An effective offloading middleware for pervasive services on mobile devices

Shumao Ou\textsuperscript{a,\ast}, Kun Yang\textsuperscript{a}, Jie Zhang\textsuperscript{b}

\textsuperscript{a} Department of Electronic Systems Engineering, University of Essex, UK
\textsuperscript{b} Department of Computing and Information Systems, University of Bedfordshire, UK

Received 2 October 2006; received in revised form 6 April 2007; accepted 11 April 2007
Available online 22 April 2007

Abstract

The practical success of pervasive services running in mobile wireless networks and devices relies on their ability to provide effective and efficient offloading support, so as to satisfy the increasing demand for mobile devices to run heavier applications (e.g., those running on desktop PCs). Offloading is an effective mechanism for leveraging the severity of resource-constrained mobile devices by migrating some computing load to nearby resource-rich surrogates (e.g., desktop PCs, servers) on home networks or their extension. This paper proposes a light-weight and efficient offloading middleware, which provides runtime offloading services for resource-constrained mobile devices. The middleware considers multiple types of resources (i.e., memory, CPU and bandwidth) and carries out application partitioning and partition offloading in an adaptive and efficient manner. The corresponding algorithms are presented. The evaluation outcomes indicate the effectiveness and efficiency of this service offloading solution.

\textcopyright{} 2007 Elsevier B.V. All rights reserved.

Keywords: Middleware; Pervasive services; Offloading; Mobile devices; Resource constraint; Partitioning

1. Introduction

Recent years have seen a significant increase in research and development of mobile and wireless networks, including UMTS (Universal Mobile Telecommunications System),
IEEE 802.11/16/20, mobile ad hoc networks, and sensor networks. The future success of these network systems lies in their ability to provide users with cost-effective services that have the potential to run anywhere, anytime and on any device without (or with little) user attention. Services with these features are termed pervasive services, which is an active branch of pervasive (or ubiquitous) computing [1].

With advances in mobile terminals and wireless communications, the demand for mobile devices to run heavier applications (such as video playing or editing) is increasing. Apparently, even the most powerful PDAs today (due to the size and weight constraints) are unable to compete against their desktop siblings with regard to any type of resource, especially battery life and network capacity. Meanwhile, in some working, entertaining and living environments, computing resources are often rich. For instance, in offices or cafes, some desktop PCs may be idle while mobile devices are busy. As such, it makes sense for these resource constrained mobile devices to make use of the resources available in their vicinity to leverage their resource insufficiency. For example, it is natural for John to expect to watch the video clips of his daughter’s Christmas concert (even if with degraded quality), which was just uploaded to their home server, during a tedious wait in a Heathrow boarding lounge. Therefore, there is a clear need for mobile devices to run heavier applications. We refer to this kind of computation or communication migration as offloading for pervasive services. It is also called surrogate computing or cyber foraging [10,1].

Offloading is not a new concept. It has been around in the form of proxies and surrogates for many years. For example, it has been used for load balancing in distributed systems [1]. However utilizing offloading mechanisms in the domain of resource constrained mobile devices has gained popularity lately.

Since pervasive devices are heterogeneous in terms of CPU power, memory, communication capabilities, battery life and software features, middleware is the most common solution for facilitating interoperability in pervasive environments [2]. The aim of our work is to provide a light-weight and effective middleware for pervasive service offloading in mobile devices.

Some research work on offloading mechanisms for resource constrained mobile devices has been proposed [3–11]. However, they are limited in one way or another. Since pervasive services tend to be more complex and dynamic [3], ideally, an offloading system should consider multiple types of resource constraint. In this paper, we present our work on a runtime offloading middleware specifically for pervasive services. Instead of considering only one type of individual resource, such as memory [3] or CPU [4], our offloading approach counts on a combination of these plus communication cost (i.e. bandwidth resources). Through grasp of service runtime feature in a cost-effective manner, the efficiency of our offloading approach is significantly increased. The adaptability and efficiency of our offloading middleware also lies in its light-weighted service partitioning algorithm and feasible offloading mechanisms.

As far as middleware for pervasive computing is concerned, the Spectra project [5,9] proposes a remote execution system for mobile devices used in pervasive computing. Chen et al. proposed an offloading framework [4] that focuses on a Java-based environment and dynamically decides whether to execute locally or remotely, based on the cost of Java-code compilation, computational complexity and communication channel conditions. The Coign [6] project proposed a system to use a min-cut algorithm to statically partition binary
applications built from Microsoft’s Component Object Model (COM) components. Li et al. [7] constructed a static cost graph and applied a partition scheme to statically divide an application’s tasks into client and server subtasks during the design time. Goyal et al. [11] proposed a cyber foraging solution using virtual machine technologies. However, these works did not provide specific offloading mechanism.

Gu et al. [8] proposed an adaptive infrastructure for Java application offloading execution, which adapted a min-cut [12] heuristic algorithm to dynamically partition an application. In addition, the Gu et al. work considers only memory and supports only one surrogate, however, most mobile applications are more sensitive to bandwidth and CPU cycles, as is the case in the John’s video playback scenario above. As such, an ideal partitioning solution should consider memory, CPU and bandwidth simultaneously. Focusing on only one of them will render it unfeasible in practice. Our offloading middleware considers both the interaction properties and the resource consumption when conducting partitioning and offloading. And we aim to relieve not only memory constraints but also CPU usage and bandwidth constraints.

The remainder of this paper is organized as follows. Section 2 describes a scenario for offloading services and presents some issues involved in designing offloading systems. The architecture of our proposed offloading middleware is discussed in Section 3. In Section 4, we present a class instrumenting approach for offloading purposes. A Service partitioning algorithm is presented in Section 5. In Section 6, we discuss some practical issues for implementing the offloading middleware. Implementation and evaluation comprise Section 7 and the paper concludes in Section 8.

2. A virtual home pervasive service scenario and some design issues

2.1. Scenario: John at Airport

Fig. 1 illustrates the John Scenario mentioned in Section 1. In John’s home network system, a server is connected to the Internet through a home gateway. The home network server serves as a file server in this scenario storing compressed video clips. We assume that the airport provides APs (Access Points) that enable mobile devices to access to the Internet.

Suppose John receives an instant message from his daughter saying that four five-minute video clips of her performance have just been uploaded onto their home network server.
Since there is still half an hour before boarding, John decides to download some video clips from their home network server and to play them back on his PDA. However, although the connections to the airport’s APs are stable, the remaining time and battery power might not be sufficient to download, decode and play all of the video clips. Fortunately, John’s PDA, with our offloading middleware installed, detects a surrogate, which provides application offloading services. The offloading middleware in his PDA partitions the video-downloading-and-playing application, offloads downloading and decoding components (which are bandwidth and computation intensive, respectively) to the surrogate. By using the broadband network resource, the surrogate manages to download video clips in a very short time; and then it decodes the video clips and sends decoded video data to the PDA for playback. The surrogate’s rapid downloading and decoding means John gets a faster response and obtains higher video quality. At the same time, his PDA conserves its battery power. Relying on the offloading service, John manages to view all the video clips before boarding.

2.2. Design issues

The John at Airport Scenario imposes new requirements upon the existing software environment. To achieve the envisioned virtual home scenario, an adaptive, cost-effective, and high efficiency runtime offloading middleware is needed. To design such an offloading middleware system, the following design issues need to be considered:

- **Service Partitioning**: Pervasive services need to be partitioned for offloading. When selecting service components to partition, components need to be weighed and distinguished as offloadable or unoffloadable. Unoffloadable components include those that handle user interaction or access local I/O devices.

- **Resource-Awareness**: The resource usages of the running environment (including memory usage, CPU utilization in the mobile device and the available network bandwidth) need to be monitored in order to facilitate offloading decision making.

- **Light-weight and transparent implementation**: The resource constrained nature of mobile devices means that offloading systems themselves should be designed in a light-weight manner. The operation of offloading should also be carried out as transparent to users as possible.

- **Easy to use**: It should be for normal mobile device users who have no special knowledge of mobile applications.

- **Easy to integrate and interoperate**: It should be integrate readily with other home network software.

Other design issues, including security mechanism and fault tolerance, are also important but are beyond the scope of this paper.

3. The proposed pervasive service offloading middleware

3.1. Overall system architecture

Applications running on mobile devices can interact with this middleware directly to invoke its offloading function. Alternatively, mobile device users can manually interact
with the middleware via a graphical user interface (refer to the implementation section of this paper for details). Pervasive services written by Java are used in our current system. However, the service partitioning and offloading principles discussed in this paper are equally applicable to the pervasive services developed by other object-oriented programming languages.

Fig. 2 depicts the architecture of the offloading middleware. The offloading middleware consists of the following modules:

- **Instrumenting module**: Used to modify the Java classes (which are in bytecode format and may be packed inside a JAR file) on the fly, to make them suitable for offloading. This is discussed in detail in Section 4.
- **Partitioning module**: Used to partition applications into one local execution partition for running on the mobile device and one or more (e.g. $k > 1$) remote execution partitions for running on surrogates. A $(k + 1)$ partitioning algorithm will be presented in detail in Section 5.
- **Resource Monitoring module**: Monitors the resource usages of the running environment including the mobile device and the underlying network.
- **Offloading Decision Engine**: Makes partitioning and offloading decisions according to changes in the runtime environment, mainly, the changing in resource utilization. An offloading is carried out only when the total cost of offloading execution (including
component migration, remote component interaction and offloading system overhead) is lower than the cost of local execution.

- **Offloading module:** Used to execute the partition offloading. Remote execution partitions along with their runtime execution statuses are serialized and migrated to surrogate(s) for remote execution.
- **Secure Communication channel:** For secure communication between the mobile device and surrogate(s).

The typical operation procedure of the above middleware system can be described as follows:

1. Service components (e.g., Java classes) are instrumented to make them suitable for offloading;
2. Resources, such as memory usage, CPU utilization, and network bandwidth are monitored;
3. Significant changes in resource usage triggers the offloading decision engine to make decisions for service partitioning and offloading;
4. The classes are partitioned to one local execution partition and one or more remote execution partitions for surrogate(s);
5. Remote execution partitions, along with their runtime execution statuses, are serialized and migrated to surrogates;
6. A secure communication channel for secure partition migration and remote communication is established (optional).

### 3.2. Solution for the virtual home scenario

Using the offloading middleware, the procedure of fulfilling the *Virtual Home* scenario is illustrated in **Fig. 3**. The numbered arrows, which illustrate the interaction amongst
the home network server, the home gateway, the surrogate, and the mobile device, are explained as follows:

1. The mobile device logs onto the home gateway;
2. The home gateway responds to the mobile device with the access information for the home network server;
3. The mobile device accesses the home network server to search for the required video clips;
4. The home network server returns with detailed information about the video clips (such as URL, size, and so forth);
5. The mobile device discovers available surrogates and send an offloading computing request to the most suitable surrogate;
6. The surrogate accepts the offloading request and opens a secure communication channel;
7. The mobile device initializes service partitioning and offloads the downloading and decoding related classes to the surrogate;
8. The surrogate loads the offloaded classes and notifies the mobile device to begin to run the service;
9. The surrogate requests to download video clips from the home network server;
10. The home network server returns with the requested video clips;
11. The surrogate decodes the downloaded video clips and sends the decoded video data to the mobile device;
12. The mobile device plays back the decoded video clips;
13. After the playback is finished, the mobile device notifies the surrogate to terminate this offloading computing session;
14. The mobile device notifies the home gateway to terminate its access to the home network server.

4. Class instrumenting

Class instrumenting is a process that transforms the classes to be offloaded into a form that is suitable for remote execution (including maintaining the correct communication with other local classes) in surrogates. When conducting a class instrumenting, a dedicated proxy class is generated automatically as a shadow class of the instrumented class. The name of the proxy class is the instrumented class’s name while the name of the instrumented class is changed (i.e. with an “I_” prefix).

Fig. 4 illustrates the procedures for class invocation with the arrows representing the invocation directions. Fig. 4(a) shows a normal (i.e. not offloading) inter-class invocation: class App directly invokes the methods in the class Decode and gets the results directly from it. In Fig. 4(b), the original class Decode is instrumented with the name of I_Decode. A proxy class using the original class’s name (i.e. Decode) is generated. All invocations to Decode from class App are directed to class I_Decode via the proxy class Decode. The instrumented class can be offloaded to a surrogate, as shown in Fig. 4(c). In this case, the invocation will be performed through a remote communication channel. Fig. 5 illustrates how the instrumenting of the class Decode is carried out. The function OffloadingCall in
the proxy class takes the class’s name, the method’s name, and the parameters for invoking the instrumented class.

For illustration purposes we used user-readable Java source code in Fig. 5. However, the instrumenting module in our middleware system is able to work on a service’s binary code as the source codes of pervasive services running on mobile devices are usually unavailable to offloading systems. Several toolkits are available for Java bytecode modification, including Javassist [13] and BCA [14]. Our instrumenting module is developed on these solutions. It directly loads Java bytecode without knowledge of source codes and instruments the bytecode on the fly.

5. Service partitioning

Service partitioning partitions a service into two or more partitions for execution in the mobile device and surrogates. The main goal of the partitioning algorithm is to keep the component interaction between the partitions (i.e. the communication cost between the mobile device and surrogates) as small as possible. In this section, we begin with cost modeling of a pervasive service. We then formulate the partitioning problem and propose a \((k + 1)\) partitioning algorithm in two steps. Furthermore, we discuss the Heavy-Edge and Light-Vertex Matching (HELVM) algorithm in detail, which is the core sub-algorithm of the \((k + 1)\) partitioning algorithm.
5.1. Service component weighing: Multi-cost graph

An undirected graph, called a multi-cost graph \( G = (V, E) \), is used to represent a pervasive service. The vertex set \( V \) represents service components. As Java applications are selected as the partitioning target, each vertex in the multi-cost graph represents a Java class. Each vertex is annotated with 3 cost weights via a 3-tuple \( (W_{\text{Mem}}, W_{\text{CPU}}, W_{\text{BW}}) \), which represents the normalized memory, CPU time, and bandwidth requirements of this vertex. To reduce computational complexity, a composite vertex weight is used to represent the three weights:

\[
    w(v) = \varepsilon_1 W_{\text{Mem}} + \varepsilon_2 W_{\text{CPU}} + \varepsilon_3 W_{\text{BW}}
\]

where \( \varepsilon_1, \varepsilon_2, \) and \( \varepsilon_3 \) are the importance factors of each weight. The edge set \( E \) of the multi-cost graph represents the interactions (including method invocation and data access) amongst classes. For example, an edge \( e(v_i, v_j) \in E \) represents the interactions between vertices \( v_i \) and \( v_j \). The weight of an edge, \( w(e(v_i, v_j)) \) is the total number of interactions amongst \( v_i \) and \( v_j \). We do not distinguish between the direction of invocation and data access as we assume that the overhead for either direction is identical. The edge-weights represent only the numbers of interactions. The bandwidth required for interactions is considered in the vertex-weights.

5.2. \((k+1)\) partitioning algorithm

The problem of pervasive service partitioning is similar to that of partitioning a finite element graph into a certain number of disjoint subsets of vertices while fulfilling some given objectives (e.g. minimizing the amount of connection between the subsets). This type of graph partitioning problem is known to be NP-complete [15]. Our algorithm is an attempt to find an optimal solution. Given a service’s multi-cost graph \( G = (V, E) \) and a non-negative integer \( k \) \((k \leq |V|)\), the \((k + 1)\) partitioning algorithm is intended to find one unoffloadable partition \( V^U \) and \( k \) disjoint offloadable partitions \( V^O_1, V^O_2, \ldots, V^O_k \) satisfying:

(i) \( \bigcup_{m=1}^k V^O_m = V \setminus V^U \) and \( V^O_m \cap V^O_n = \emptyset \) for \( 1 \leq m, n \leq k \) and \( m \neq n \);

(ii) the edge-cut of \( V^O_m \) for \( V^O_m, V^O_n \in \{ V^U, V^O_1, V^O_2, \ldots, V^O_k \} \)

\[
    C^{(m,n)} = \sum_{e(u,v) \in E \cap m \in V^O_m \wedge v \in V^O_n} w(e(u,v))
\]

i.e. the sum of the edge-weights whose incident vertices belong to different partitions is minimized subject to the constraints defined by (iii);

(iii) \( \forall m : \psi^m \leq (T^m \pm \delta^m) \), where \( \psi^m \) is the sum vertex-weight in partition \( V^O_m \), i.e.,

\[
    \psi^m = \sum_{v \in V^O_m} w(v)
\]

\( T^m \) and \( \delta^m \) are the constraints that are predefined to represent the threshold and the fluctuation in partition \( V^O_m \). \((T^m \pm \delta^m)\) define the lower and upper bounds of the constraints.

The algorithm involves two main steps: unoffloadable vertex merging and coarse partitioning.

1. Unoffloadable vertex merging: All unoffloadable vertices need be merged into a multinode \( U \). (A multinode is composed of two or more vertices. The multinode will
be treated as a normal vertex afterwards). Let Ω represent the set of all unoffloadable vertices. The weight of vertex U is the sum of all the unoffloadable vertices’ weight, i.e. \( w(U) = \sum_{v \in \Omega} w(v) \). The edges connecting to U are the unions of the edges of which connecting the unoffloadable vertices being merged, i.e. \( e(U, v) = \bigcup_{u \in \Omega} e(u, v) \). The weight of a united edge is the sum of the weights of those edges being merged, i.e. \( w(e(U, v)) = \bigcup_{u \in \Omega} w(e(u, v)) \).

(2) \((k + 1)\) Coarse partitioning: Let the multi-cost graph with all the unoffloadable vertices merged be the graph \( G_0 \), the aim of this step is to coarsen \( G_0 \) to the coarsest graph \( G_i \), such that \( |G_i| = k + 1 \) and all the vertex-weights fulfill the multiple constraints defined. Note that the multinode U from the previous step is treated as a normal vertex during coarsening. The final coarsest graph \( G_i \) consists of \( k + 1 \) multinodes. The multinode including U becomes the unoffloadable partition \( V_U \) and the other \( k \) multinodes are the \( k \) offloadable partitions, i.e. \( V_O^1, V_O^2, \ldots, V_O^k \). This step is described by the following pseudo-code.

The \((k + 1)\) coarse partitioning algorithm:

\begin{enumerate}
\item \textbf{begin}
\item Define multi-constraints of partitions: \( T^U, \delta^U, T_1^O \ldots T_k^O \) and \( \delta_1^O \ldots \delta_k^O \);
\item \( \psi^U = 0; \psi_1^O = \cdots = \psi_k^O = 0; \) // sum of vertex-weights
\item Appoint the multinode including U as \( V_U \);
\item \textbf{while} \(|V| > k + 1\) \textbf{do} // if the graph is not coarse enough
\item Appoint \( k \) weightiest multinode as \( V_1^O \ldots V_k^O \);
\item \textbf{if} \( \psi^U \geq (T^U - \delta^U) \) \textbf{then} Mark multinode \( V_U \) as matched;
\item \textbf{for} \( m = 1 \) to \( k \) \{ // check the \( k \) partitions
\item \textbf{if} \( \psi_m^O \geq (T_m^O - \delta_m^O) \) \textbf{then} Mark \( V_m^O \) as matched; \}
\item \textbf{if} \( V_U, V_1^O, \ldots, V_k^O \) are matched and \textbf{hasUnmatchedVertex()} \textbf{then}
\item \textbf{Partition Failure};
\item Invoke HELVM Algorithm to coarsen the graph;
\item Update \( V_U, V_1^O, \ldots, V_k^O \) to add new merged vertices;
\item Update \( \psi^U, \psi_1^O, \ldots, \psi_k^O \) to add new merged vertex-weights;
\item \}
\item \textbf{end}
\end{enumerate}

In lines 7 and 9, the total weight of each multinode (i.e. partition) is checked to examine whether it has already reached the \textit{lower bound} of the predefined cost constraint. If yes, it is marked as matched and no more vertices will be added in. If all partitions satisfy the cost constraints and there is still an unmatched vertex left, the partition is a failure (lines 10 and 11), which means the partitions cannot be found under predefined constraints. In this case, either the constraints need to be lightened or the service executes without being offloaded. In line 12, the HELVM algorithm is invoked for graph coarsening.

5.3. The HELVM algorithm

The Heavy-Edge and Light-Vertex Matching (HELVM) algorithm is the core of the \((k + 1)\) coarse partitioning. At the graph coarsening phase, a sequence of successively
coarser graphs $G_1, G_2, \ldots, G_n$ is constructed from the graph $G_0$ such that $|V_i + 1| < |V_i|$, i.e., the number of vertices in the successively coarser graph is smaller. Two main approaches have been proposed in [16] for coarsening a graph. The first is to merge the highly connected vertices into a multinode, while the second is to find a matching and then to collapse the matched vertices into a multinode. We adopt these two approaches in our HELVM algorithm. HELVM coarsens the graph by collapsing the heavy edges.

A matching of a graph is a subset of edges with no two edges incident upon the same vertex. The task of finding a maximum matching is to select a maximum subset of such edges. The coarser graph $G_{i+1}$ is constructed from $G_i$ by finding a matching of $G_i$ and collapsing the matched vertices into multitudes. The unmatched vertices are simply copied over to $G_{i+1}$. Since the goal of collapsing vertices using matching is to decrease the size of the graph $G_i$, we are trying to find the maximum matching of the graph $G_i$.

For finding maximum matching, Karypis and Kumar proposed Random Matching (RM), Heavy Edge Matching (HEM), and Light Edge Matching (LEM) in [16, 17]. All these heuristics only consider the edge-weights, however. In the context of service partitioning for offloading systems, they are not sufficient as most of the resource constraints are related to the vertex-weights. The work in [18] considered the vertex-weights; however, its focus is to balance constraints in partitions by selecting vertex-pairs with minimized difference for matching.

In our offloading middleware, we aim to keep more highly connected vertices in one partition, satisfying the multiple constraints. In the HELVM algorithm, the heavy-edge means the incident vertices are tightly connected (i.e. with heavy edge-weight); whereas the light-vertex means that more vertices will be merged under the predefined constraints. The basic idea of the algorithm is that when selecting an edge for matching, instead of only comparing the edge-weight, we also compare the vertex-weights of the incident vertices. We use a vertex-and-edge-composite-weight to scale the weight of an edge and its incident vertices. The vertex-and-edge-composite-weight of vertex $v$ in relation to vertex $u$ is:

$$CW(u, v) = \lambda_1 w(e(u, v)) + \lambda_2 / w(v)$$

(4)

where $w(e(u, v))$ is the edge-weight, and $w(v)$ is the composite-vertex-weight of $v$ calculated by Eq. (1); $\lambda_i$ ($i = 1, 2$) ($0 \leq \lambda_1, \lambda_2 \leq 1$ and $\lambda_1 + \lambda_2 = 1$) are the important factors of edge-weight and vertex-weight, respectively. If $\lambda_2 = 0$, then the HELVM algorithm becomes heavy-edge matching.

If vertex $v$ is selected to match with $u$ due to the vertex-and-edge-composite-weight $CW(u, v)$ is maximum, then $v$ is called the tightest-and-lightest vertex in relation to $u$. If there is more than one vertex in relation to vertex $u$ that has the same maximum vertex-and-edge-composite-weight, then one of them is selected by the following approach: let $H$ be the set of such tightest-and-lightest vertices, the vertex is selected if

$$\text{Adj}_W(u, v) = \sum_{e(v, y) \in E \land (e(u, y)) \in E \land y \neq u} w(e(v, y))$$

(5)

is maximized; i.e., the weight sum of the edges that connect $v$ to the vertices which are also adjacent to $u$ is maximized. That means to choose the one not only tightly linked with vertex $u$ but also tightly linked with the vertices adjacent to $u$. If still more than one is found, select the first one or a random one in $H$. 
The HELVM algorithm is described as follows:

The HELVM algorithm:
1. **begin**
2. Mark all vertices of vertex set $V$ as unmatched;
3. while hasUnmatchedVertex() do {
4. $u = \text{RandomSelectUnmatchedVertex}()$;
5. $v = \text{GetTheTightestLightestVertex}(u, V)$;
6. if ($v \neq \text{null}$) then {
7. Put edge $e(u, v)$ into the matching;
8. Mark $v$ as matched vertex; }
9. Mark $u$ as matched vertex;
10. }
11. **end**

The $\text{GetTheTightestLightestVertex}$ function is used to select the tightest-and-lightest vertex, which is implemented as follows:

The tightest-and-lightest vertex selection function:
1. **Function** $\text{GetTheTightestLightestVertex}(u, V)$
2. **Input**: $u$ - the given vertex; $V$ - the vertex set
3. **Output**: the tightest-and-lightest vertex in relation to $u$
4. **begin**
5. $\text{Adj}[n] = \text{GetUnmatchedVerticesAdjacentTo}(u)$;
6. $\text{CurrentTightestLightestVertex} = \text{null}$;
7. $\text{CurrentMaxCW} = 0$; // the maximum composite-weight
8. $\text{NumberOfTightestLightestVertex} = 0$;
9. $H = \text{null}$; // The TightestLightestVertex set, $H$
10. for $i = 1$ to $n$ {
11. $\text{CW}(u, \text{Adj}[i]) = \text{GetCompositeWeightOf}(u, \text{Adj}[i])$;
12. if $\text{CW}(u, \text{Adj}[i]) \geq \text{CurrentMaxCW}$ then {
13. if $\text{CW}(u, \text{Adj}[i]) > \text{CW}(u, \text{Adj}[i])$ then {
14. $\text{CurrentMaxCW} = \text{CW}(u, \text{Adj}[i])$;
15. $H = \text{null}$; // Clear the TightestLightestVertex set;
16. $\text{NumberOfTightestLightestVertex} = 1$;
17. } else {
18. $\text{NumberOfTightestLightestVertex} + +$; }// more then one
19. $H.add(\text{Adj}[i])$; //Put current vertex into the set $H$
20. }
21. }
22. if $\text{NumberOfTightestLightestVertex} > 1$ then
23. $\text{CurrentTightestLightestVertex} = \text{GetMostTightestLightestVertex}(H)$; // by Eq. (5)
24. **return** $\text{CurrentTightestLightestVertex}$;
25. **end**
By using the HELVM, a maximum matching can be found. The vertices being matched will be collapsed into multinodes. The weight vectors of the multinodes are set equal to the sum of the weight vectors of the vertices being merged. Meanwhile, to keep the connectivity information in the coarser graph, the edges of a multinode are the union of the edges of vertices being merged.

Fig. 6 shows an example of how to select the tightest-and-lightest vertex for matching. The vertices in the areas circled by dotted lines are already matched. There are still five unmatched vertices: c, d, e, f and g. Suppose vertex d is randomly selected now, one of its unmatched adjacent vertices: c, e, f and g, will be selected to match with d. As shown in the figure, the vertex-weight of d is \( w(d) = 2 \). By using Eq. (4), we calculate \( CW(d, x) \), for \( x \in \{c, e, f, g\} \), to select the tightest-and-lightest vertex. Let \( \lambda_1 = \lambda_2 = 0.5 \) (i.e. the edge-weight and the vertex-weight have the same importance). The vertex-and-edge-composite-weight of the four candidate vertices in relation to d are calculated by:

\[
\begin{align*}
CW(d, c) &= 0.5 \times 1 + 0.5/2 = 0.75 \\
CW(d, e) &= 0.5 \times 2 + 0.5/2 = 1.25 \\
CW(d, f) &= 0.5 \times 2 + 0.5/1 = 1.5 \\
CW(d, g) &= 0.5 \times 2 + 0.5/1 = 1.5.
\end{align*}
\]

We need to select the maximum one as the tightest-and-heaviest vertex so as to put the related edge into the matching. As can be seen, in relation to d, there are two tightest-and-heaviest vertices: f and g. Eq. (5) is then used for further selection. For vertex f, it is connected to e and g which are adjacent to d, so:

\[
\text{Adj} W(d, f) = w(e(f, e)) + w(e(f, g)) = 2 + 1 = 3.
\]

For vertex g, it is connected to b and f which are adjacent to d, so:

\[
\text{Adj} W(d, g) = w(e(g, b)) + w(e(g, f)) = 1 + 1 = 2.
\]

As \( \text{Adj} W(d, f) > \text{Adj} W(d, g) \), the vertex f is finally selected as the tightest-and-heaviest vertex and the edge \( e(d, f) \) is put into the matching.
5.4. Complexity of the \((k + 1)\) partitioning algorithm

The first step of the algorithm merges all unoffloadable vertices. It traverses each vertex in the multi-cost graph to examine if it is unoffloadable. In the worst case, all the vertices may need to be examined. Thus, the worst case complexity of this step is \(O(|V|)\).

In step 2, the algorithm coarsens the multi-cost graph by using the HELVM algorithm. The worst case is that all the vertices are mesh connected. In this case, the complexity of the \(GetTheTightestLightestVertex\) function is \(O(|V|)\). Accordingly, the complexity of HELVM algorithm is \(O(|V|^2)\). Thus, the worst case complexity of step 2 is \(O(|V|^3)\).

6. Practical issues

As discussed in Section 3, the costs of each service component change during service executions. In general, there are two approaches to weighing the costs of service components: online-profiling and offline-profiling. The latter uses a pre-defined profile to describe the resource consumption of each service component. An offline-profile needs to be defined during the design period and with the knowledge of the source code. The competitive advantage of offline-profiling is its simple implementation. If an offloading is needed before the service’s execution, the offline-profiling has to be used. The drawback of offline-profiling is that it cannot reflect dynamic resource changes during application execution. On the other hand, the online-profiling approach generates a multi-cost graph dynamically by monitoring the runtime execution environment. It does not need knowledge of the source code. The demerit of online-profiling is that it introduces extra overhead due to its real-time and dynamic nature. Our offloading middleware supports both online-profiling and offline-profiling. These two methods are evaluated in our experimental environment.

For making offloading decisions, there are two viable approaches for monitoring the runtime resource utilization of the environment: interval-based and event-driven resource snapshots. The first takes resource utilization snapshots by a time interval. The interval can be smaller for more precise evaluation, such as a few milliseconds. However, frequent snapshotting will increase overhead. Whereas, by subscribing system resource allocation and release events (e.g. the events of JVM’s memory allocation and release), the event-driven approach takes snapshots only when resource utilization changes. In our implementation, we take the event-driven snapshots and complement them with a periodical triggering of longer interval snapshots (e.g. a few seconds). The following approaches are used to obtain the cost value of edges and vertices in the multi-cost graph.

- **Memory usage**: The memory usage of a class changes during its execution. The resource monitoring module obtains the memory usage of each class by monitoring the JVM heap.
- **CPU Utilization**: It is difficult to measure CPU utilization. For a given time slot, if only one class is running, the CPU utilization can go to 100%. If two classes are running and the CPU utilization is 100%, it is unreasonable to say that the classes’ CPU utilizations are 50% each or in some kind of ratio. Obviously, the percentage of CPU occupied is not a feasible way to represent a class’s CPU utilization. We use an objective and simple
approach to tackle this issue. We calculate the cumulated CPU occupying time of a class during the snapshot intervals and normalize it as the CPU processing cost of this class.

• **Bandwidth usage**: Network bandwidth usage comes from two aspects: data access between the mobile device and surrogate(s), and the mobile device exchanging data with remote hosts (e.g. downloading or uploading). The resource monitoring module monitors remote data accesses to obtain the bandwidth usage of each class.

• **Invocation and data access number (i.e. edge-weight)**: The number of invocations between classes is relatively easy to measure. All the method invocations can be monitored by the invocation stacks. As the granularity of service partitioning is at the level of the Java class, only the interactions between classes are recorded. For example, there are two methods, \(a\) and \(b\), which are located in two classes \(A\) and \(B\), respectively. Let \(I^{\{a,b\}}\) be the set of method invocations from \(a\) to \(b\), the invocation number between \(A\) and \(B\) is the cardinal of \(\{I^{\{a,b\}} \cup I^{\{b,a\}} | a \in A \land b \in B\}\). The number of data accesses can be obtained in the same manner.

The importance factors \(\varepsilon_1\), \(\varepsilon_2\), and \(\varepsilon_3\), defined in Eq. (1), are utilized to weigh different type of resources, i.e., memory, CPU, and bandwidth, respectively. The values of these factors can be determined using one of the following three means: (1) their values can be set by the service designers based on the knowledge of the source code. (2) Alternatively they can be directly chosen by the end users according to their real-time scenarios. For instance, if the memory utilization is low in the mobile device, the user can lower the memory factor and increase the CPU and bandwidth factors in order to save battery time. (3) These values can be dynamically decided by the offloading systems according to resource availabilities in the mobile device.

7. Implementation and evaluation

7.1. Implementation

A testbed simulating the real-life scenario as demonstrated in Fig. 1 has been set up, which consists of two sub-networks: one simulating the home network and the other the airport lounge network. In the home network, the home network server and the home gateway are desktop PCs (P4-2 GHz, 1 GB Memory). In the airport lounge network, a desktop PC of the same specification is used to serve as a surrogate. The mobile device used is an HP iPAQ HX2750 PDA. Wireless communications are through IEEE 802.11b WiFi links, Desktop PCs and APs are connected in a switched 100 Mbps Ethernet. The PDA has Microsoft Pocket PC 2003 as its operating system and the JVM used is IBM J9. The desktop PCs are installed with Redhat Linux 9 and Sun’s J2SE JVM version 1.5.0. All these hosts have the offloading middleware installed.

A screenshot of the offloading middleware test application running at the mobile device side is illustrated in Fig. 7. It shows a user interface allowing a mobile device user to carry out the following operations:

• To monitor the system resources such as the memory, CPU and bandwidth utilizations;
• To select the available surrogates based on their properties (such as performance);
• To manually offload a running pervasive service;
Fig. 7. User interface of offloading system test application.

- To confirm or discard an offloading solution prompted by the system;
- To monitor running and offloaded pervasive services.

7.2. Evaluation

The evaluation of the proposed middleware is carried out by utilizing various applications running on mobile devices. These applications are used as carriers or demonstrators to show different performance aspects of the middleware. Four experiments are carried out to evaluate various aspects of the middleware. One is to validate the proposed offloading mechanism. The second is to validate the efficiency of the \((k + 1)\) service partitioning algorithm. The third and the fourth pick up specifically on an application demanded by mobile device consumers such as John in the above scenario: Video Explorer. These four experiments aim to justify the introduction of an offloading mechanism into mobile devices and associated home networks and their extension (e.g. the airport lounge network in John’s scenario) and to demonstrate its potential to contribute to the prompt embrace with efficient service provisioning such as quality video processing on mobile devices.

Energy consumption concern: since the mobile devices concerned here are battery-rechargeable, we put response time to users at the first priority, especially when these two factors (response time and energy consumption) conflict with each other. The offloading decision engine can be adjusted to take energy consumption into consideration. Our future work will evaluate the relationship between energy consumption and other measures of
the middleware such as response time. In this paper we assume that energy saving is a secondary concern in comparison with user-perceived parameters such as response time.

7.2.1. Experiment 1: π calculator

We developed a π calculator for testing the offloading middleware. It consists of two Java classes: PiCalculator and Pi. The class Pi computes the value of π, whereas the class PiCalculator handles the graphic user interface (GUI) which firstly gets user’s input for the accuracy required (i.e. how many decimal places) of π; and then invokes the π calculation function in class Pi; and, finally, outputs the result. The π calculator is computation intensive.

The π calculator runs in three cases: mobile device only, using the offloading middleware (Offloading), and entirely running in surrogate (Surrogate Only). In this experiment, the offline-profiling is used and only one surrogate is engaged.

Fig. 8 shows the time consumption, memory usage and CPU utilization for the calculation, respectively. The Y-axes represent the resource usage and the X-axes represent the accuracy of π (i.e. decimal places). The curves in Fig. 8(a) show that the response time in the mobile-device-only case is the slowest; it gets faster in the offloading case because the class Pi is offloaded to the surrogate to take advantage of the surrogate’s rich computational resources. The surrogate-only case has the quickest response since all the classes are running on the surrogate. Note that this specific application does not involve much inter-class interaction. Fig. 8(b) and (c) show that the memory usages and CPU utilizations in the mobile device are significantly decreased in the offloading cases. The reason for higher memory usage in the surrogate-only case is the larger JVM heap size setting for providing more memory space for offloading in the surrogate.

As can be seen in the figures, the offloading middleware itself caused some overhead. If the calculation is carried out to less than 100 decimal places, the time used for offloading is longer than non-offloading. Memory usage and CPU utilization remain the same if the calculation is performed to less than 80 and 50 decimal places, respectively.

7.2.2. Experiment 2: M4Play partitioning

As provision of multimedia pervasive services to mobile devices becomes popular, we utilize a Java MPEG-4 player, M4Play, to conduct an experiment for testing our offloading middleware, particularly, the class instrumenting and service partitioning algorithm. M4Play is an MPEG-4 video/audio player provided by IBM Toolkit for MPEG-4 [19].

We use our offloading middleware to load the IBMToolkitForMpeg4.jar file. Note that no source code is provided for this toolkit. The offloading middleware directly loads binary bytecode from the jar file. There are a total of 1871 classes included in this jar file. Most of the names of classes are obfuscated to meaningless strings (as shown in Fig. 9). Since the package also includes classes for audio/video generation and interactive video processing, only 374 classes are involved for the purposes of MPEG-4 video clip playing.

There are 85 system classes, including: java.io.*, java.lang.*, javax.imageio.*, java.awt.image.*, javax.sound.*, and java.util.*. For the sake of showing the relationship between the system classes and the non-system classes, we keep all these system classes in the graph. There are a total of 49 unoffloadable classes filtered out according to certain unoffloadable rules. Fig. 9 shows a snapshot of the multi-cost graph of the M4Play. The
black-filled circles represent the unoffloadable classes and the grey-filled circles represent the offloadable classes. Each class is annotated with the class’s name and a 3-tuple weight. For example, the arrow in the bottom right corner of the graph points to an offloadable class named \textit{avc} that is annotated with a 3-tuple cost value $(16, 64, 0)$; these values represent the accumulated CPU time (in the unit of ms), the memory usage (kb), and the bandwidth usage (kb/s), respectively. Each edge in the graph is labelled with its weight; it represents the number of interactions and data access between two classes. Let $k = 2$, our $(k + 1)$ partitioning algorithm partitions the multi-cost graph of \textit{M4Play} into a unoffloadable partition, which is for running in the mobile device, and two partitions for running in two surrogates. \textbf{Fig. 9} shows the partitioning. The cost constraints and
partitioning parameters are listed in the top right corner. The edge-cuts between three partitions are: $C(U, O_1) = 312$, $C(U, O_2) = 126$, and $C(O_1, O_2) = 210$.

7.2.3. Experiment 3: VideoExplorer partitioning for edge-cut comparison

In this experiment, we compare our HELVM algorithm with Random Matching (RM), Heavy Edge Matching (HEM), and min-cut algorithms by partitioning services in which
the constituent classes have different features. For example, some classes are computation intensive, whereas some of them are memory or bandwidth intensive. We designed a Java program, VideoExplorer, which integrates the functionalities of M4Play and AVgen (an MPEG-4 audio/video generator in IBM Toolkit for MPEG-4 [19]) and also includes some dedicated classes to download files from a remote host. The program executes the following sequences: (1) downloads a video sequence and audio sequence from a remote host; (2) generates an MPEG-4 format video clip from the downloaded sequences; (3) plays the MPEG-4 clip.

The program includes classes with different intensives. The classes involving MPEG-4 generation are computational and memory intensive. The classes for remote file downloading are bandwidth intensive. The classes for MPEG-4 playback are computation intensive. Different matching algorithms are implemented in our offloading system to partition VideoExplorer. For comparing with min-cut (it only partitions an application into two partitions), k is set equal to 1. The partition parameters are set as following: $T^U = 1000 \sim 5000$, $\delta^U = 200$, $T^{O_1} = 25000$, $\delta^{O_1} = 300$, $\lambda_1 = \lambda_2 = 0.5$, $\epsilon_1 = 0.35$, $\epsilon_2 = 0.35$, $\epsilon_3 = 0.3$. In our evaluation network environment, since we assume that the bandwidth resource is relatively rich in comparison with the other two resources, $\epsilon_3$ is set to a smaller value. $\epsilon_1$ equals to $\epsilon_2$ due to the assumption that memory and CPU being equally important in this VideoExplorer application.

Fig. 10 shows all the edge-cuts with the change of $T^U$ (the threshold of the unoffloadable partition). The values of RM are big and without regularity; this reflects its random-selecting feature. The edge-cuts of HEM get bigger when $T^U$ gets larger. The min-cut gets smaller edge-cuts when $T^U \leq 2000$. However, in the rest of the cases, it gets bigger edge-cuts. Due to the HELVM selecting the tightest and the lightest vertex for matching, it gets significantly smaller edge-cuts and it is threshold insensitive.

### 7.2.4. Experiment 4: VideoExplorer offloading

In this experiment, we use our offloading middleware to partition, to offload and to run VideoExplorer. For comparison, we implemented our offloading middleware in two different ways. One is the normal offloading as described above — named Normal Offloading. The other one caches offloaded classes in the surrogate for performance improvement — Cached Offloading. In the latter case, before a class is migrated to a surrogate, our middleware first of all checks if the class is already cached in the surrogate.
If it is cached, the class does not need to be migrated again. The overhead of this checking is measured in terms of message size in bytes. Each class is identified by an ID which is of length $\lceil \log_2 n \rceil$ where $n$ is the number of offloading classes. Then the size of the checking message is $n \lceil \log_2 n \rceil / 8$ B, which is very low. However, the related runtime status always needs to be sent to the surrogate in either case.

We compare response time and bandwidth requirement when running VideoExplorer on the PDA to download and playback a 10 s MPEG-4 encoded video clip. The results in response time are listed in Table 1, which shows a clear advantage of offloading over non-offloading in video applications. The results also indicate a slight gain for cached offloading over non-cached offloading in terms of response time. As the video size becomes bigger this advantage will also become greater.

Fig. 11 shows the bandwidth requirements of the three cases during VideoExplorer execution. In the normal offloading case, mobile device used the first 9 s to offload where bandwidth requirement was high; then the surrogate carried out video download while the mobile device waited almost idle (consuming almost no bandwidth) for about 4 s. After the end of video downloading at the surrogate side, the mobile device started to play the video which was decoded by the surrogate. In this stage, bandwidth requirement soared up again for 10 s. Then the whole playback procedure finished after 25 s. The Cached offloading mechanism showed a similar curve development with a shift to left. This bandwidth saving is explained by the fact that cached classes in surrogates do not need to be offloaded again.
In non-offloading case, the mobile device used bandwidth consistently for a longer period of time and the whole playback process used about 52 s — more than double the time of the offloading mechanisms. The bigger the video size, the smaller the proportion of the overall application execution time is accounted for by the class offloading phase, and the more benefit can be achieved.

Another observation is that, in the scope of this particular experiment, offloading consumes more bandwidth than non-offloading at a given active time point. This increase in bandwidth usage is not caused by side effects such as signaling or control messages but rather by the fact that decoded raw video, which is bigger in size, is transmitted between the mobile device and the surrogates. The main goal of this experiment is to reduce the response time and to save the CPU time for transcoding and decoding, as a showcase implementation of the John’s scenario described at the beginning of the paper. This is under the assumption that the connection between John’s mobile device and the surrogate is stable and that the bandwidth resource is not important compared with the CPU and memory resources.

8. Conclusion and future work

In this paper, with a real-life scenario, we presented an offloading middleware for mobile devices, with specific focus on its two core modules: instrumenting and partitioning modules. The partitioning algorithm considers both edges and vertices when deciding a merger. Specifically, our algorithm considers different types of costs associated with individual application classes (including CPU cycle, memory and bandwidth) and the costs amongst these classes (mainly invocation frequency). Since this middleware works on Java bytecode instead of source code, increases confidence in its practicability and efficiency. The middleware system has been validated via a scenario prototyping. The experimental results have demonstrated its effectiveness and efficiency.

Currently, an efficient surrogate discovery algorithm operating in converged wired and wireless networks is under development. The development of a fault-tolerance mechanism that deals with potential risks of an offloaded task failing during its remote execution is also our near-term plan. Introducing security mechanisms into the middleware system is our long-term future plan.

Acknowledgement

The work of this paper was partially funded by the UK EPSRC (Engineering and Physical Sciences Research Council) under the Project PANDA (EP/D061881/1).

References


Shumao Ou is currently a Ph.D. student in the Department of Electronic Systems Engineering, University of Essex, UK. He obtained an M.Sc. with distinction from the same department and a B.Eng. from the National Technical University of Defence, China. His research interests include pervasive computing, mobile computing, pervasive service engineering and model driven engineering. He has published several papers in journals and conferences.

Kun Yang received his Ph.D. from the Department of E&EE of University College London (UCL), UK, and M.Sc. and B.Sc. from the Computer Science Department of Jilin University, China. He has been a lecturer in the Department of Electronic Systems Engineering, University of Essex, UK since 2003. Before that he worked as a research fellow at E&EE-UCL on several EU IST projects. His main research interests include wireless networks, mobile ad hoc and sensor networks, pervasive service engineering. He has published more than 20 journal papers and books. He serves on the editorial board of IEEE Communication Survey & Tutorials and Wiley Int. Journal of Wireless Communication and Mobile Computing (WCMC). He is a chair or TPC member of many international conferences. He is a Member of IEEE and ACM.
Jie Zhang is Professor of Wireless Communications and Networks at the Dept. of Computing and Information Systems, University of Bedfordshire (UoB), UK. He joined UoB as a Senior Lecturer in 2002, becoming Reader and Professor in 2005 and 2006 respectively. He received his Ph.D. degree from the Dept. of Automatic Control and Electronic Eng., East China Univ. of Sci. and Tech., Shanghai, China, in 1995. From 1997 to 2002, he was a Research Fellow with E&E Eng., University College London, the Dept. of Eng. Sci., Oxford University, and a Software Engineer with Aircom. He is the founding Director of Centre for Wireless Network Design, UoB. He is the Project Coordinator of EU FP6 RANPLAN-HEC and GAWIND projects. As the Principal Investigator, he is currently responsible for some US$ 2.5M research funding. His main research interests include Wireless Communications and Networks, QoS in IP networks and High Performance Computing. He has some 50 publications in refereed journals and conferences.