

Democratizing The Cloud

Erik Meijer
Microsoft SQL Server
Erik.Meijer@microsoft.com

ABSTRACT

Programming distributed data-intensive web and mobile applications is gratuitously hard. As the world is moving more and more towards the software as services model, we have to come up with practical solutions to build distributed systems that are approachable for normal programmers. Just like Visual Basic democratized programming Windows by removing much of the boilerplate, such as message pumps and window handles, that contributed more to the problem than to the solution, we propose a toolkit of language extensions, APIs, and tools that do the same for web programming. As a result, ordinary programmers can concentrate on the essential aspects of building distributed and mobile applications such as partitioning and flowing code and data across tiers, deployment, security, etc. without getting bogged down in low level details.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Distributed programming

General Terms

Languages, Design

Introduction

A typical modern web-based application involves various tiers. Each of these tiers usually brings along a myriad of different languages and a tier-specific data model. Moreover, as a rule, each of tier also runs on a different machine, or at best in a different process. For instance, the client-tier runs in the browser and uses JavaScript, CSS, HTML as languages and most often uses XML or JSON as the underlying data model. The data-tier runs inside the database and uses SQL as the language and relational tables as the data model. The middle-tier ties the other two tiers together and runs on the web server with Java, Perl, PHP, Ruby, C#, or Visual Basic as the programming language with their respective (object-oriented) data models. By virtue of being in the middle, it additionally needs to deal with the data models of the two outer tiers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
Copyright 2007 ACM 978-1-59593-786-5/07/0010 ...\$5.00.

Our goal is to develop distributed applications by successive program transformations (i.e. refactorings), starting from a "specification" of the problem as a client-tier application. We want to delay decisions about partitioning of code and data as long as possible. Once the time has come to make a decision, we use a combination of refactoring and declarative annotations to specify the high-level architectural concerns, and rely on compiler and tool support to take care of all irrelevant details and the generation of boilerplate code. By stretching and adopting the programming model and languages from the client-tier to cover both the middle- and data-tiers, and making it easy to move computation and data between tiers, we eradicate the accidental complexity of building distributed applications

A prerequisite to make this plan work, is to have (a) the same computational model available at each tier, that (b) allows queries across the three data models within that single computational model.

Cross-tier Computational Model

We pick the .NET Common Language Runtime (CLR) as our universal computation model. We prefer, of course, to use the already available CLR implementation on each respective tier: SQLCLR on the data-tier; regular CLR on the middle-tier; and Silverlight for Web-clients, or the regular CLR for desktop-clients.

When no CLR is on hand, we use the materials already available in the room. On the data-tier we compile MSIL to SQL. This is the approach currently taken by LINQ-to-SQL and LINQ-to-Entities. On the client-tier we compile MSIL to JavaScript or Flash. This is the approach taken by Volta. The upshot is that we uniformly provide (the illusion of) the .NET platform on each tier, in effect stretching it to cover the Cloud. Application programmers only need to care that they can run MSIL everywhere, not about how this is technically accomplished under the hood.

Cross-model Queries

Each of the big three data models is deeply rooted in their originating tier for historical and technical reasons:

- Object graphs rely on reference identity with inside-out pointers to implement associations. Unreachable objects are swept up automatically by the garbage collector. Objects offer numerous abstraction mechanisms such as in-

terfaces, abstract base classes, inheritance, and virtual methods that facilitate reuse of behavior.

- Persistent data uses the notion of primary keys for identity, outside-in primary-foreign key relationships for associations, and cascade delete for explicit garbage collection. Updates to the database can be grouped into atomic transactions.
- Mixed content and significant whitespace in XML is an artifact that can be traced back to its roots in the document world, and has no correspondence, or use, in either the relational or object-oriented models. In fact, it conflicts with both. But this document-centricity makes XML an obvious choice as a wire-serialization format.

It makes no sense to pick one of these data models as the favorite, or to unify and extend them into a single über-model.

To solve the data model impedance mismatch, we leverage the fact that each data model can implement a common interface of standard query operators (in mathematical terms, each data model can be viewed as a monad) and then define a single query comprehension syntax whose semantics is defined in terms of those base operators. This is the essence of LINQ as embodied in C# 3.0 and Visual Basic 9.

Tier Splitting

Once we have a common mechanism for querying across data models, and have a unified computational model on each tier, the next step is to facilitate partitioning a program that conceptually or logically runs on a single tier (we like to start on the client) across different physical tiers.

Traditionally writing a multi-tier application is a bottom up process that requires a tremendous amount of boilerplate code and many manual steps. Developers have to decide up-front how their application is distributed across tiers. Before they write a single line of code, they first need to define interfaces and contracts using XML configuration files that are often larger than the code for the service they are exposing. This fragmentation makes the design very brittle, and impossible to refactor or rearchitect along the way.

In the spirit of lean programming, we want to allow programmers to make irreversible decisions about the architecture of their applications, in particular tier partitioning, at the last possible responsible moment. To support this programming style, we offer a range of options for tier-splitting including refactoring support in the IDE as well as declarative support via custom attribute annotations.

To see how this works in practice let's define a super simple "dictionary suggest" application. The Suggest method in the class Dictionary returns the topmost n descriptions that match a given word p. The Suggest method is implemented by a straightforward LINQ query over a collection entries of the word descriptions contained in this dictionary:

```
IEnumerable<Description> Suggest(string p, int n)
{
    return (from e in this.entries
            where p.Length > 0 && p.Matches(e)
            select e).Take(n);
}
```

At each keystroke, we look up the completions of the word in the dictionary using the value of the the textbox i, in order to populate a listbox o with the properly formatted first 10 results:

```
var d = new Dictionary();
var i = new TextBox();
var o = new ListBox();
... set up UI ...
i.KeyUp += delegate {
    o.Items = Format(d.Completions(i.Text, 10));
};
```

We want to test and tweak this application purely on the client-tier until we are satisfied with its functional behaviour. To run the Dictionary class as a service on the middle-tier, we only need to add a custom attribute [RunAt(Server="...")] to the class declaration that specifies that the dictionary should be exposed as a web-service on the indicated server.

```
[RunAt(Server="...")]
class Dictionary {...}
```

Guided by this additional meta-data, the compiler performs an MSIL \mapsto MSIL rewrite that replaces the client-side Dictionary class with a proxy that calls into a server-side proxy to the original Dictionary implementation. Client and server communicate via a stateful RPC-style protocol using JSON, XML, or S-expressions as the serialization format. We allow the runtime infrastructure to pick the serialization format and the optimal channel implementation based on additional quality of service arguments specified in the custom attribute.

There is no such thing as a free lunch, and the first thing we need to worry about when making remote calls across the network is latency and availability. Instead of making a synchronous, blocking, call, we should make asynchronous remote calls across the network. This is as simple as decorating a new overload of the Suggest method with the [Async] custom attribute.

```
[Async]
extern void Suggest(string p, int n,
                    Action<IEnumerable<Description>> cont);
```

As usual, continuation-passing style turn the call to Completions inside-out

```
i.KeyUp += delegate {
    d.Completions(i.Text, 10,
                 s => {o.Items = Format(s); });
};
```

Tier-splitting is not just applicable for partitioning computation between client and server. In practice, we would continue partitioning our application until all computation happens at the proper tier. For instance, we might want to move the dictionary query into the data-tier, and perhaps split the client-tier to run the UI in Silverlight but leave the Dictionary class in the browser.

Conclusions

By making the compiler and tools intrinsically aware of the distribution aspects of applications, we can ensure a large number of correctness, safety, and security properties about the resulting code by construction. This would be hard to guarantee if the program is partitioned manually. In the long run, we aim at supporting the standard repertoire of enterprise application integration patterns out of the box.