. (2000). *An Overview of the NOMADS Mobile Agent System.* In Proceedings of European Conference on Object-Oriented Programming (ECOOP'2000), Nice, France, 2000.

[106] Suri N., Carvalho M., Bradshaw R. and Bradshaw J.M. (2002). *Small Mobile Agent Platforms*. In Proceeding of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS) 2002. Bologna Italy.

[107] Tan H. K. and Moreau L. (2001). *Mobile Code for Key Propagation.* In Proceedings of First International Workshop on Security of Mobile MultiAgent Systems (SEMAS'2001), pages 10. Fischer, K. and Hutter, D., Eds.

[108] Tan H.K., and Moreau L. (2002). *Certificates for Mobile Code Security*. In The 17th ACM Symposium on Applied Computing (SAC'2002) - Track on Agents, Interactions, Mobility and Systems, pages 76-81, Madrid, Spain, March 2002.

[109] Tarasewich P., Nickerson R., and Warkentin M. (2002). *An examination of the issues in mobile e-commerce*. Communications of the AIS 8, (2002), 41-64.

[110] Tardo J., Valente L. (1996). *Mobile Agent Security and Telescript*. In the Proceedings of IEEE COMPCON Spring '96, 58-63, 1996.

[111] Terry D. B., Theimer M. M., Petersen K., Demers A. J. (1995). *Managing Update Conflict in Bayou, a Weakly Connected Replicated Storage System.* In Proceedings of the fifteenth

ACM symposium on Operating systems principles, Copper Mountain, CO USA, Dec. 3-6, 1995, 172-182.

[112] Thai B., Wan R., Seneviratne A. and Rakotoarivelo T. (2003). *Integrated Personal Mobility Architecture: A Complete Personal Mobility Solution.* Mobile Networks and Applications (MONET) 8(1): 27-36 (2003).

[113] Thompson M., De Roure D., Michaelides D.T. (2000). *Weaving the Pervasive Information Fabric.* In Proceedings of Open Hypermedia Systems and Structural Computing, 6th International Workshop, OHS-6, 2nd International Workshop, SC-2, San Antonio, Texas, USA, May 30-June 3, 2000 Proceedings 1903, pages pp. 87-95. Reich, S. and Anderson, K. M., Eds.

[114] Tripathi A. R. and Karnik N. M. (2000). *Mechanisms for Delegation of Privileges to Mobile Agents in Ajanta.* In Proceedings of Internet Computing 2000 (IC2000), 379-385, Monte Carlo Resort, Las Vegas, June 26-29, 2000.

[115] Tripathi A., Koka M., Karanth S., Osipkov I., Talkad H., Ahmed T., Johnson D., and Dier S. (2004). *Robustness and Security in a Mobile-Agent based Network Monitoring System.* Technical Report 04-003. 2004.

[116] Varsheny, U., and Vetter, R. (2000) *Emerging mobile and wireless networks.* Communications of the ACM 43 (6), 73-81

[117] Varshney U. and Vetter R. (2002). *Mobile Commerce: Framework, Applications and Networking Support.* Mobile Networks and Applications, Volume 7, Issue 3, June 2002, Pages 185 - 198.

[118] Vigna G. (1998). *Cryptographic traces for mobile agents.* In Mobile Agents and Security, number 1419 in LNCS. Springer-Verlag, 1998.

[119] Vigna G. (2004). *Mobile Agents: Ten Reasons For Failure.* In Proceedings of MDM 2004 298-299 Berkeley, CA January 2004.

[120] Volpano D. and Smith G. (1998). *Language issues in mobile program security.* In Mobile Agents and Security, number 1419 in LNCS. Spriuger-Verlag, 1998.

[121] Vuong S.T. and Fu P. (2001). *A security architecture and design for mobile intelligent agent systems.* ACM SIGAPP Applied Computing Review, Volume 9, Issue 3 Fall 2001. (21 - 30).

[122] Waldo J. (1998). *Jini Architecture Overview.* Sun Microsystems, 1998.

[123] Waldo, J., *JavaSpaces, specification 1.0.* Technical report (March 1998), Sun Microsystems.

[124] Weiser M. (1991). *The computer of the 21st century.* Scientific American, pages 94--100,

September 1991.

[125] White J. (1995). *Telescript Technology: An Introduction to the Telescript Language.* General Magic White Paper GM-M-TSWP3-0495-V1, General Magic Incorporated, 1995.

[126] Wilhelm U. G., Staamann S. and L. Buttyan (1999). *Introducing trusted third parties to the mobile agent paradigm.* In Secure Internet Programming: Security Issues for Mobile and Distributed Objects, number 1603 in LNCS. Springer-Verlag, 1999.

[127] WLANA. What is a Wireless LAN? http://www.wlana.com/

[128] Wong D., Paciorek N. and Moore D. (1999). *Java-based mobile agents.* Communications of the ACM, Volume 42 , Issue 3 (March 1999).

[129] Wooldridge M. and Jennings N. R. (1995). *Intelligent agents: Theory and practice.* The Knowledge Engineering Review, vol. 10(2) pp. 115-152, 1995.

[130] Wyckoff P., McLaughry S. W., Lehman T. J., and Ford D. A. (1998). *T Spaces.* IBM Systems Journal 37, 3, 454-474. 1998.

[131] Yee B. S. (1999). *A sanctuary for mobile agents.* In Secure Internet Programming : Security Issues for Mobile and Distributed Objects, number 1603 in LNCS. Springer-Verlag, 1999.

[132] Yemini Y. and da Silva S. (1996). *Towards Programmable Networks.* In Proceedings of IFIP/IEEE International Workshop Distributed Systems: Operations and Management, L'Aquila, Italy, Oct. 1996.

[133] Zaini N., Moreau L. (2002). *Coordination of Mobile Intermediaries Acting on Behalf of Mobile Users.* In Proceedings of the 8th International Euro-Par Conference Paderborn, Germany. Page 973-977. 2002.

[134] Zaini N., Moreau L. (2002). *Mobile Intermediaries Supporting Information Sharing between Mobile Users.* In Proceedings of the 6th International Conference, MA 2002, Barcelona, Spain. Page 121-137. 2002.

[135] Zhang J., Helal A. and Hammer J. (2003). *UbiData: Ubiquitous Mobile File Service.* In Proceedings of the ACM Symposium on Applied Computing (SAC), Melbourne, Florida 2003.