Caching Components for Disconnection Management in Mobile Environments

Nabil Kouici, Denis Conan, and Guy Bernard

GET/INT, CNRS UMR SAMOVAR, 9 rue Charles Fourier, 91011 Évry, France
{Nabil.Kouici, Denis.Conan, Guy.Bernard}@int-evry.fr

Abstract. With the evolution of wireless communications, mobile handheld devices such as personal digital assistants and mobile phones are becoming an alternative to classical wired computing. However, mobile computers suffer from several limitations such as their display size, CPU speed, memory size, battery power, and wireless link bandwidth. In addition, service continuity in mobile environments raises the problem of data availability during disconnections. In this paper, we present an efficient cache management for component-based services. Our ideas are illustrated by designing and implementing a cache management service for CORBA components conducted on the DOMINT platform. We propose deployment and replacement policies based on several meta-data of application components. A novel aspect is the service-oriented approach. A service is seen as a logical composition of components cooperating for performing one functionality of the application. Dependencies between services and between components are modelled in a hierarchical dependency graph.

Key words: Mobile computing, disconnection, cache management, component-based middleware.

1 Introduction

Since 1990, more and more progress has been done in computer networks and machines used in distributed environments. Computer networks are becoming increasingly heterogeneous ranging from fixed high-end machines to mobile low-end machines like mobile phones and personal digital assistants (PDA). This evolution has opened up new opportunities for mobile computing. For example, a user with a mobile device can access various kinds of information at any time and any place. However, mobile computing suffers from several limitations: the mobile terminals are limited in terms of CPU speed, memory size, battery power and wireless link bandwidth. Wireless connection is more expensive than wired connection and it is characterised by frequent disconnections.

In such environments, disconnection is a normal event and should not be considered as a failure freezing the application. We distinguish two kinds of disconnections: voluntary disconnections when the user decides to work on their

own for saving battery or communication costs, or when radio transmissions are prohibited as aboard a plane, and involuntary disconnections due to physical wireless communication breakdowns such as in an uncovered area or when the user moves out of the reach of base stations. We also consider the case where the communication is still possible but not at an optimal level, resulting from intermittent communication, low-bandwidth, high-latency, or expensive networks. Furthermore, with the connectivity variation in space and in time, the mobile terminal may be *strongly connected* (connected to Internet via a fast and reliable link), *disconnected* (no network connection at all to Internet), or *weakly connected* (connected to Internet via a slow link) [20].

The adaptation to the characteristics of mobile environments can be performed by the application (*laissez-faire* strategy), by the system (*transparent* strategy), or by both the application and the system (*collaboration* strategy) [25]. As surveyed in [12], there is much work dealing with mobile information access that demonstrates that the *laissez-faire* and the transparent approaches are not adequate. Our collaboration approach is then twofold. Firstly, we use caching to obtain work continuity while being disconnected. Secondly, the application must be built in such a way that it specifies the behaviour while being disconnected. This is achieved by using some meta-data to specify application's components and functionalities: which components or functionalities can be cached and which ones must be present for the disconnected mode.

These ideas are illustrated by designing and implementing a cache manager service for CORBA components conducted on DOMINT [6]. Furthermore, unlike file caching where no interactions between files occur, we maintain in memory components that are connected with other local components. Thus, the solution must take components' dependencies into account. A novel aspect of this paper is the use of a service-oriented approach. A service is seen as a logical composition of components cooperating for performing one functionality of the application. Dependencies between services and between components are modelled in a hierarchical dependency graph. The DOMINT platform deal with the reconciliation (also called data synchronisation) of discomponents after the reconnection ; since the paper doesn't present that issue, please refer to [6].

The remainder of this paper is organised as follows. Section 2 gives our motivations for disconnection management. Section 3 gives a classification of application's entities according to some criteria. The methodology for determining and manipulating the dependency graph is described in Section 4. In Section 5, we describe the cache deployment strategy and the cache replacement strategy. The implementation of the cache manager and first experimental results are presented in Section 6. Section 7 compares our approach with related work, and finally, Section 8 summaries the paper, presents conclusions, and discusses future research issues.

2 Motivations and objectives

Traditional programming environments are mainly connection-oriented programming environments in which a client must maintain a connection to a server. In mobile computing, the challenge is to maintain this logical connection between a client and its servers using the concept of disconnected operation [14]. A disconnected operation allows clients to use services when the network connection between the mobile client and the server is unavailable, expensive, or slow. Hence, mobile terminals must cache some data or even some code from remote servers so that clients in mobile terminals use these data while being weakly connected or disconnected.

Three important issues exist in designing an effective cache management. First of all, the deployment strategy determines what to cache, when and for how long. Secondly, the replacement strategy computes which entity should be deleted from the cache when the cache does not have enough free space to add a newly-required entity. Finally, the consistency strategy maintains data consistency between data in the cache and data in the original server. In this paper, we do not address consistency issues. In a mobile application, the distribution of application entities can be done in fixed terminals [26, 21, 13], or in fixed and mobile terminals [30]. In the first case, the client's GUI in the mobile terminal uses the server parts installed in fixed hosts. In the second case, a mobile terminal can be a client for servers and can be a server for other hosts (mobile or fixed). This last case was rarely studied in mobile environments because of the limited capacity of mobile terminals and because of the difficulty in implementing these applications with traditional object-oriented, database-oriented, and file-oriented programming paradigm.

The development of distributed applications converges more and more towards the use of component-oriented middleware such as EJB [7], CCM [23] and .Net [19] that better addresses the application complexity by separating functional and extra-functional aspects [29]. Unfortunately, these middleware are inadequate for mobile environments where the resources are unstable. In addition, components are seen as independent pieces of software that can be assembled to realise complex software. These components cooperate with each other to accomplish system functionalities in a distributed manner. Thus, service and component dependencies must be managed and made implicitly during execution.

3 Application's service and profile

In this Section, we define the concept of service of mobile distributed applications and propose an application's profile for service continuity.

A distributed application can be viewed as a set of components. They use and provide functionalities that are accessed through connections between components. The functionality of a multi-component application is accessed through a component that itself can use some parts of the functionalities offered by others. This interaction fulfils a function that may be described as the provision of

a service. A set of components that interact with each others to achieve a functionality is defined as a *logical composite component*, that is, according to [3], a service is defined as "a contractually defined behaviour that can be implemented and provided by any component for use by any component, based solely on the contract". The application as a whole may be regarded as a set of services which are accessed by users through a GUI acting as a "Façade" (design pattern) [9]. Thus, we define two types of interactions: intra-service (between components in the same service) and inter-service (between services). For example, an Internet travelling agency application may be regarded as a set of services (booking, getting prices, canceling... reservations) and each service is realised by collaborations between several components. For example, the service "booking a seat in plane" uses a component to get available seats and another one to obtain prices.

In [16], we have introduced a meta-model for designing applications that deal with disconnections. This meta-model is based on meta-data that define an application profile. The disconnectability meta-data indicate whether a component residing on a fixed server can have a proxy component on a mobile terminal that we call a *discomponent*. If this is the case, the original component is said to be disconnectable. A discomponent achieves the same functionalities as the component in the fixed server, but is specifically built to cope with disconnection and weak connectivity. The design of a discomponent from the corresponding remote component is an open issue not treated in this paper. We are currently devising and experimenting design patterns and idioms for that construction. Software architects set the disconnectability meta-data since they have the best knowledge of the application's semantics. Furthermore, disconnectability implies design constraints that the developers must respect. For example, for security reasons, one may decide to deploy some components on dedicated secure hosts and to prevent clients from loading them on mobile hosts, thus not allowing the disconnectability of these components.

Next, the necessity meta-data indicate whether a disconnected component must be present on the user terminal. Clearly, the necessity applies only on disconnectable components. The necessity is specified both by application's developers and end-users. The former stake-holders provide a first classification in developer-necessary and developer-unnecessary components, and the latter stake-holders can overload a developer-unnecessary component to be user-necessary at runtime.

Finally, the priority meta-data indicate the priority between unnecessary components and between user-necessary components. The priority is needed in order to select cached components when the cache size of the mobile terminal is too small.

By analogy, we apply these meta-data to the concept of service. Thus, we define a disconnectable service as being a service which can be performed in the mobile terminal during disconnection and which is the logical composition of several disconnectable components. In addition, we define a necessary service as a service that contains at least one necessary component. Also by analogy, services are given priorities.

4 Dependency graph

As just described, an application as a whole may be regarded as a set of services. In the software architecture, these services are identified in use cases diagrams. Architects specify which use cases are disconnectable and necessary during a disconnection, and their corresponding priority. This Section first presents the design of the dependency graph and then the propagation of the meta-data within this dependency graph.

4.1 Design of dependency graph

Our collaborative approach for dealing with disconnections is expressed in a development process called Mobile Application Development Approach (MADA) [17] that is model-driven, architecture-centric, and component-based, and which follows the Model-Driven Approach (MDA) of the Object Management Group (OMG). MADA is based on the "Façade" design pattern [9] and the "4+1" view model [18]. The "Façade" design pattern allows to simplify the access to a related set of services by providing a single entry point to call services, thus, reducing the number of components presented to the user. The "4+1" view model makes possible the organisation of the software architecture in multiple concurrent views (logical, process, physical, development, and use cases). Each one addresses separately the concerns of the various stake-holders of the software architecture. In addition, it helps in separating functional and extra-functional aspects.

Software architects specify which services are disconnectable in the use cases diagram, and for each disconnectable service, an extended use case (using the `extend` dependency) is defined replacing the original use case during disconnection. Then, they tag use cases with the necessity meta-data and give a priority to each service. Finally, for each disconnectable service, they provide the necessity and the priority for components that collaborate to perform this service by using classes and collaborations diagrams.

Figure 1 depicts a simplified use cases diagram for an Internet travelling agency application that we have used as an example application test-bed. Figure 1-a depicts services offered by the application and Figure 1-b describes the same application with disconnection management. The service *"Buy a ticket"* is set non disconnectable for security reasons, whereas the other services are disconnectable. In addition, the service *"Book a ticket"* is necessary to ensure the optimal service continuity in disconnected mode. The service *"Book a ticket"* uses the service *"Prices"* that can also be used directly by the user. Thus, the use of the service *"Book a ticket"* while being disconnected requires the presence of the service *"Prices"* in the cache. Solving this issue requires the determination and computation of dependencies between services. These dependencies are presented within a directed graph where nodes denote services and edges denote the `include` dependency which is annotated with the necessity meta-data.

Service availability in disconnected mode implies the presence of some components which are used for achieving this service. Thus, by analogy, component

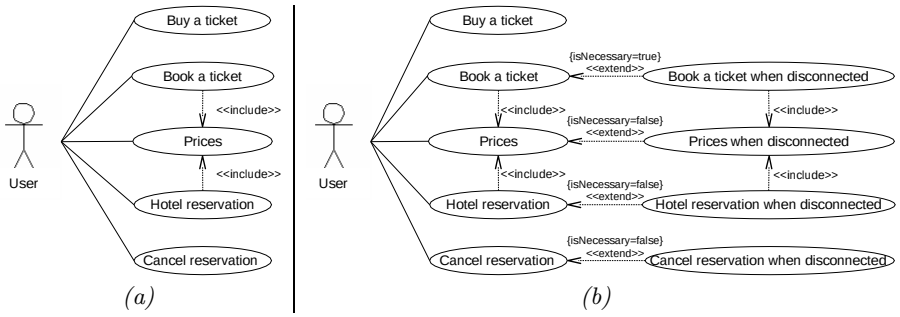


Fig. 1. *(a)* Use cases diagram *(b)* Use cases diagram with disconnectable use cases.

dependencies are also drawn within the dependency graph where nodes denote components and edges denote dependencies between components. Figure 2 depicts a simplified dependency graph for our example application. The "Façade" component represents the component accessed by the GUI. Thus, the dependency graph comprises three types of interactions: between the "Façade" and services, between the services and components, and between components. The last two types present the entry point to perform service functionality. In addition, it is clear that components can be used by different services (*e.g.*, *"Price-Provider"* component in Figure 2). The dependency graph is used by the deployment strategy (cf. Section 5.1) and the replacement strategy (cf. Section 5.2).

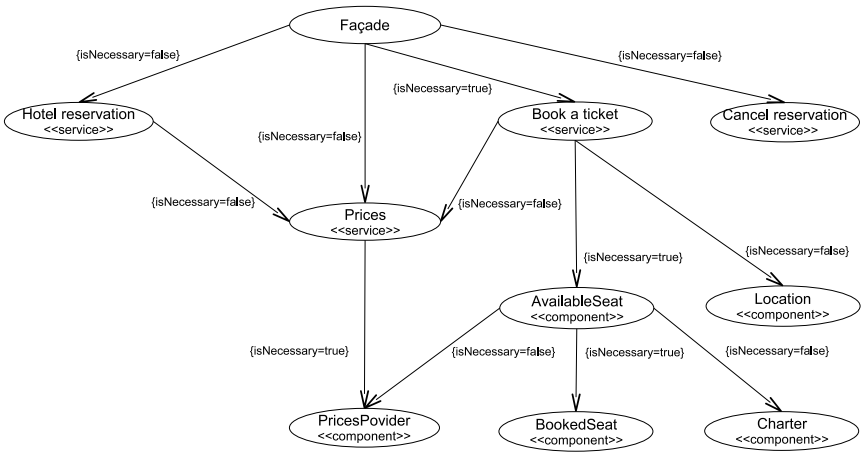


Fig. 2. Application's dependency graph.

4.2 Meta-data propagation

As described in Section 3, the end-user can overload some meta-data. These overloads lead to a propagation of the meta-data intra- and inter-services. In this paper, we only detail the propagation of the necessity. For the sake of clarity, in a dependency relation, we use the prefix "pre-" to express the starting entity (pre-component and pre-service) and the prefix "post-" to express the target entity (post-component and post-service). We describe the meta-data propagation from the `Façade` component (the root of the graph) to components at the leafs of the graph.

As described in Section 4.1, end-users are only aware of services. In addition, the necessity of a service corresponds to the necessity of the edge between the "Façade" component and this service. Let $\mathcal{S}$ be the set of services, F the "Façade" component, $\mathcal{L}$ the set of edges, *necLFC(l)* a predicate evaluating to true if edge l between F and s is necessary, and *necs(s)* a predicate evaluating to true if service s is necessary. The previous statement is written as follows:

$$\forall s \in \mathcal{S} : (\exists\ l_{F \longrightarrow s} \in \mathcal{L} \wedge necLFS(l_{F \longrightarrow s})) \implies necs(s) \tag{1}$$

If the edge between two services is tagged as being necessary and the pre-service is also necessary, then the post-service becomes necessary. Let *necLS(l)* be a predicate evaluating to true if edge l is necessary between services. The previous statement is written as follows:

$$\forall s_1, s_2 \in \mathcal{S}, \forall l_{s_1 \longrightarrow s_2} \in \mathcal{L} : necLS(l_{s_1 \longrightarrow s_2}) \wedge necs(s_1) \implies necs(s_2) \tag{2}$$

In addition, the necessity will be propagated from services to components. If the service is necessary and the edge between a service and a component is tagged as being necessary, then the component becomes necessary. Let $\mathcal{C}$ be the set of components, *necLSC(l)* a predicate evaluating to true if edge l between a service and a component is necessary, and *necc(c,s)* a predicate evaluating to true if component c is necessary for service s. The previous statement is written as follows:

$$\forall s \in \mathcal{S}, \forall c \in \mathcal{C} : (\exists\ l_{s \longrightarrow c} \in \mathcal{L} : necLSC(l_{s \longrightarrow c})\ \wedge\ necs(s)) \implies necc(c, s) \tag{3}$$

By analogy with services, let *necLC(l)* be the predicate evaluating to true if edge l between two components is necessary for service s. The necessity of the pre-component and the necessity of the link between the two components imply the necessity of the post-component for service s. The previous statement is written as follows:

$$\forall s \in \mathcal{S}, \forall c_1, c_2 \in \mathcal{C}, \forall l_{c_1 \longrightarrow c_2} \in \mathcal{L} : necLC(l_{c_1 \longrightarrow c_2}) \wedge necc(c_1, s) \implies necc(c_2, s) \tag{4}$$

Finally, it is clear that a component is necessary if and only if it is necessary in at least one necessary service. The previous statement is written as follows:

$$\forall c \in \mathcal{C} : (\exists s \in \mathcal{S}\ :\ necc(c, s)\ \wedge\ necs(s)) \iff necc(c) \tag{5}$$

In Figure 2, if the end-user tags the link between the "Façade" component and the service *"Book a ticket"* as necessary, the necessity will be propagated to the service *"Book a ticket"* by (1), and then the necessity will be propagated to the component *"AvailableSeat"* by (3). Finally, the necessity will also be propagated to the component *"BookedSeat"* by (4).

5 Cache manager

We describe the deployment strategy in Section 5.1 and the replacement strategy in Section 5.2. Both strategies are based on the meta-data introduced in Section 3 and on the application dependency graph described in Section 4.

5.1 Deployment strategy

We define three complementary deployment times. At launching-time, developer-necessary services are all deployed and cached. We assume that the cache size is higher than the size of all the developer-necessary services and the application starts only if the loading of developer-necessary services is successfully performed. User-necessary services are then deployed depending on their priority. Deploying a service corresponds to deploying the necessary components of this service. However, before creating a discomponent into the cache, the cache manager checks whether a discomponent has already been deployed for other services since the cache is shared between the applications running on the mobile terminal. One potential drawback of this strategy is that the end-user must wait till all developer-necessary discomponents of developer-necessary services are deployed before beginning to work.

During execution, the end-user can specify which services should be deployed locally for disconnection management. This is *end-user-demand* deployment in which the end-user is presented with a list of services offered by the application and their meta-data (disconnectability, necessity, and priority). The end-user can use this deployment type for example before a voluntary disconnection. As described in Section 4.2, the change of service necessity (from unnecessary to user-necessary) can impact the necessity of other services. Thus, the cache manager updates the dependency graph and deploys the new user-necessary services.

At invocation-time, when a client on the mobile terminal requests a user-necessary service or an unnecessary service, the cache manager checks whether the service is already deployed in the cache. If not, the service is deployed and may replace some services already in the cache (following the replacement strategy). This deployment type is also used when a discomponent in the cache requests a component not cached, yet. In both cases, the cache manager also deploys services or components needed according to the dependency graph.

5.2 Replacement strategy

When a new entity (service or component) should be placed in the cache, if the cache exceeds its capacity, some entities should be ejected in order to make some room for the newcomers. The replacement strategy plays the key role of determining which entities to eject. We define two cases in the replacement strategy.

In the first case, called *end-user-demand* replacement, the end-user can specify which services should be evicted from the cache when there is not enough memory size. This is realised by presenting the user with a list of currently cached services. Since developer-necessary services are mandatory, the list is only made of user-necessary services or unnecessary services. To evict a service from the cache, the replacement strategy executes Algorithm 1. The removal of the service fails if this service is developer-necessary (line 4). From line 8, a discomponent is evicted from the cache if it is not in conflict[1] with the component in the fixed server, and if it is not used by other services in the cache. In addition, to avoid having orphan components, components in conflict will be evicted once reconciled.

Algorithm 1: `Boolean` **serviceRemoval**(`Service` *sr*)

```
1  boolean necessary ← getServiceNecessity(sr) {true if the service is necessary}
2  string necessityKind ← getNecessityKind(sr)        {"User" or "developer"}
3  DisconnectedComponent dc
4  if necessary and necessityKind="developer" then
5     return false                                {Exit without removing the service}
6  ComponentSet components ← getComponentSet(sr)
7  for all dc ∈ components
8    if (dc.updated and ¬dc.shared) then removeComponent(dc)
9  removeService(sr)
10 return true                  {Exit with success after suppression of the service}
```

In the second case, called *periodic* replacement, the replacement process is executed periodically in order to try keeping a part of the cache free for critical use to anticipate and thus accelerate the deployment of new services or components. The size of the critical part of the cache is configurable by the end-user. Algorithm 2 gives the functioning of the *periodic* replacement. The cache manager obtains the set of cached services, the size of the free memory of the cache and the size of the critical memory of the cache. While the free memory size is lower than the critical memory size, the cache manager executes for each service Algorithm 1 to release some memory space. In addition, the removal of services is performed according to a replacement policy (line 2).

[1] In conflict means that operations performed locally have not yet been executed on the remote component.

Algorithm 2: Periodic_strategy()

```
1  Collection services ← getServices()          {All services loaded in the cache}
2  orderServices(policy, services)              {Order according to a policy}
3  Service sr ← services.getFirst()             {Obtain the first service}
4  while (getFreeCacheSize() < getCriticalSize())
5    serviceRemoval(sr)                         {Apply Algorithm 1}
6    sr ← services.getNext()                    {Obtain the next service}
7    if sr = null then break
```

As a new replacement policy, we propose the LFUPP (*Least Frequently Used with Periodicity and Priority*) policy which is an improvement of the basic LFU (*Least Frequently Used*) policy. When a service is to be removed from the cache, the one with the lowest frequency is selected. If there are several services whose frequency is the lowest, one of them is selected according to their priority. To avoid having services with a larger frequency due to scattered bursts of accesses, LFUPP periodically resets to 0 the frequency.

6 Implementation and performance measurements

We present the implementation of the cache manager in Section 6.1 and some performance measurements in Section 6.2 to evaluate the efficiency of our proposition.

6.1 Cache manager service

The cache manager is a CORBA service and is integrated in the component-oriented middleware OpenCCM [22] conducted on DOMINT [6]. DOMINT is a platform which adapts distributed component-based applications so that they ensure service continuity even while being weakly connected or disconnected. In addition, the cache manager service is modelled and implemented using the Fractal component model [22].

Figure 3-a describes the `CacheManager` component architecture. The cache manager component is a composite of four Fractal components. The `DisComponentFactory` component represents the entry point of the `CacheManager` component. It coordinates the deployment and the management of the services and the their discomponents. The `DisComponentCreator` component allows creating discomponents. In the context of CORBA, it uses the OpenCCM deployment tool. The `DiscEntryFactory` component allows creating a cache entry per discomponent. An entry is composed of an object that encapsulates the CORBA reference of the remote component, the CORBA reference of the discomponent, and the meta-data used by the replacement strategy. The `PerseusCacheManager` component, from the ObjectWeb Perseus project [22], gathers existing components that we have reused in order to realise our replacement strategy. The structure of the `PerseusCacheManager` is depicted in

Figure 3-b. The `DiscReplacementManager` component extends the (abstract) `ReplacementManager` component of Perseus in order to integrate a new replacement policy.

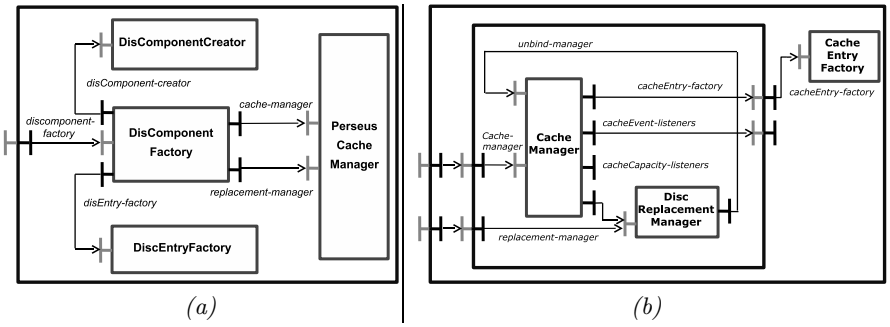


Fig. 3. *(a)* Cache manager component *(b)* Perseus component.

The dependency graph presented in Section 4 is implemented using GXL (*Graph eXchange Language*) [11], which is an XML based notation. Our main motivation for using GXL is the presence of a flexible and extensible mechanism to define a notation for the description of services, components, and their interactions. In addition, GXL allows modelling hierarchical graphs. We have extended the GXL implementation which is freely available to take our meta-data into account in the parsing process.

6.2 Performance measurements

We have performed some experiments in order to evaluate the efficiency of our propositions. We focus firstly on the amount of time required to extract meta-data of services and components from the dependency graph, and then evaluate how well our replacement strategy performs.

Figure 4 shows the average time to extract the meta-data from the dependency graph. This test was run on GNU/Linux RedHat 9.0 powered by a 933 MHz Pentium 3 with 528Mo RAM. Each test was executed 1000 times in order to obtain meaningful averages. A garbage collection occurred before each execution in order to have no interference with previous operations. For 1 service and 20 components, the time to extract the meta-data from the dependency graph is 1.32ms, and for 20 services and 381 components, it takes 20.2ms. The results show that the time to extract the meta-data remains very low even for extreme situations with tenths of services and hundreds of components. Of course, these extreme situations are unreasonable for mobile terminals. In addition, the number of services does not have an influence over the execution time, and with a high number of components, the overhead increases slightly because of swapping (which does not exist right now in most of the mobile terminals).

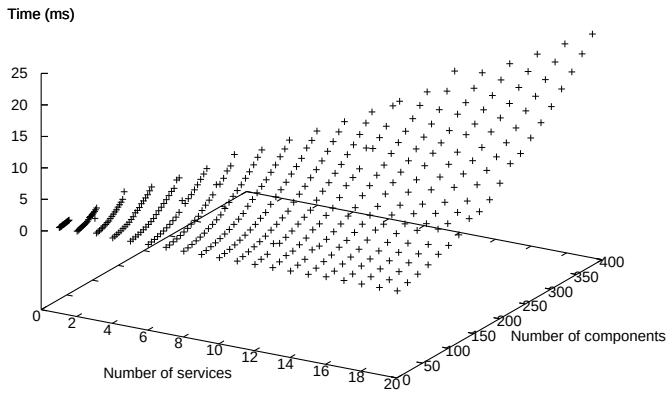


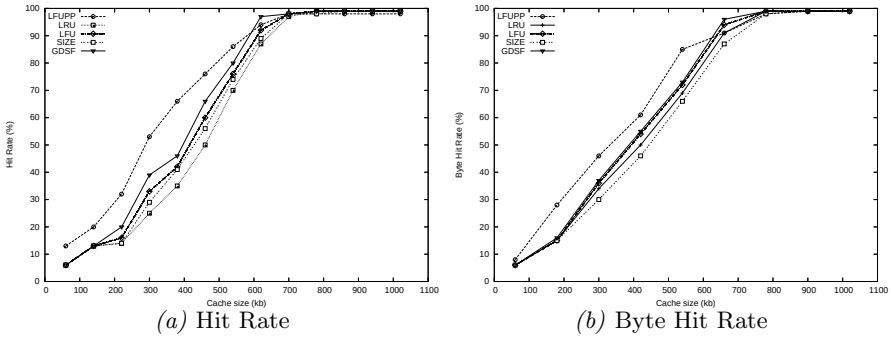
Fig. 4. Deployment time

In order to evaluate how well our replacement strategy performs, we use two performance metrics: *Hit Rate* (HR) and *Byte Hit Rate* (BHR). HR is the ratio between the number of requests satisfied by the cache and the total number of requests. BHR represents the percentage of all the data size that is used from the cache rather than from the original server. In addition, we calculate HR and BHR for two cases: the granularity of the replacement strategy is either the component or the service.

We have conducted the experiments using a simple application with 15 services (5 of them are necessary) and 50 components on a laptop PC (Intel 700 MHz Pentium 3, 128M RAM) running Microsoft Windows2000. In addition, we have implemented a simulator that artificially generates access traces. It takes into account the number of components, the number of services, the size of the components, the necessity, the priority, and the number of requests. Each test was run 10000 times, the range of component's (*resp.* service's) sizes is 10–160Kb (*resp.* 50–160Kb) with an average component's (*resp.* service's) size of 45Kb (*resp.* 80Kb). The size of a service corresponds to the sum of the size of necessary components of this service. In addition, for the component access traces, we have used a trace with 80% of requests referencing necessary components, 10% of requests referencing components with high priority (necessary or unnecessary), and 10% of random traces. For the services, we have used the same percentages as for the component access traces.

Our experiments consist to evaluate the efficiency of Algorithm 1 and Algorithm 2 studied in Section 5.2. We examine five replacement policies: two "traditional" policies (Least Frequently Used and Least Recently Used), two

replacement policies investigated in the WWW (Greedy Dual Size with Frequency [4] and SIZE [31]), and LFUPP described is Section 5.2. Figure 5 and Figure 6 compare the average *hit rate* and the *byte hit rate* achieved by each policy using respectively the component and the service as the access unit, and for cache sizes ranging from 60Kb to 1020Kb.

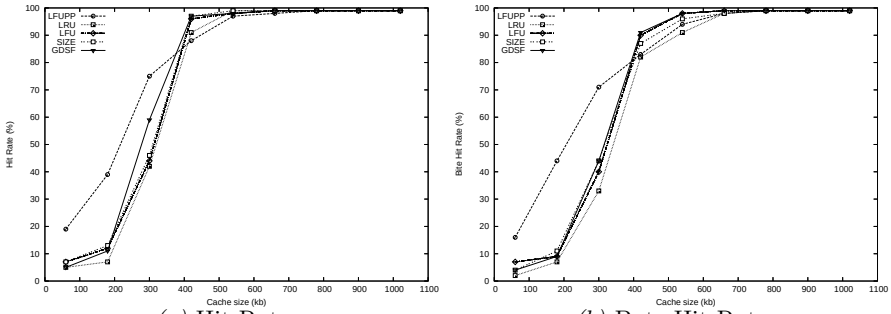


(a) Hit Rate *(b)* Byte Hit Rate

Fig. 5. Analysis of policies with the component as the access unit.

According to Figure 5, LFUPP is the best choice for the small cache sizes (60Kb–500Kb). It outperforms all other policies by at least 7% for the *hit rate*, and between 2% and 12% for the *byte hit rate*. This success can be attributed to the following factors. Clearly, we assume that a well-designed application may contain a reasonable percentage of necessary components access traces (we consider 80%), therefore favouring LFUPP. Similarly, the other meta-data considered, namely the priority, help in making best choices. For large cache sizes, all the policies perform roughly the same with a small advantage for GDSF. However, although SIZE treats more favourably small components, our results show that this policy has the worst *byte hit rate* despite the inclusion of both large and small components in the simulated traces. LRU achieves the lowest *hit rate* since it does not consider enough information in the replacement process, in particular, the component priority.

Figure 6 indicates that using the service as the access unit, LFUPP is superior to other policies for small cache sizes (60Kb–340Kb). However, for large cache sizes, LFUPP is roughly the worst. Finally, the results obtained with the service as the access unit converge rapidly compared to the results obtained using the component as the access unit. This is because a service request is only performed when all necessary components of this service are in the cache. In addition, in practice, once cached, the service will often be accessed in burst. Therefore, using the service as the access unit is more efficient than the component for devices with small memory size.



(a) Hit Rate *(b)* Byte Hit Rate

Fig. 6. Analysis of policies with the service as the access unit.

7 Discussion and related work

Caching is a fundamental paradigm for ensuring service continuity while being disconnected. This issue is studied in different fields such as file system, WWW, database, object-oriented system, and component-oriented system. Caching has also been highly investigated in operating system virtual memory management. Various page replacement algorithms have been proposed including *NRU* (*Not Recently Used*), *FIFO* (*First In, First Out*), *LRU* (*Least Recently Used*), and *LFU* (*Least Frequently Used*) algorithms. These and other algorithms are surveyed in [28].

In the file system field, Coda [27] defines the notion of implicit and explicit data which are used as an application profile to choose the files to prefetch and replace. The implicit data are composed of the client history and the explicit data take the form of a client database called HDB (*Hoard Data Base*). The cache manager called *Venus* can be in three modes: hoarding, emulation, and reintegration. In the hoarding mode (strongly connected), Coda anticipates disconnections by locally storing files. It deploys the files in the HDB according to a priority given by the user. The other files are loaded after a cache miss and are managed using a LRU replacement policy. However, an application may not be usable during disconnections if the end-user has made bad choices when filling the HDB. In our approach, we have added the developers' choices since they have the best knowledge of the application semantics and functioning. Using a cache system like Coda, Seer [10] adds predictive file hoarding to automatically detect which files to load. The granularity of the caching is a *project*, that is a group of closely related files. It uses the notion of *semantics distance* to quantify the relationship between files. Seer observes the end-user behaviour, calculates the distances, and automatically generates the corresponding projects and uses these projects to prepare the cache for disconnection. Even though the project defines a logical dependency between files, Seer does not define relationships between projects. In Amigos NFS layer [1], the cache manager deploys files on the mo-

bile terminal using a user-defined profile. This profile orders files and directories thanks to a user-assigned priority like in Coda. Periodically, Amigos revalidates the cache contents, purges dirty files, and updates the list of files to be deployed according to the user profile. In our work, the end-user collaboration for the disconnection management could be envisioned to be performed dynamically, and priority meta-data could be refined using collected statistics like in Seer.

In the WWW field, caching and prefetching are used for improving the performance of Internet accesses. *SIZE* [31] replaces the largest document by a bunch of small ones. However, some documents can be brought into the cache and never requested again. *GDSF* [4] assigns for each page P a key $\mathcal{K}(P)$ and when a replacement is needed, the page with the lowest key value is replaced. The key is calculated according to the function $\mathcal{K}(P) = \mathcal{L} + \mathcal{F}(P) * \mathcal{C}(P)/\mathcal{S}(P)$ where $\mathcal{L}$ is an aging factor that starts at 0 and is updated to the key value for the last replaced document, $\mathcal{F}(P)$ is the access count of page P, $\mathcal{C}(P)$ is the cost to bring page P into the cache and $\mathcal{S}(P)$ is the page size. However, *SIZE* and *GDSF* policies do not take into account users' preferences nor developers' ones.

In the database field, Bayou [30] provides a framework for highly-available mobile databases in the context of collaborative applications. Bayou uses a whole database as the caching granularity. Thus, each mobile terminal holds full replicas of databases. Bayou takes application's semantics into account to detect and resolve conflicts using a peer-to-peer anti-entropy algorithm [24]. However, mobile terminals that cannot hold the full replica of a database cannot offer service continuity during disconnections. [8] describes a client caching mechanism for a data-shipping database in which clients and servers interact using fixed-length physical units of data such as pages (four or eight Kbytes). Client caching mechanism is based on a dynamic replication mechanism in which page copies are created and destroyed based on the runtime demands of clients. This mechanism does not anticipate disconnections and does not take into account application's semantic.

In the object field, Rover [13] introduces two concepts: Relocatable Dynamic Objects (RDO) and Queued Remote Procedure Call (QRPC). Rover imports objects into the cache using RDO at the first invocation without taking application's semantics into account, and programmers must design and code their applications in terms of RDO. CASCADE [5] is a generic caching service for CORBA objects. Cached copies of each object are organised into a hierarchy. Clients always use objects from the nearest server. [2] describes two replacements policies used in CASCADE: *H-BASED* and *LFU-H-BASED*. In *H-BASED*, for each object in the hierarchy, the replacement key is the number of direct descendents that were evacuated from the cache. When the cache is full, the cache manager will evict the object with the smallest key. In addition to *H-BASED*, *LFU-H-BASED* associates a priority for each object in the cache. When an object must be removed from the cache, the one with the lowest priority is chosen. If there are several objects with the lowest priority, the *H-BASED* policy is used. However, CASCADE is not designed for disconnection management but to improve response time.

In the component field, ACHILLES [15] is a system for on-demand delivery of software from stationary servers to mobile clients. The granularity used in deployment and replacement strategies is a software element (whole application or single component). Users have the choice between two deployment policies: *automatic* and *manual*. In both cases, only the local copies are used. In the *manual* mode, the user specifies which software elements should be in the cache permanently and which ones can be removed if necessary. In the *automatic* mode, software elements are deployed locally at first use if there is no copy. The *automatic* policy uses a *Minimal Cost* strategy. The cost is used to determine which software element should be removed. It is a function of the cost to reload a software element once it has been removed and the importance of the software element. The importance of the software element is the number of software elements that depends on the former. Similarly to our work, ACHILLES uses a resource dependency graph to calculate the importance of software elements. As far as cache management for disconnection handling is concerned, ACHILLES does not handle involuntary disconnections and the *Minimal Cost* policy does not take into account the priority of software element according to the end-user's choice.

8 Conclusion

The purpose of this work is to investigate the problem of disconnection in mobile environments and to provide a platform for keeping working while being disconnected. A novel aspect of this paper is the service-oriented approach for the cache management. A service is seen as a logical composition of several components which cooperate to perform a functionality of the application. We have proposed the use of meta-data to build an application profile for managing the cache. The disconnectability meta-data indicate whether an entity can have a proxy on the mobile terminal, the necessity meta-data specifies whether the presence of the proxy on the mobile terminal is mandatory for the execution of the application during a disconnection, and the priority meta-data is used to select cached entities when the cache size is too small.

We have proposed an approach to analyse and manage dependencies in which intra-service and inter-services relationships are modelled in a dependency graph. The structure of the graph is static whereas the annotations of nodes and edges is dynamic. Based on the dependency graph and the application's profile, we have designed and implemented a cache manager service. We have defined the deployment strategy and a generic replacement strategy depending on the necessity. We have investigated several replacement policies to evaluate the replacement strategy. Our performance results show that the LFUPP (*Least Frequently Used with Periodicity and Priority*) policy described in this paper performs better when the cache size is small, which is the case for mobile terminal, and GDSF (*Greedy Dual Size with Frequency*) [4] provides a somewhat better *hit rate* and *byte hit rate* when the cache size is large. In addition, using the service as the replacement unit is more efficient than the component.

As future work, we plan to extend our approach in several points. First of all, our work currently assumes that the cache size is large enough to deploy developer-necessary services. We are currently investigating this limitation. According to the results studied in Section 6, we believe that taking into account the priority meta-data in the cost function of GDSF can achieve more exciting results in the case of small cache size. In addition, factors like bandwidth, connection establishing time between client and server, and deployment time can be considered in the cost function. The dependency graph is built once during the application development process. We are designing solutions to make it evolving depending on current resources availability. Finally, end-user collaboration for the disconnection management can be envisioned to be performed dynamically using a predictive approach like in Seer [10].

References

1. B. Andersen, E. Jul, F. Moura, and V. Guedes. File System for Semiconnected Operation in AMIGOS. In *Proc. 2nd USENIX Symposium on Mobile and Location-Independent Computing*, Dec. 1994.
2. H. Atzmon, R. Friedman, and R. Vitenberg. Replacement Policies for a Distributed Object Caching Service. In *Proc. International Symposium on Distributed Objects and Applications*, pages 661–674, California, Irvine, USA, Oct. 2002.
3. G. Bieber and J. Carpenter. Introduction to Service-Oriented Programming. http://www.openwings.org, 2002.
4. L. Cherkasova. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. Technical report, HP Labs, Palo Alto, Nov. 1998.
5. G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a caching service for distributed CORBA objects. In *Proc. 2nd IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 1–23, 2000.
6. D. Conan, S. Chabridon, L. Chateigner, N. Kouici, N. Sabri, and G. Bernard. DOMINT: Disconnected Operation for Mobile INternetworking Terminals. In *Poster of the 2nd ACM International Conference on Mobile Systems, Applications, and Services*, Boston, Massachussetts, USA, June 2004.
7. L. DeMichiel. *Enterprise JavaBeans Specifications, version 2.1, proposed final draft*. Sun Microsystems, http://java.sun.com/products/ejb/docs.html, Aug. 2002.
8. M. Franklin. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Transactions on Database Systems*, 22(3):315–363, Sept. 1997.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
10. H. Geoffrey and J. Gerald. Automated Hoarding for Mobile Computers. In *Proc. 16th Symposium on Operating Systems Principles*, pages 264–275, 1997.
11. R. Holt, A. Schurr, S. Elliott, and A. Winter. GXL home page. http://www.gupro.de/GXL/, 2002.
12. J. Jing, A. Helal, and A. Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, 31(2), June 1999.
13. A. Joseph, J. Tauber, and M. Kaashoek. Mobile computing with the Rover toolkit. *ACM Transactions on Computers*, 46(3), 1997.

14. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 213–225, Pacific Grove, USA, May 1991.
15. G. Kortuem, S. Fickas, and Z. Segall. On-Demand Delivery of Software in Mobile Environments. In *Proc. 11th IPPS Workshop on Nomadic Computing*, Apr. 1997.
16. N. Kouici, D. Conan, and G. Bernard. Disconnected Metadata for Distributed Applications In Mobile Environments. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, June 2003.
17. N. Kouici, N. Sabri, D. Conan, and G. Bernard. MADA, a Mobile Application Development Approach. In *Proc. Ubiquity and Mobility*, Nice, France, June 2004. ACM Press. In French.
18. P. Kruchten. Architectural Blueprints: The 4+1 View Model of Software Architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.
19. Microsoft. Microsoft Developer Network. http://www.msdn.microsoft.com.
20. L. Mummert. *Exploiting Weak Connectivity in a Distributed File System*. PhD thesis, Carnegie Mellon University, Pittsburg, USA, Sept. 1996.
21. B. Noble and M. Satyanarayanan. Experience with Adaptive Mobile Applications in Odyssey. *Mobile Networks and Applications*, 4(4):245–254, 1999.
22. ObjectWeb Open Source Software Community. ObjectWeb home page. http://www.objectweb.org, 2004.
23. OMG. CORBA Components. OMG Document formal/02-06-65, Version 3.0, Object Management Group, June 2002.
24. K. Petersen, D. Terry, M. Theimer, A. Demers, and M. Spreitzer. Flexible Update Propagation for Weakly Consistent Replication. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint Malo, France, Oct. 1997.
25. M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proc. 15th Symposium on Principles of Distributed Computing*, pages 1–7, 1996.
26. M. Satyanarayanan. Mobile Information Access. *IEEE Personal Communications*, 3(1), Feb. 1996.
27. M. Satyanarayanan. The Evolution of Coda. *ACM Transactions on Computer Systems*, 20(2):85–124, May 2002.
28. A. Silberschatz and P. Galvin. *Operating system concepts*. Addison-Wesley, 1994.
29. C. Szyperski, D. Gruntz, and S. Murer. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 2002.
30. D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou: A Weakly Connected Replicated Storage System. *Proc. 15th ACM Symposium on Operating Systems Principles*, 1995.
31. S. Williams, M. Abrams, C. Standridge, A. G., and E. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. ACM SIGCOMM*, Stanford University, CA, USA, 1996.