Transparent mobile middleware integration for Java and .NET development environments

Edgar Marques, Luís Veiga, and Paulo Ferreira

Distributed Systems Group at INESC-ID/Technical Univ. of Lisbon,
Rua Alves Redol N. 9, 1000-029 Lisbon, Portugal
emarques@gsd.inesc-id.pt,{luis.veiga,paulo.ferreira}@inesc-id.pt
http://www.gsd.inesc-id.pt

Abstract. Developing a distributed application for mobile resource constrained devices is a difficult and error-prone task that requires awareness of several system-level details (*e.g.*, fault-tolerance, ...).
Several mobile middleware solutions addressing these issues have been proposed. However, they rely on either significant changes in application structure, extensions to the programming language syntax and semantics, domain specific languages, cumbersome development tools, or a combination of the above. The main disadvantages of these approaches are lack of transparency and reduced portability.
In this paper we describe our work on enabling transparent integration between applications and middleware without changing application structure, extending the programming language or otherwise reducing portability. We used the OBIWAN middleware but our solutions are general. To achieve this goal we employ program analysis and transformation techniques for extending application code with hooks for calling middleware services. Application code extension is performed automatically at compile-time by a code extension tool integrated with the development environment tool set. We describe the implementation of our .NET and Java prototypes and discuss evaluation results.

Key words: distributed mobile applications, fault-tolerance, incremental object replication, transparent middleware integration, program transformation, aspect-oriented programming, integrated development environment

1 Introduction

Mobile devices like cellular phones and other resource constrained devices are inherently hard to program. CPU power, memory footprint and battery consumption are examples of some of the issues developers must be constantly aware of when developing standalone, connectionless applications for such devices.

The development of network applications over mobile networks is even more difficult. In addition to the afore mentioned issues associated with these devices, the developer must also handle issues like loss of connectivity, variable bandwidth (due to variation of link signal quality) and increased battery consumption (while connected). All these issues play an important role on design decisions specific to network applications like fault tolerance, shared data consistency, object location and security.

Over the years, several middleware solutions for developing applications running over fixed and mobile networks have been proposed [15, 2, 3]. Each one of them addresses

one or more design issues while exposing the developer to a simpler and higher level view of the system, with the purpose of reducing development effort and allowing him to focus on business logic.

However these solutions suffer from lack of transparency and reduced portability, due to the following reasons: i) inflexible and awkward application structure, ii) employment of domain specific languages or programming language extensions, and iii) cumbersome development tools.

In this paper we describe our work on enabling transparent integration between applications and a research mobile middleware called OBIWAN [7]. The OBIWAN mobile middleware is a platform developed for the purpose of aiding the development of distributed applications running on resource constrained mobile devices. Currently there are two prototypes of the middleware running on the .NET Common Language Runtime and the Java Virtual Machine (both standard and mobile editions). The middleware requires no changes to the underlying virtual machine thus assuring portability among a wide range of devices. The OBIWAN middleware provides several features which have been discussed in previous work [19, 21, 16, 20]. It's important to note that we could have used other middleware instead of OBIWAN as our solutions are general.

The motivation for this work started from the fact that despite the various features provided by OBIWAN, interacting with the middleware was complicated and tedious even for simple tasks. The developer not only had to issue several API calls, but he also had to write additional code and data structures for the sole purpose of interfacing between the application and the middleware.

A first step towards making the interaction with the middleware easier was to write a helper tool for automatically generating some of the required code and data structures. This tool, called *obicomp* [7, 19], made use of reflection in order to perform simple program analysis. However, it was very limited in scope (*e.g.*, it couldn't analyse method bodies) and it imposed an awkward process for building applications (because the tool was based on reflection, the developer had to partially build the application before running the tool and then build it again after including the generated code). Furthermore, the developer still had to manually issue the required API calls.

To overcome these faults, we redesigned and rewrote the *obicomp* tool from scratch to achieve seamless integration with the programming languages (either Java or C#) and integrated development environments (Eclipse [6] or Visual Studio 2005 [22]). The new version employs program analysis and transformation techniques for extending application code with hooks for calling middleware services. The tool runs at compile-time and it directly manipulates application source code. By manipulating source code instead of bytecode, the generated code is not only easier to understand and verify as it is also more portable, running on a wider range of devices. It also makes the tool easier to integrate with existing development tool sets.

The tool currently performs source code analysis and requires the developer only to specify the classes, packages/assemblies or fields whose behaviour is to be extended by the mobile middleware. To ensure maximum portability, we chose not to add any extensions to the programming languages, relying instead on standard language constructs (*attributes* in the case of the C# language and *annotations* in the case of the Java language).

In summary, the contributions of this work are:

– A framework for transparent middleware integration based on automatic code generation and without sacrificing portability, *i.e.*, without requiring any changes to programming languages or underlying virtual machines.
– Measurements of the performance penalty on compilation times introduced by the code generation stage.
– Identification of usability issues associated with our solutions.

The remainder of this paper is organized as follows. Section 2 presents an overview of the middleware integration architecture, including integration with existing tool sets. Section 3 exposes the implementation of the code generation tools for the C# and Java languages and how these were integrated with existing development environments for these languages. Section 4 presents performance results for the code generation tools with respect to possible overhead to regular project/solution compilation and deployment. Current usability issues are also reported. In Sections 5 and 6, we compare our work with others' and draw some conclusions, respectively.

2 Middleware integration architecture

As we mentioned earlier, transparent integration in OBIWAN is accomplished through the use of a code generation tool for:

– Automatic generation of middleware service calls.
– Automatic generation of additional auxiliary code and data structures.

Because the nature of the generated code depends mostly on the specific feature being used, a concrete example is necessary for better clarification. For this purpose and without loss of generality we focus on the particular case of incremental object replication.

In the next subsection we give a brief overview of incremental object replication and describe the necessary API calls and data structures that are required for integrating applications and middleware.

We conclude this section with an explanation of the generic code generation architecture of our development tools and how they integrate with existing tool sets.

2.1 Example: incremental object replication

Object replication improves both availability and scalability of distributed applications. By allowing individual nodes to work on local replicas of shared data, network failures and quality of service concerns can be masked from the application, thus permitting uninterrupted execution even in connectionless scenarios.

Due to the resource constrained nature of mobile devices (e.g., limited battery power, scarce available memory, etc.), incremental object replication is employed in OBIWAN. Since only the objects that are actually required by the application are replicated, this technique reduces the total amount of memory used by the application and the bandwidth required for message exchange between processes. Battery power consumption is also reduced because of reduced network usage and the fact that the radio interface is only activated when needed.

Overview of incremental object replication Incremental object replication allows mobile applications to fetch data from remote processes as they need. When an object is replicated from a remote process, all references that object has to other objects are replaced with references to proxy objects. When the application needs to access an object which is represented by a proxy object, the proxy object fetches a replica of the remote object it represents and replaces itself with that replica. The proxy object is no longer referenced by any other object in the local process and the garbage collection mechanism will reclaim it.

The whole process is transparent from the application's point of view: it is the proxy's responsability to request a new object replica when the application needs it. The application can't tell the difference between a replica and its proxy because they both implement the same interfaces [7, 19]. A thorough discussion of incremental object replication is given in [19].

Integration of incremental object replication We now describe the data structures and API calls that are automatically generated to add support for incremental object replication in a distributed mobile application. The full generated code is quite extensive so we only show a small subset for explanation purposes. Additional details can be found in [19]. All code examples are written in the C# language and are quite similar to the Java versions. The main difference between the two versions is the usage of *attributes* in C# and of *annotations* in Java.

Consider the following example of a C# class for implementing an incrementally replicatable singly-linked list. We use a singly-linked list for clarity; other more complex data structure would complicate the description. Each node of the list is replicated on demand while it is being traversed. Notice the use of the C# attribute `[Replicatable]` for denoting that support for incremental replication should be added to the class. This annotation must be explicitly specified by the developer since *obicomp* isn't capable of telling replicatable types and non-replicatable ones apart on its own. In Java the equivalent annotation `@Replicatable` would be used instead.

An example C# class implementing an incrementally replicatable singly-linked list.

```
using INESCID.GSD.Obiwan;

[Replicatable]
public class SinglyLinkedList {
  private SinglyLinkedList _next;
  public SinglyLinkedList GetNext() { return _next; }
  public void SetNext(SinglyLinkedList next) { _next = next; }
}
```

obicomp analyses the source code of the class and automatically generates the interface definition for it (which also includes *properties* in the case of the C# language), as shown in the next code snippet. Notice how *obicomp* replaced every type reference to the class with a type reference to its interface.

C# interface generated from the example C# class.

```
public interface ISinglyLinkedList__Obiwan__ {
```

```
  public ISinglyLinkedList__Obiwan__ GetNext();
  public void SetNext(ISinglyLinkedList__Obiwan__ next);
}
```

This is necessary so that the application is unable to tell the difference between a replica and a proxy. The same type reference replacement is made throughout the entire application, including the replicatable class itself. The next code snippet shows the class after code extension. Notice how the class now implements not only the newly generated interface but also other auxiliary interfaces (`IObiwanObject`, `IDemander` and `IProvider`) that handle automatic replica creation and proxy replacement.

The same C# class after code extension.

```
using INESCID.GSD.Obiwan;

[Replicatable]
public class SinglyLinkedList : ISinglyLinkedList__Obiwan__,
  IObiwanObject, IDemander, IProvider {
  private ISinglyLinkedList__Obiwan__ _next;
  public ISinglyLinkedList__Obiwan__ GetNext() { return _next; }
  public void SetNext(ISinglyLinkedList__Obiwan__ next) { _next = next; }
  ...
}
```

These auxiliary interfaces define callback methods which are invoked by the middleware during the incremental replication process. Further information on these callbacks is given in [19].

obicomp also generates the corresponding proxy type for the replicatable class. The proxy implements the generated interface and other auxiliary interfaces that allow the middleware to handle association with the actual object and its parent (refering) object.

Generated C# proxy-out class for the example C# class.

```
using INESCID.GSD.Obiwan;

public class SinglyLinkedListProxyOut__Obiwan__ :
  ISinglyLinkedList__Obiwan__, IObiwanObject, IDemandee {
  private ISinglyLinkedList__Obiwan__ _replica;
  public ISinglyLinkedList__Obiwan__ GetNext() {
    _replica = (ISinglyLinkedList__Obiwan__) this.Demand();
    return _replica.GetNext();
  }
  public void SetNext(ISinglyLinkedList__Obiwan__ next) {
    _replica = (ISinglyLinkedList__Obiwan__) this.Demand();
    _replica.SetNext(next);
  }
  ...
}
```

The proxy type implements the object fault mechanism. A proxy object stands in place for an object not replicated yet. When the application invokes a method

of a proxy object, it is handled as an object fault, and the proxy object calls the middleware object replication service through its `Demand` method. The method returns the replicated object and the proxy redirects the initial method call to it. Although not shown in the previous examples, the `Demand` method also updates the object reference that points to the proxy object to point instead to the newly replicated object. This is done by calling the callback methods added to the extended C# class.

2.2 Generic code generation architecture

Our approach is based on program transformation techniques that are also commonly employed in the *Aspect-Oriented Programming* (AOP) paradigm [8]. In AOP they are generically known as *aspect weaving*, *i.e.*, the automatic generation and insertion of code relating to non-functional requirements, such as fault-tolerance, security, etc. Aspect weaving can be performed at compile-time or at run-time, at source code level or at bytecode level.

In our work we chose to perform source code generation at compile-time. Source code generation is easier to implement and to check and it's also more portable than bytecode, because many virtual machines running on mobile devices use a different, more compact representation of bytecode that can change between implementations. The main disadvantage of weaving source code is the inability to extend third party components whose source code isn't available. Although run-time code generation is a simpler and more powerfull technique for building highly adaptable applications, it requires run-time services (such as reflection) that aren't available in most virtual machine implementations running on mobile devices.

Application source code is parsed into an *Abstract Syntax Tree* (AST). Code generation is performed by grafting new nodes to the AST and modifying or deleting existing ones. The AST is then translated into temporary source code files (pretty-printing) which are then compiled. The original source code files remain unchanged. The whole process is illustrated in Figure 1.

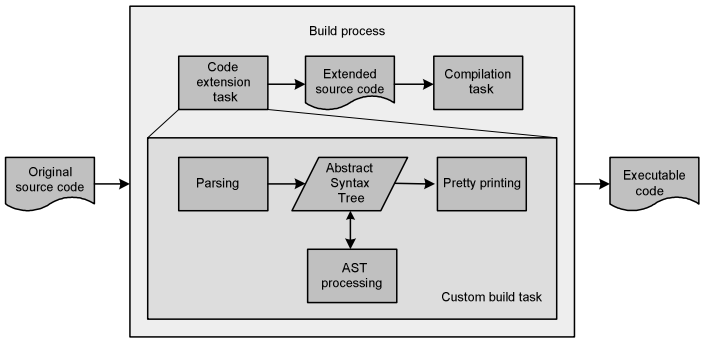

Fig. 1. Code generation flowchart.

obicomp is implemented as a custom pre-compilation build tool which can be easily added to existing build tool chains. Seamless integration is achieved by supplying the developer with project templates for developing applications on top of the OBIWAN

middleware. These templates include all the middleware libraries, the *obicomp* build tool for automatic code generation and a template project build file which references the libraries and the *obicomp* build tool.

3 Implementation

In this section we discuss the implementation details of the middleware integration framework for the .NET and Java environments.

3.1 .NET

We developed a custom build task for the MSBuild [10] building system used by the .NET Framework 2.0 [11], Visual Studio 2005 [22] and following versions. We took advantage of .NET's CodeDOM framework [4] for implementing our code extension build task. CodeDOM provides a metamodel for representing and transforming .NET application source code written in any .NET language. The namespaces `System.CodeDom` and `System.CodeDom.Compiler` define AST nodes for representing program elements and code generators (pretty-printers) for the VB.NET and C# languages. We have used an existing C# CodeDOM parser [5] and extended it according to our needs. We also used Visual Studio 2005's built-in support for developing project templates for the OBIWAN middleware.

3.2 Java

We developed a custom build task for the Ant build tool [1] using Spoon [14]. Spoon is a Java program processor fully compliant with the Java Language Specification 3 (JLS3). It provides a Java metamodel for analysing and transforming Java programs. Spoon parses Java programs into an AST that can be read and modified by Spoonlets written by the developer. A Spoonlet is a program processor that implements the *visitor* design pattern [13]. It traverses the entire AST and executes when a specified condition is met. Program transformation performed by the Spoonlet can be specified programatically or through the use of Spoon templates, written as pure Java code. Spoon supplies a custom Ant task for executing Spoonlets during the build process. Our custom build task for Ant is in fact the Spoon build task calling our code generation Spoonlets.

4 Evaluation

We have performed quantitative and qualitative evaluations of the OBIWAN middleware integration with the targeted development environments.

Regarding the quantitative evaluation we calculated the time overhead due to the code extension stage by measuring how long it takes to complete the global building process with code extension disabled and measuring again with code extension enabled. Overhead must be low relative to the global building process time in order to ensure usability and transparency to the developer. Figure 2 shows the time measurements for building projects with a varying number of replicatable types and fields. The test machine used is a Pentium 4 at 3.2 GHz with 1 GB of DDR2 RAM running Windows XP Professional with Service Pack 2. All measurements were taken by running the .NET

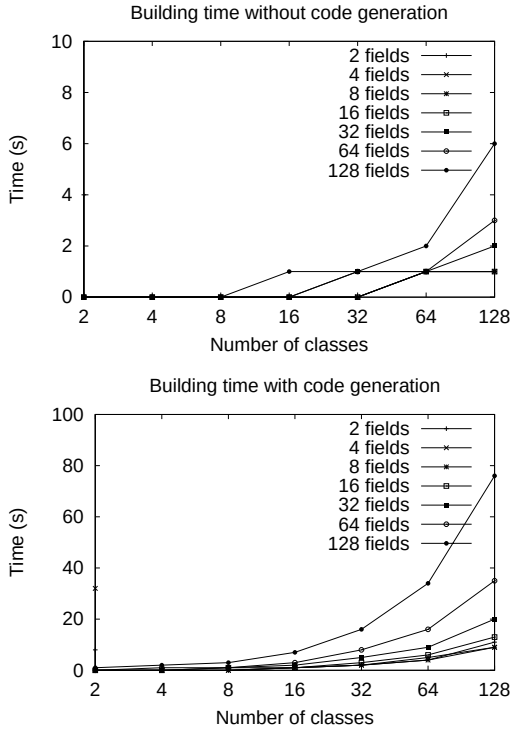

Fig. 2. Project building times with and without code generation.

implementation of the *obicomp* code generation tool with MSBuild on a command line interface.

The building time measurements show that the code generation stage introduce a very high overhead, accounting for about 90% of the global building time. The results also show that code generation time is more sensitive to the variation in the number of fields than the variation in the number of classes. Code generation time increases linearly with an increase in the number of classes and it also increases linearly with an increase in the average number of fields per class. We should note that in most real world applications very few classes have more than 16 fields. In most cases a number of fields higher than that is indicative of a need to refactor the application. In our measurements, the time for building a project with 128 classes with an average number of 16 fields per class was about 8 seconds, which is quite tolerable for the developer. We also draw attention to the fact that the *obicomp* isn't optimized regarding memory usage, and that lower building times might be achieved.

For the qualitative evaluation, we tested the usability of the OBIWAN middleware integrated in the targeted development environments by developing some sample mobile applications. The main usability issues encountered are a lack of support for partial builds, requiring the whole application to be compiled again in each rebuild, the inability to extend third-party components whose source code isn't available and exposure of the generated code to the developer while debugging. The latter can be alleviated by including source code comments in the generated code that leverage ex-

isting integrated development environments' ability to selectively hide code snippets (*e.g.*, employing `#region` directives in Visual Studio 2005).

5 Related work

WrapperAssistant [17] is a code generation tool (aspect weaver) for adding fault-tolerance support to applications targetting the .NET Framework and written in the C# language, which is very similar to our own. It uses C# *attributes* for parameterizing fault-tolerance support for chosen classes and runs at compile-time. However, it manipulates bytecode instead of source code and makes use of unmanaged metadata interfaces, making it less portable between development environments. Also, the developer must directly participate in the aspect weaving process by using the tool's graphical interface, thus lacking the same level of transparency of *obicomp*.

Afpac [18] is a middleware for dynamic adaptation that makes use of a static weaver (Taco) for generating C++ code and integrating the middleware without changes in program structure. Taco offers a very fine grain view of the application code, allowing it to target individual code statements, while *obicomp* currently only handles class declarations, method calls and type references. However, Afpac makes use of an aspect language while OBIWAN requires none, thus making Afpac less transparent to the developer than OBIWAN.

Jarcler [12] is an aspect-oriented middleware for building distributed applications using replicated objects. Just like Afpac, it relies on an aspect language (AspectJ [9]).

6 Conclusions and future work

We presented our approach for enabling transparent middleware integration. Our main goals are transparency and portability. Although our main research is in the area of distributed mobile applications, we use generic program transformation techniques that can be applied to any kind of middleware and integrate easily in existing development tool sets.

Our approach requires no language extensions, no special support from the underlying virtual machine running on the device and no modifications to the development tools. The OBIWAN middleware code generation tool fits naturally in the application building process, providing seamless integration from the developer's point of view.

We gave implementation details on prototypes developed for the Java and .NET environments. We showed quantitatively that the additional time overhead for source code extension is tolerable and reported current usability issues.

For future work we plan to address the usability issues discovered during application development with OBIWAN and optimize the code generation process. We are also considering extending OBIWAN's language support to other languages targeting the JVM and the CLR, such as Python, Ruby and VB.NET.

References

1. The Apache Ant Project. `http://ant.apache.org`.
2. Denis Caromel. Toward a method of object-oriented concurrent programming. *Commun. ACM*, 36(9):90–102, 1993.

3. Steve J. Caughey, Daniel Hagimont, and David B. Ingham. Deploying distributed objects on the internet. In *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, pages 213–237, London, UK, 1999. Springer-Verlag.
4. Using the CodeDOM. `http://msdn2.microsoft.com/en-us/library/y2k85ax6.aspx`.
5. Ivan Zderadika's C# CodeDOM Parser. `http://www.codeproject.com/csharp/codedomparser.asp`.
6. Eclipse.org home. `http://www.eclipse.org`.
7. Paulo Ferreira, Luís Veiga, and Carlos Ribeiro. Obiwan: design and implementation of a middleware platform. *Transactions on Parallel and Distributed Systems*, 14(11):1086–1099, Nov. 2003.
8. Jeff Gray and Suman Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 36–45, New York, NY, USA, 2004. ACM.
9. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.
10. MSBuild - MSDN Library. `http://msdn2.microsoft.com/en-us/library/wea2sca5.aspx`.
11. .NET Framework Developer Center. `http://msdn2.microsoft.com/en-us/netframework/default.aspx`.
12. Muga Nishizawa and Shigeru Chiba. Jarcler: Aspect-oriented middleware for distributed software in java. Technical Report C-164, Tokyo Institute of Technology, December 2002.
13. Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15, 19–21 1998.
14. Renaud Pawlak. Spoon: annotation-driven program transformation — the aop case. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, New York, NY, USA, 2005. ACM Press.
15. Davy Preuveneers, Peter Rigole, Yves Vandewoude, and Yolande Berbers. Middleware support for component-based ubiquitous and mobile computing applications. In Ackbar Joolia and Sebastien Jean, editors, *ACM/IFIP/USENIX 6th International Middleware Conference Workshop Proceedings, Demonstrations Extended abstracts*, pages 1–4, Grenoble/France, November 2005. cd-rom.
16. Nuno Santos, Luís Veiga, and Paulo Ferreira. Transaction policies for mobile networks. *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 55–64, 7-9 June 2004.
17. Wolfgang Schult and Andreas Polze. Aspect-oriented programming with c# and .net. In *ISORC '02: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 241, Washington, DC, USA, 2002. IEEE Computer Society.
18. Gaëtan Vaysse, Françoise André, and Jérémy Buisson. Using aspects for integrating a middleware for dynamic adaptation. In *AOMD '05: Proceedings of the 1st workshop on Aspect oriented middleware development*, New York, NY, USA, 2005. ACM Press.
19. Luís Veiga and Paulo Ferreira. Incremental replication for mobility support in obiwan. *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 249–256, 2002.
20. Luís Veiga and Paulo Ferreira. Poliper: policies for mobile and pervasive environments. In *ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware*, pages 238–243, New York, NY, USA, 2004. ACM Press.
21. Luís Veiga, Nuno Santos, Ricardo Lebre, and Paulo Ferreira. Loosely-coupled, mobile replication of objects with transactions. *Parallel and Distributed Systems, 2004. ICPADS 2004. Proceedings. Tenth International Conference on*, pages 675–682, 7-9 July 2004.
22. Visual Studio 2005 Developer Center. `http://msdn2.microsoft.com/en-us/vstudio/default.aspx`.