Object-Swapping for Resource-Constrained Devices

Luís Veiga and Paulo Ferreira
INESC-ID/IST
Distributed Systems Group
Rua Alves Redol N. 9,1000-029 Lisboa, Portugal
{luis.veiga,paulo.ferreira}@inesc-id.pt

Abstract

Mobile devices are still memory-constrained when compared to desktop and laptop computers. Thus, in some circumstances, even while occupied by useful objects, some memory must be freed. This must be performed while preserving referential integrity in object-oriented applications and without permanently losing data.

We propose a novel approach to object swapping in constrained devices, that favors portability and avoids most of the requirements of the previous approaches.

*Objects are incrementally replicated to devices in groups (*clusters*) of adaptable size. When there is shortage of memory, the middleware* detaches *the objects belonging to selected clusters from the application graph. It maintains graph correctness (via proxy replacement objects), and stores the swapped objects (in XML-format) in any nearby device with wireless connectivity and available storage.*

Nearby devices do not require any specific VM or middleware. They simply must be able to store and provide XML text. This approach is suited to an envisioned future in which there will be a myriad of small memory-enabled devices with wireless connectivity, scattered all-over, available to any user either to store data or to relay communications.

1. Introduction

Replication is a widely used technique to enable mobile computing by improving data availability and application performance as it allows to collocate data and code. Data availability is ensured because, even when the network is not available, data remains locally available. Furthermore, application performance is potentially better (when compared to a remote invocation approach) as all data accesses are local. Together with the file system paradigm (the most fundamental and ubiquitously adopted), object-oriented programming is the most widely used programming model as it is highly flexible (e.g., Java, C# and VB.Net, Python, etc.). It is natural then, that application development for mobile devices should be based upon replicated objects (thus keeping application data and logic locally, as most as possible) and adhering to the same development environment programmers are accustomed to.

However, mobile devices are so memory-constrained that, in some circumstances, even the memory occupied by useful reachable objects must be freed. This may occur because, at a particular instant, there are other more relevant replicas for which there is no memory available. Freeing the memory occupied by useful objects is delicate. Given that such objects can be accessed by applications through navigation of the object graph, the middleware must still ensure the referential integrity, while freeing such memory.

Shortcoming of Current Solutions: There is previous work in the literature regarding the decrease of memory occupation by applications, particularly w.r.t. mobile constrained devices. They address memory limitations either by transferring objects residing in mobile devices to other near-by computers [6, 1] or to persistent storage [7] and re-fetching them later, or they attempt at reducing memory usage by compressing object data [2, 3].

All existing solutions impose changes to the existing underlying VMs on mobile devices and computers receiving transferred objects, such as modified object tables, use of object surrogates, and dedicated distributed garbage collection (DGC) algorithms. This limits the range of devices where these solutions may be applied, as opposed to one based exclusively on user-level code. Furthermore, some solutions impose important additional CPU load and energy cost, paramount in mobile devices, since compression is a computational-intensive process.

Contribution and Paper Organization: We propose an alternative approach to freeing memory on mobile devices that favors portability and, as it avoids most of the requirements of the previous approaches, is more suited to be deployed on a myriad of existing and future devices. It resorts exclusively to user-level code and therefore does not require modification of the underlying virtual machine, making it rather portable. It further obviates the need to manage inter-process references among individual resident and swapped-out objects.

It was developed in the context of OBIWAN [4, 8], a middleware platform for object replication, with a compiler to automatically generate proxies and augment classes, that runs on mobile devices, on top of Java and .Net.

In OBIWAN, objects are incrementally replicated to devices in groups (*clusters*) of adaptable size. Objects not yet replicated are replaced, on the device, by proxies transparent to application code. When these proxies are invoked, object replication is triggered and, after replicating another cluster of objects, the proxies are removed from the object graph (i.e., *replaced* by the actual object replicas). Thus, there are no further indirections w.r.t. object invocation (i.e., the application runs at full-speed), once objects are replicated.

Our approach consists in considering a number (also adaptable) of chained (via references) object clusters as a single macro-object. We call such macro-object a *swap-cluster*. For every reference linking two different swap-clusters, proxy *replacement* (taking place when objects are replicated) is performed differently. For objects belonging to different swap-clusters, a special proxy always remains in the way. There is a performance penalty associated with these extra invocations, which can be rendered negligible, even when compared to existing solutions.

The remainder of this paper is organized as follows. The next section provides a system overview of the OBIWAN middleware, upon which *Object-Swapping* is incorporated. Section 3 describes in detail the architecture of the *Object-Swapping*, namely integration with object replication and memory management mechanisms, alongside with a prototypical scenario. In Section 4, we describe the main implementation aspects, and relevant issues regarding integration with programming languages such as Java and C#. *Object-Swapping* is evaluated in Section 5, and the relevant related work is presented and discussed in Section 6. The paper closes with some conclusions in Section 7.

2. OBIWAN Middleware

OBIWAN [4, 8] is a middleware platform aimed at providing flexibility for application development and runtime adaptability, allowing applications to cope with the multiple requirements and usage diversity found in mobile settings. Typically, applications running on OBIWAN, invoke objects locally replicated into the computer where they are being executed.

OBIWAN consists in a set of middleware component modules, portrayed in Figure 1, which provide runtime services to applications and to other higher-level services. Runtime services execute on top of a virtual machine (Java or .Net) in every node. These are services that are available to the application regular code if explicitly wanted, but were designed to be used by code automatically generated that extends application code. The most relevant modules w.r.t. *Object-Swapping* are described next.

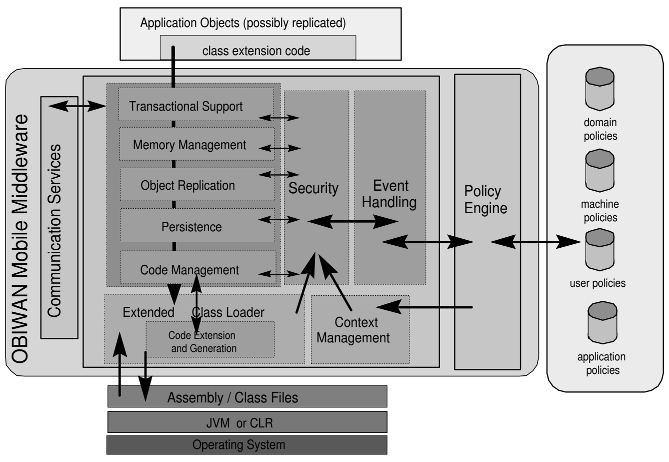


Figure 1. OBIWAN Middleware Components.

The *Object Replication* [9, 13] module provides the mechanisms for handling object-faults transparently, and supporting incremental object replication.The architecture is based on a set of OBIWAN core interfaces which deal with : i) creation and update of object replicas, ii) object-fault handling by proxy objects, and iii) proxy replacement, once the corresponding object is replicated.

These core interfaces are implemented by middleware code in application objects and proxies, which is automatically generated. The programmer is oblivious of them. Proxy objects also implement (via generated code) the same interfaces as application objects, since invocation of a proxy triggers object replication.

The *Memory Management* [11, 12] module is responsible for the distributed garbage collection (DGC), integration with the local garbage collector (LGC) provided by the virtual machine, and *Object-Swapping*. Memory management depends on object replication to be aware of which objects have been replicated in the process, and/or swapped-out.

The *Context Management* module abstracts resources and manages the corresponding properties whose values vary during applications execution. In particular, it is responsible for monitoring available memory and network connectivity.

The *Policy Engine* [10] is the inference component that manages, loads, and deploys declarative policies to oversee and mediate responses to events occurred in the system. Policies are stored and categorized by nature. A policy engine receives events generated by OBIWAN modules and applications, evaluates policy rules and triggers events, han-

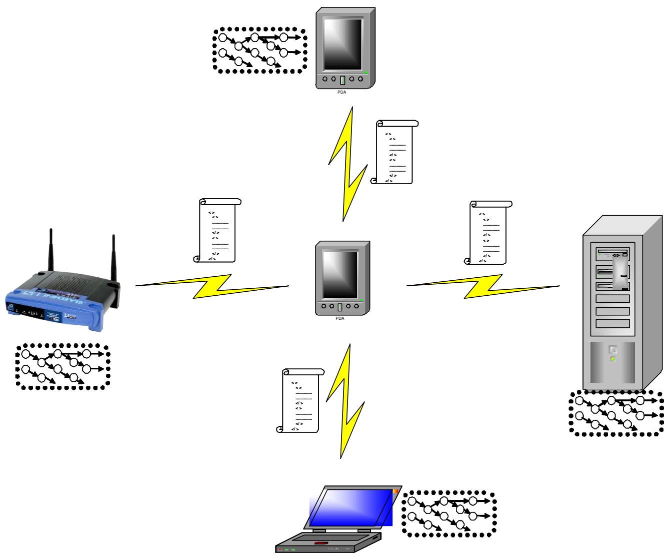


Figure 2. Object-Swapping to nearby devices.

dled by actions based on evaluation results.

The *Communication Services* abstract applications, and the rest of the OBIWAN middleware, from the limitations of existing virtual machines for mobile constrained devices (e.g., absence of remote method invocation and proper object serialization). They are circumvented by using a communication bridge [13] based on web-services, and automatic conversion of objects into wrappers, using XML.

3. Transparent Object-Swapping in OBIWAN

Mobile devices are so memory-constrained that, in some circumstances, even the memory occupied by useful reachable objects must be freed. This may occur because, at a particular instant, there are other more relevant replicas for which there is no memory available. This is more evident in resource constrained devices but also occurs in desktop systems [3] even with large memory heaps.

The memory management premise of preserving objects which are reachable from thread stacks and global variables (i.e., live objects) must be enforced with a somewhat relaxed approach: there are situations where live data must be ”demoted” to accommodate for other data being replicated that is considered, at that moment, more important. Data should not be plainly discarded, but the memory occupied by it, should nevertheless be cut down.

Our proposal consists in swapping-out the content of such objects to other devices with more resources available, in particular, free memory. Freeing the memory occupied by useful objects is delicate. Given that such objects can be accessed by applications through navigation of the object graph, the middleware must still ensure referential integrity, while freeing such memory.

Figure 2 depicts a prototypical scenario in which a PDA is running applications, on behalf of the user, on top of OBIWAN middleware. From time to time, the memory occupied by the object graphs of applications reaches a threshold value, possibly near the limit of the memory capacity of the device. At those moments, the OBIWAN middleware, evaluating the policies loaded, decides to swap-out a set of objects to nearby devices, if there are any. This action frees some memory while not discarding the swaped objects permanently. Later, each set of objects previously swapped-out may be fetched back from the device where it was transferred to.

The devices that receive swapped objects need not have neither OBIWAN nor even a virtual machine installed. They need only be able to store and return a textual representation of the serialized objects being swapped-out. If a device is able to store more than one set of swapped objects, each set must be given a unique ID (e.g., a number, a file name). Therefore, objects may be swapped-out to desktop and laptop PCs, other PDAs, or future wireless devices, with extended memory capacity, present in the room.

Management of *Swap-Clusters*: Taking into account the management of replicas described in Section 2, clusters of objects (or groups of clusters) are natural candidates to be swapped-out as they have been incrementally replicated, previously, into the mobile device as a whole. Hopefully, when one of the objects enclosed in the cluster becomes needed again, there is a high probability that the others will be as well. So, later, they will be swapped-in as a whole.

A *swap-cluster* is the basic unit of swapping. Each one contains all the objects comprised in a group of one or more object clusters, previously replicated. Swap-clusters are created by regarding a number (also adaptable) of chained (via references) object clusters as a single macro-object, i.e. a single swap-unit. As mentioned in Section 1, for every reference linking two different swap-clusters, proxy replacement (taking place when objects are replicated) is performed differently, as we describe next.

The proxy object is replaced with the corresponding object replica, but another type of proxy is also created. We call it a *swap-cluster-proxy* to distinguish it from proxy objects that once replaced, are discarded. The *swap-cluster-proxy*, in turn, holds a reference to the newly replicated object. This way, the reference returned by the middleware to application code does not target an object replica, but a swap-cluster-proxy instead. Thus, for objects belonging to different swap-clusters, a proxy always remains in the way.

Figure 3 depicts a situation in which the object graph of a process ($P1$) is divided in four swap-clusters: $swap-cluster-1$ to $swap-cluster-4$. Global variables (i.e., static fields), and variables defined in static methods, are

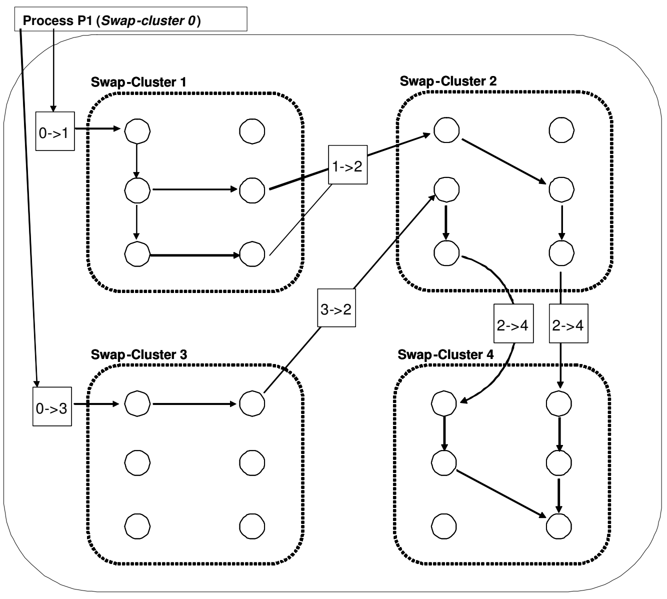


Figure 3. Object graph of a process comprising four swap-clusters.

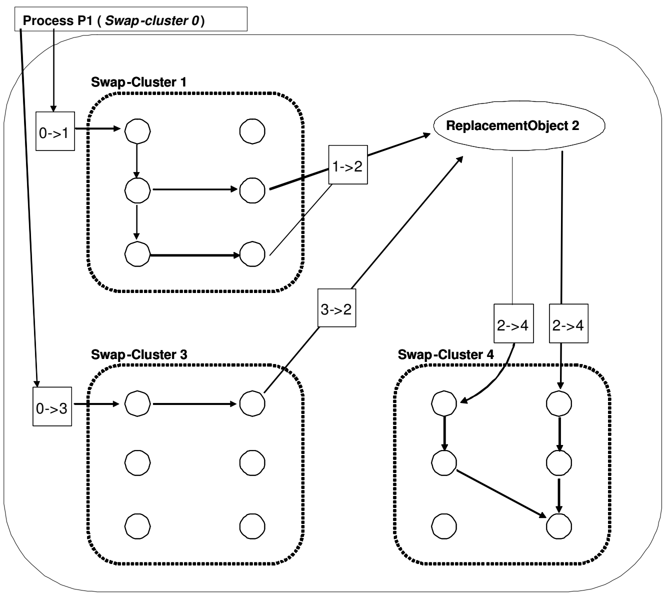


Figure 4. Object graph of a process after swapping-out $swap - cluster$ **2.**

regarded as belonging to a special swap-cluster, $swap - cluster - 0$. $Swap - cluster - 1$ and $swap - cluster - 3$ are reachable directly from the variables of the application. Both of them contain objects that reference other objects contained in $swap - cluster - 2$. When there are multiple references to the same object, across the same pair of swap-clusters, only a swap-cluster-proxy is required. This is the case in $swap - cluster - 1$. Objects in $swap - cluster - 4$ are only referenced from objects in $swap - cluster - 2$. Each individual swap-cluster may contain any number of objects.

Middleware code in swap-cluster-proxies also monitors reference-passing across swap-cluster boundaries (analogously to monitoring of inter-process references), and creates/reuses the appropriate swap-cluster-proxies. The middleware keeps track, for each swap-cluster, of swap-cluster-proxies regarding it, and their usage. This provides information about inbound/outbound references from/to other swap-clusters, and basic data w.r.t. recency and frequency, as these boundaries are transversed by the application.

Swap-Cluster Swapping-Out: When required (e.g., due to shortage of memory), the middleware may *detach* the objects belonging to a specific swap-cluster from the application graph, while maintaining correctness. This process is described by the following example depicted in Figure 4. It portrays the resulting situation after detachment of a swap-cluster, in this case, $swap - cluster$ 2.

A replacement-object for a swap-cluster (i.e., $ReplacementObject - 2$ which is simply an array of references) is created and filled with references to every swap-cluster-proxy referenced by $swap - cluster - 2$. Then, every swap-cluster referencing objects contained in $swap - cluster - 2$ will be made to reference $ReplacementObject - 2$ instead, by patching the internal references of every swap-cluster-proxy targeting objects in $swap - cluster - 2$.

Once a swap-cluster (e.g., $swap - cluster - 2$) is detached from the object graph (though still referenced by middleware code), the enclosed objects are serialized to XML and sent to a nearby device (e.g., via Bluetooth), along with a swap-cluster ID. The receiving device needs no other infrastructure (e.g., specific VM, adhering to a specific middleware, holding application class files, etc., as opposed to other existing solutions) other than being able to receive XML data and store it.

After a swap-cluster is swapped-out, the objects enclosed in it are completely detached from the application graph, and eligible for collection by the LGC running on the device. Thus, memory is released without destroying application graph integrity. Any swap-clusters may be swapped-out using this mechanism.

Swap-Cluster Reload: When a replacement-object is invoked, this means that the application is trying to access an object that belongs to a swap-cluster, which was previously swapped-out. Since one of the objects enclosed in the swap-cluster becomes needed again, there is a high probability that the others will be as well. So, they are swapped-in back as a whole by demanding the same XML-data, containing wrapped objects, that was sent earlier during swap-out.

All swap-cluster-proxies targeting objects enclosed in the swap-cluster being swapped-back must be updated. To this purpose, their internal references are patched in order to target the corresponding object replicas being swapped-in. Then, the replacement-object, as it is no longer needed, becomes eligible for local reclamation. Therefore, after $swap-cluster-2$ is brought back to $P1$, the resulting object graph would be similar to the one initially shown in Figure-3.

Integration with GC Mechanisms: The middleware prevents dead objects, belonging to swap-clusters that later became unreachable (unusable to the application), from consuming resources while being stored on the swapping devices. Therefore, the middleware cooperates with the LGC running on the virtual machine, w.r.t. to managing those objects. However, the reachability of a swap-cluster must be considered as a whole.

Thus, when a replacement-object, standing in for a swap-cluster that has been swapped-out, becomes unreachable, this means that all object replicas enclosed in it are already unreachable to the application. Therefore, the swapping device may be instructed to discard the XML text with the contents of the swap-cluster.

Conversely, while the replacement-object (and its associated swap-cluster) is reachable, the LGC must behave conservatively. Thus, it must regard as reachable all objects belonging to the swap-cluster, even if all but one of them are garbage. Therefore, the whole swap-cluster must be preserved on the swapping device.

When ultimately deemed as unreachable to the application, a swap-cluster may be dropped from the swapping node, or set-aside if their content is still required for other purposes (consistency, reconciliation, versioning, etc.). The replacement-object is simply reclaimed by the LGC.

The integration with LGC mechanisms just described does not constitute a DGC infrastructure covering swapping devices. There are no explicit references among the objects residing in devices running applications, and those serialized in swapping devices. All the decisions are made locally to the device running the application. The swapping device is instructed just to store, return, or drop XML-data.

4. Implementation

Object-Swapping in the OBIWAN middleware was implemented in the context of its *Memory Management* module. It runs on top of the .Net Compact Framework installed on a IPAQ 3360 Pocket PC with Bluetooth connectivity at 700Kbps. The primary programming language used is C#. Policies that deploy the various modules are coded in XML. Transfer of swapped-out objects is achieved resorting to the *Communication Services* module which leverages the ability of .Net CF to invoke web-services.

Code for swap-cluster-proxies is automatically generated by the OBIWAN compiler (`obicomp`). It generates a specific class of swap-cluster-proxy, for each type class (e.g., class A) defined by the application (analogous to generation of proxy objects). Thus, the class for each type of swap-cluster-proxy implements two interfaces: i) the `ISwapClusterProxy` interface for common methods such as `patch` and `detach`, and ii) the interface containing the public methods of the type class (e.g., interface IA containing public methods of class A).

The code generated for swap-cluster-proxies implements all methods of the application interface (e.g., IA), with a similar code excerpt that verifies references being passed as parameters and return values, while also relying on invoking the actual object replica it refers to. With every referenced intercepted, this code verifies whether it is necessary to: i) create another swap-cluster-proxy to wrap a reference from/to another cluster, ii) patch an existing swap-cluster-proxy that is being handed to/from another swap-cluster, iii) dismantle a swap-cluster-proxy received but that refers to an object within the same swap-cluster.

Implementation of methods belonging to interface `ISwapClusterProxy` (e.g.,`patch`, `detach`) delegate to static methods of a class (`SwapClusterUtils`) that contains behavior common to all swap-cluster-proxy types.

Enforcing Object Identity: Within each swap-cluster, object identity is ensured because references to object replicas are never compared against references to swap-cluster-proxies, referring to objects in the same swap-cluster, due to rule iii) presented earlier.

Enforcing object identity when comparing references to objects in other swap-clusters (i.e., actually comparing references to swap-cluster-proxies) cannot rely solely on reference comparison (operator ==). A simple example would be that of an object in $swap-cluster-X$, if referenced from two different swap-clusters, will be necessarily represented by two different swap-cluster-proxies (because they regard different source swap-clusters).

This is solved by overloading the reference comparison operator `==` in C#, for each class of swap-cluster-proxy, with a method that verifies whether the two arguments received are swap-cluster-proxies (i.e., implement interface `ISwapClusterProxy`), and actually refer to the same object. In other languages, such as Java, that do not allow this overloading, comparisons must rely solely on method `Object.Equals`, that can be overloaded.

Optimizing Code for Iterations: The use of global variables, while iterating object graphs (e.g., lists) that may span several swap-clusters, causes the creation of a new swap-cluster-proxy for each object reference returned, and consequent discard of the swap-cluster-proxy that the variable previously pointed to. In this cases, with a little help from the programmer, this behavior can be optimized re-using always the same instance of swap-cluster-proxy (as it was indeed the actual variable).

Class `SwapClusterUtils` provides a static method (`assign`) that may be invoked with swap-cluster-proxies with source in $swap - cluster - 0$. This method updates an internal field in the swap-cluster-proxy that marks it, and changes its behavior. The next time the swap-cluster-proxy intercepts a reference to be returned, instead of creating a new swap-cluster-proxy to be returned to application code (discarding itself), it patches itself. This way, it now refers to the object being returned by the application method that was invoked. Therefore, in practice, the swap-cluster-proxy will return to application code a reference to itself (though already modified internally), that minimizes creation of swap-cluster-proxies and optimizes iterations.

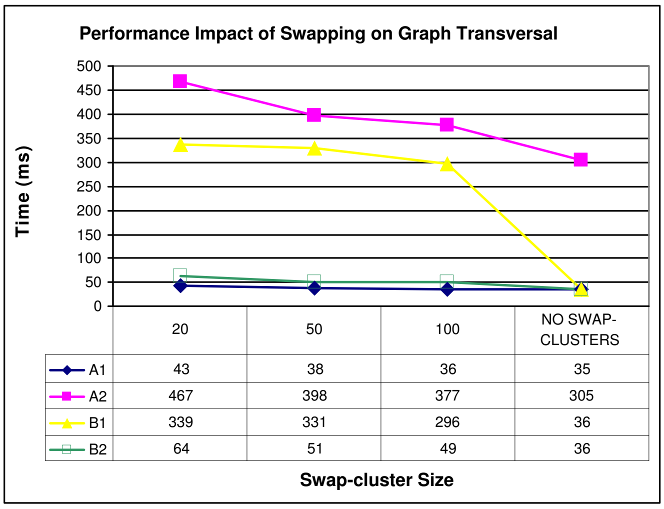
Swapping Manager: The `SwappingManager` class, by policy definition, is registered as a listener of all events regarding replication of clusters of objects, by using specific methods as actions. It also triggers specific events regarding object-swapping but that are presently not listened by any other module.

It manages swapping by maintaining information regarding all swap-clusters (loaded or swapped), and all objects belonging to each one, stored in hash-tables. It also contains entries for all swap-cluster-proxies w.r.t. references to/from each swap-cluster (using weak-references). When a swap-cluster-proxy becomes unreachable, its `finalizer` invokes code that eliminates entries referring to it.

5. Evaluation

Qualitative (Portability): *Object-Swapping* in OBIWAN is a novel approach to reduce memory usage by replicated objects in mobile devices that is portable, and imposes fewer demands on the surrounding infrastructure, w.r.t. other approaches found in previous related work. It is compliant with LGC and DGC managing replicated objects. It does not require modification of the underlying virtual machine on the mobile device. It further obviates the need to manage inter-process references among individual resident and swapped-out objects.

With object-swapping, devices receiving swapped objects do not need to have VM or middleware installed. The swapping device is instructed simply to store, return or drop XML-data. This favors portability since it does not require swapping devices to install a specific VM, or even a middleware platform like OBIWAN, but simply store portions of text (XML-encoded) data with a corresponding key (that would be the cluster name).



	20	50	100	NO SWAP-CLUSTERS
◆ A1	43	38	36	35
■ A2	467	398	377	305
▲ B1	339	331	296	36
□ B2	64	51	49	36

Figure 5. Performance penalty of Object-Swapping w.r.t. swap-cluster size and graph transversals.

Quantitative (Performance): Our approach incurs a minor performance penalty since there are extra invocations, due to the indirection maintained in swap-cluster-proxies. Nonetheless, this only happens when an swap-cluster boundary is crossed. In favorable scenarios, they are only required with frequency inverse to the average number of objects per swap-cluster (e.g., 1/20, 1/50, 1/100), which could render them negligible.

In Figure 5, we present a study of the performance impact (i.e., slowdown) imposed by the *Object-Swapping* mechanism (namely due to the management of swap-clusters and the indirection of method calls in swap-cluster-proxies) during normal application execution. We employed a micro-benchmark that measures the performance of three illustrative graph transversal strategies. These strategies are based on recursive and iterative invocations, on a list of 10000 64-byte objects, of simple (quasi-empty) methods, in order not to mask the overhead being measured.

The benchmark consists of 4 tests (*A1*, *A2*, *B1*, and *B2*). Each test was executed against 4 different swapping configurations, i.e., with swap-clusters containing 20, 50, and 100 objects each and, finally, without swapping (*No Swap-Clusters*) in order to provide a lower-bound case w.r.t. timing. Naturally, the overhead will decrease as the size of swap-clusters increases, and be null if *Object-Swapping* is not active.

Test *A1* consists of recursively executing a simple method on each object along the list containing the 10000 objects. The method receives, as argument, an integer that is initially set to 0 and incremented in each recursion step, thus measuring the *recursion depth*. In this test, each swap-cluster-proxy is invoked only once when the corresponding swap-cluster boundary is crossed during the recursion. The results range from 37 to 43 ms, with a maximum overhead of 16%. Therefore, it is not significative and is negligible for swap-clusters containing 100 or more objects.

Test *A2* is an extension of Test *A1* with higher computational complexity and is *explicitly* aimed at producing greater overheads due to larger number of swap-cluster-proxies invocations. It makes use of the same recursive execution (named *outer recursion*) along the list of 10000 objects. In this test, however, each step of the recursion, instead of simply passing an integer argument as in *Test A1*, is extended to trigger an *inner recursion* that stops when it reaches depth 10 (or the end of the list) and returns a reference to the object reached in the list (i.e., at most 10 positions ahead within the list) without modifying the object graph.

This test takes substantially more time since there are roughly 10 times more object invocations. Furthermore, whenever an *inner recursion* crosses a swap-cluster boundary, an additional swap-cluster-proxy is created to mediate the object reference being returned. The swap-cluster-proxy is later reclaimed by the LGC when the *outer recursion* advances to the next step.

The results range from 305 to 467 ms, indicating a maximum overhead of 53%. This larger overhead is due to the extra swap-cluster-proxies being created. In the case of swap-clusters containing 20 objects, this happens for roughly half of the object references returned by the *inner recursions* (recall these have a maximum depth of 10).

Test *B1* consists of performing a full *iteration* over the 10000 objects of the list, using a *for* loop and a global variable. When invoked, each object returns a reference to the next element in the list. Recall that global variables are considered as belonging to a special swap-cluster (*swap-cluster-0*), as described in Section 3. Therefore, a swap-cluster-proxy is created for every reference transversed during the loop.

The results range from 36 to 339 ms. As expected, this kind of iterations based on successive assignments of global variables produces a significant overhead due to the constant creation and deletion of swap-cluster-proxies (one for each new value the variable assumes). Recall that this situation has been predicted and described in Section 4.

Therefore, **Test *B2*** consists of performing the same full *iteration* over the complete list, this time using the optimizations for iterations described in Section 4. The results of this test are much more encouraging and range from 36 to to 64 ms. The speed-up provided by the optimizations described is more than five-fold in all cases.

In **overall analysis**, note these tests depict worst-case scenarios since the remaining overheads could easily be masked if there was any object functionality actually being invoked in each iteration or recursion step (e.g., accessing some object properties or invoking on or more methods). Therefore, these results show that it is feasible to address *Object-Swapping* for resource-constrained devices, by employing a portable approach, resorting exclusively to user-level code, without modifying existing virtual machines used in mobile computing (e.g., .Net CF).

In addition, our proposed solution also has several benefits over a *naive* one that would have one proxy per each object and all references mediated by them. Common application objects are small. So, this could potentially double memory occupation when fully-loaded or roughly the same as full. This approach would also inevitably impose a higher performance penalty, due to indirections. Furthermore, even when all objects were swapped, the proxies would still remain, which would incur in higher memory overhead.

6. Related Work

Freeing memory on mobile devices has been addressed previously. In [6, 1, 5], some objects are migrated to a nearby server machine from where will be re-fetched later, if needed. To provide transparency to applications, such objects are replaced by a surrogate. It imposes changes to the underlying VM. These include i) object tables must account for objects residing in other machines; ii) modifications to LCG behavior instrumenting the LGC to monitor on an object-by-object basis, which objects to swap-out; and iii) there must be a distributed garbage collection (DGC) algorithm managing references among resident and migrated objects. Even though these approaches are distributed, they do not address replication and related issues (replica management, consistency, etc.).

In [2], a mechanism is proposed to perform compression on the Java virtual machine heap, where large objects (greater than 1.5 Kb) are compressed and decompressed. In addition, large array objects are broken down into smaller sub-objects, each being “lazily allocated” upon its first write access. Constant on-the-fly data compression performed on the heap saves memory but imposes additional CPU load and energy cost, since compression is a computational-intensive process. This solution also imposes the use of a modified VM.

The work in [3] describes an analytical model for main memory compression, based exclusively on software. Applying compression to pages in main memory was first suggested in [14]. The system reserves a number of pages in main memory that act as an additional intermediate level in memory hierarchy (compressed memory pool). Pages about to be swapped to disk are compressed and swapped to the compressed pool instead. Both page compression and disk writing can be performed asynchronously. Nevertheless, even if compression takes more time than disk writing, the gains obtained during page reload to main memory (decompression is much faster than reading for disk) are much greater.

The only disadvantage is that the compressed-memory pool actually reduces the memory available to applications. Therefore, the system must balance, according to application behavior and needs, the size of the compressed memory pool. Thus, devoting too much memory to the compressed-memory pool hurts performance as much as not reserving enough [14]. In multi-core systems, one of the processors may be dedicated solely to compression/decompression activities. This solution is not suited for resource constrained devices as it is directed mainly at workstations. Furthermore it requires modifications to the OS kernel which hinders portability.

The .Net Micro Framework [7] is a very small .Net virtual machine for embedded systems with very limited resources (e.g., wrist-watches). It employs several techniques to reduce memory foot-print w.r.t. both application code and data. It maintains a global string table that is shared to store names of types, methods and fields, to reduce RAM (w.r.t. application code) and ROM (w.r.t. framework code) usage.

Another innovation provided by .Net Micro is the use of extended weak references. References of this type have precedence over regular weak references. A specialized garbage collector attempts to copy to available persistent memory (e.g., flash-memory in CompactFlash/Secure Digital cards) unreachable objects that are targeted by extended weak references, instead of reclaiming them, which is the case with objects targeted exclusively by regular weak references. This is performed locally and is therefore limited by the total memory (main, and additional memory cards) of the device.

7. Conclusion

We present an approach to reduce memory usage by replicated objects in mobile devices that is portable, and imposes fewer demands on the surrounding infrastructure.

As a negligible trade-off, our approach requires that every reference between objects in different swap-clusters be mediated by a proxy. It does not require modification of the underlying virtual machine on the mobile device. It further obviates the need to manage inter-process references among individual resident and swapped-out objects. Nodes receiving swapped objects do not need to have VM or middleware installed. The swapping node is instructed just to store, return or drop XML-data.

In the future, we expect a scenario where (much as wireless access points are becoming omni-present) there will also be an increase in small memory-enabled devices with wireless connectivity, scattered all-over, that are available to any user (either to store data or to relay communications). Our approach is the most suited to this scenario.

References

[1] D. Chen, A. Messer, D. Milojicic, and S. Dwarkadas. Garbage collector assisted memory offloading for memory-constrained devices. In *Fifth IEEE Workshop on Mobile Computing Systems and Applications*. IEEE Press, 2003.

[2] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. Heap compression for memory-constrained java environments. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 282–301. ACM Press, 2003.

[3] I. Chihaia and T. Gross. An analytical model for software-only main memory compression. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 107–113, New York, NY, USA, 2004. ACM Press.

[4] P. Ferreira, L. Veiga, and C. Ribeiro. Obiwan - design and implementation of a middleware platform. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1086–1099, November 2003.

[5] X. Gu, A. Messer, I. Greenberg, D. Milojicic, and K. Nahrstedt. Adaptive offloading for pervasive computing. *IEEE Pervasive Computing*, 3(3):66–73, 2004.

[6] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu. Towards a distributed platform for resource-constrained devices. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 43. IEEE Computer Society, 2002.

[7] D. Thompson and C. Miller. Microsoft's.net microframework, product positioning and technology whitepaper. http://www.aboutnetmf.com/entry.asp, sep 2006.

[8] L. Veiga. Obiwan: A middleware for memory management of replicated objects in distributed and mobile computing. Technical report rt/27/2006, INESC-ID Lisboa, oct 2006.

[9] L. Veiga and P. Ferreira. Incremental replication for mobility support in OBIWAN. In *The 22nd International Conference on Distributed Computing Systems*, pages 249–256, Viena (Austria), July 2002.

[10] L. Veiga and P. Ferreira. Poliper : Policies for mobile and pervasive environments. In *3rd Workshop on Reflective and Adaptive Middleware. In 6th ACM International Middleware Conference*, Toronto, Canada, October 2004.

[11] L. Veiga and P. Ferreira. Asynchronous complete distributed garbage collection. In *19th IEEE International Parallel and Distributed Processing Symposium*, Denver, CO, USA, april 2005.

[12] L. Veiga and P. Ferreira. A comprehensive approach for memory management of replicated objects. Technical report rt/07/2005, INESC-ID Lisboa, april 2005.

[13] L. Veiga, N. Santos, R. Lebre, and P. Ferreira. Loosely-coupled, mobile replication of objects with transactions. In *Workshop on Qos and Dynamic Systems. 10th IEEE International Conference On Parallel and Distributed Systems(ICPADS 2004)*, 2004.

[14] P. Wilson. Operating system support for small objects. *Object Orientation in Operating Systems, 1991. Proceedings., 1991 International Workshop on*, pages 80–86, 1991.