Replicated Distributed Shared Memory For The .NET Framework

Thomas Seidmann
Department of Computer Science and Engineering
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology
seidmann@dcs.elf.stuba.sk *

Abstract

This paper introduces a software-only object based Distributed Shared Memory (DSM) implementation designed as an extension to the Microsoft .NET framework. This implementation is facilitated by a previously described memory coherence protocol, which uses group communication by multicasting on IP networks. The described DSM implementation allows the construction of distributed applications with a simple programming model.

Keywords: Object-based distributed shared memory, object replication, causal consistency, IP multicasting, .NET Framework, .NET Remoting.

1 Introduction

The .NET Framework represents a platform for building applications by providing an infrastructure commonly described as 'middleware'. It consists of several layers, the lower-most being a virtual machine called Common Language Runtime (CLR) with a Just-In-Time (JIT) compiler for Intermediate Language (IL) code. Applications running on top of the CLR, called Managed Applications, use the .NET Framework Class Library (FCL).This approach is not unlike the Java Virtual Machine (JVM) one, although there are several differences, the main one being the availability of the framework exclusively on Windows platforms. However a subset of the CLR, the Shared Source Common Languange Infrastructure (SSCLI), is available for the FreeBSD, MacOS X and Windows operating systems, which can be (in fact has been) ported to other systems as well. Another key difference is also the availability of multiple programming languages (compilers being delivered by Microsoft and third parties), all sharing a Common Type System (CTS) and the same set of class libraries. Microsoft delivers among others a compiler for C#, a Java-inspired C-spin-off.

1st Int.Workshop on C# and .NET Technologies on Algorithms, Computer Graphics, Visualization, Computer Vision and Distributed Computing

February 6−8, 2003, Plzen, Czech Republic.

ISBN 80−903100−3−6

For the purpose of building distributed applications the CLR can be viewed as a component activation platform, facilitating the deployment of software components in both local and remote locations. The class library provides several means of communication, ranging from lower-level socket-layer programming to high-level .NET Remoting and Web Services, the latter two forming a real-life middleware for distributed objects. From the communication point of view are Web Services a subset of .NET Remoting, limiting to RPC-style of invocation via an XML-centric protocol Simple Object Activation Protocol (SOAP). .NET Remoting provides more universal communication means than Web Services; however, most of the documented usage of it still goes the direction of RPC-style client/server relationship.

The .NET Framework thus leaves a developer, who wants to create distributed applications, solely with remote access to objects shared between application instances. Besides the data (attributes) contained in the object instance also its functionality (methods) is accessed remotely. The scenario, where several .NET peer applications collaborate on a set of shared objects, which would be replicated among the application instances in order to provide local access to both data and functionality is not directly supported.

Our aim was to develop a distributed shared objects runtime based on the .NET Framework that provide

*Postal address: Baumacher 12, 6244 Nebikon, Switzerland, Tel. +41 62 7563203

distributed applications with replicated shared objects, while still utilizing existing facilities of .NET Remoting. Contrary to the notion of remote objects being **provided by** a server, we wanted to adapt .NET Remoting to handle objects **shared among** peer .NET applications. Instead of remote method invocation the focus is on replication of shared objects and keeping the replica coherent according to an agreed consistency model.

2 Architectural Elements of .NET Remoting

The basic building blocks of .NET Remoting are runtime serialization and channel services.

Runtime serialization is used for transporting object instances across communication channels; it is customizable by the use of Serialization Formatters, which encode and decode messages between .NET applications. Two of them are present in the .NET Framework: the Binary and the SOAP Serialization Formatters, the latter utilizing XML.

Channel services provide the actual transport mechanism (channels) for messages between .NET applications. Serialization Formatters can be plugged into channels. The .NET Framework supplies the HTTP and TCP channels. By default the HTTP channel uses SOAP, whereas the TCP channel uses the Binary Serialization Formatter.

The programming model of .NET Remoting employs a singleton object named Activator for activating remote objects. The terminology of .NET Remoting heavily leans toward the client/server paradigm by classifying remote objects as server-activated and client-activated. In both cases the focus is on method invocation of remote objects, facilitated through two kinds of proxy classes provided by the .NET Framework, one of which can be extended (for details refer to Section 4). The configuration of .NET Remoting applications can be done internally in the program code or externally in XML-formatted files.

3 Consistency Model and Coherence Protocol

We have previously developed a coherence protocol, which provides causally consistent Distributed Shared Memory (DSM) and utilizes group communication in the form of IP multicasting [1] [1].

The protocol is basically a Multiple Reader Multiple Writer (MRMW) protocol with write-update using multicast transfer of *diffs* between the involved processes. The causality relationship between shared memory operations is achieved by means of **vector logical clocks** [2], which represent the basic building block in the algorithm. Every process maintains for every shared object a vector logical clock value - timestamp - consisting of:

- The process's own logical (Lamport, monotonic) clock value of the last write operation upon the object.
- Other known processes' logical timestamps of write operations upon the object.

This vector logical timestamp is transmitted within every message that concerns this particular object. The vector logical timestamp values are represented as an associative array consisting of pairs $(PID, value)$, where PID denotes the process' global ID composed of the node's global ID (in the case its IP address) concatenated with the node local PID.

Every shared object in the DSM must be uniquely identified. The current implementation uses a Globally Unique Identifier (GUID) for this purpose, which must be agreed upon by all participating processes.

This coherence protocol was chosen as the foundation of the envisioned distributed shared object framework for .NET. Due to the nature of the protocol, namely no special provisions by the communication subnetwork except of the availability of IPv4 or IPv6 (unreliable) multicast, deployment within the worldwide Internet is technically possible.

4 DSM Implementation for the .NET Framework

4.1 Implementation Overview

To implement the coherence protocol described in Section 3 the availability of three mechanisms is required:

1. Mechanism for obtaining the state of an object suitable for transporting either in its entirety or as a *diff* against a previous state.
2. Registration of every change of a shared object by a local process in a database.
3. Listening to changes of an object made by a remote process and to object state queries.

The first mechanism is delivered by runtime serialization in .NET. Due to accessibility of type information via reflection in the .NET Framework serializability of objects is achieved easily by attaching the `[serializable]`[2] attribute to the class definition.

[1] The paper is available at http://www.cdot.ch/thomas/

[2] Syntax in C#

The Binary Serialization Formatter delivers content suitable both for transport and for calculation of object state diffs.

The second mechanism is achieved by *Interception*, a built-in feature of the .NET Framework, which provides means to intercept every call to an object's method and perform some additional action. Interception is implemented in the form of two proxy classes: *TransparentProxy* and *RealProxy*, the latter of them being extendible by derivation. The DSM implementation does this by providing the class *DSMProxy*. On entry to every method call on the shared object the object diff storage is checked for updates from other nodes. On exit from the method call the diff is calculated and in case it is non-zero, an entry consisting of the diff and the vector logical clock value is added to the storage of object diffs and multicast to all other nodes.

The third mechanism is provided by a coherence thread which accesses the object diff storage and the shared object itself via its DSMProxy instance to perform state changes according to remote write update messages or provide its state to other processes, respectively. The coherence thread is also responsible for sending out object updates from the object diff storage and requesting other nodes for sending of object diffs upon the request of DSMProxy (for example after the creation of a new shared object at the node).

Figure 1 illustrates the main players in the DSM implementation on one particular node. When an application instance creates a shared object (by using the *new* operator), the .NET Runtime uses Reflection to get the object's type information. After it detects that the object must be created in another AppDomain, it creates a TransparentProxy instance in the calling AppDomain instead. TransparentProxy has exactly the same interface as the shared object, but the implementation of all methods and properties result in calling the *Invoke* method of the specified RealProxy descendant, DSMProxy. A request for object diffs is sent out to other nodes via the coherence thread.

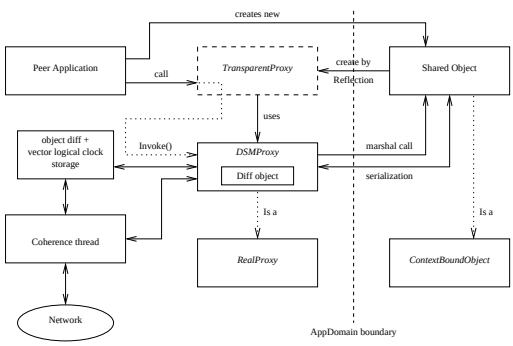

Figure 1: DSM implementation overview

4.2 The Multicast Channel Service

Since both built-in channels of the .NET Framework are TCP-based, none of them is multicast enabled. To implement the coherence protocol for shared objects a multicast-able channel is needed. A class becomes a .NET Remoting channel by implementing the *IChannel* interface. The channel implementation is contained in the class *UdpChannel*, and as the name indicates, it uses UDP as its transport protocol. This channel is usable not only for the presented scenario, but also anywhere reliable and order-guaranteed communication (which provides TCP) is not needed or is implemented elsewhere (like in the application layer).

4.3 Programming Model for Distributed Shared Objects in .NET Applications

For a class to be usable as shared object type in the presented DSM, the following three requirements must be met:

1. The class must be descendant of *ContextBoundObject.*
2. The class must be serializable.
3. The class must be annotated with the *ProxyAttribute* attribute specifying *DSMProxy* as the attribute value.

An example written in C# would be:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Proxies;
using McastDSM;

[Proxy(DSMProxy)]
[serializable]
public class MySharedObject :
  ContextBoundObject, ISharedObject
{
  private Guid myGuid = null;
  public Guid GuidProp
  {
    get {return myGuid;}
    set {myGuid = value;}
  }
  public int myMethod(string)
  {
    //... do something with string
    return 0;
  }
}
```

The following code segment is an example of a shared object instantiation:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using McastDSM;
using McastDSM.Channels;

public class App
{
  public static int Main(string[] args)
  {
```

```
    ChannelServices.RegisterChannel (
      new UdpChannel());

    // Create an instance of a
    // MySharedObject class
    MySharedObject myObj = new MySharedObject;
    myObj.GuidProp =
"478280B9-874E-4795-B3C7-05CFDD96CD2C";
    myObj.myMethod("Hello World");
    return 0;
  }
}
```

Additionally, on assembly level, the value of the DSMPort has to be specified, containing the multicast address and port to be used by the application:

```
[assembly: DSMPort("udp://[ff05::1234:5678]:8888")]
```

In this example a IPv6 site-local, transient multicast address with group ID 12345678 (hexadecimal) and port 8888 are used.

5 Related Work

The use of group communication is not a new idea, there are already DSM implementations which make use of multicast.

Speight and Bennett reported in [3] about the use of multicasting in the Brazos project. DSM in Brazos is concentrated on scope consistency, that means a memory consistency model using synchronization variables; our approach is to provide causal consistent DSM. Moreover Brazos relies implicitly on reliable multicast, whereas we do not.

The Orca distributed system [4] uses totally ordered group communication for a MRMW write-update coherence protocol. The protocol employs a centralized component - the sequencer - to achieve totally ordered multicast on top of the potentially unreliable IP multicast. In our design this centralized component is avoided.

The OpenMP implementation based on the TreadMarks system has been extended to use multicasting by Honghui Lu [5] in some specific situations, namely during the access to shared data in the (replicated) sequential part of a parallel program. In our system multicasting is the main means of communication between processes.

6 Conclusions and Future Work

The implementation of the presented runtime has been first tested on local area networks with two well-known scientific applications, Barnes Hut (from the SPLASH suite) and FFT-3D, later with two decentralized financial domain applications. The most important parts of the testing process were, besides the functionality, the amount of network traffic and recovery from lost multicast messages. Loss of messages was simulated with the help of FreeBSD-based routers with traffic shaping abilities. Next the testing has been successfully performed between two 6bone (IPv6 testbed on the Internet) sites, which is a more realistic environment for envisioned large-scale distributed applications.

The presented runtime has successfully proven its functionality. All components of it are written in the C# language and consist only of managed (that means IL) code, since all needed base functionality (like socket-level routines etc.) is available in the .NET class libraries and thus no transition to native code or direct access to the operating system is needed. It provides an agreeable programming model and a consistency model suitable for most studied applications at reasonable performance with a small communication overhead thanks to group communication.

The use of other serialization formatters is being considered, namely the Soap Serialization Formatter. It might provide better means of *diff* calculation. In conjunction with the XMill, an XML aware compression technique, the size of messages might be reduced even compared to compressed binary messages [6].

References

[1] Thomas Seidmann. Multicast-based runtime system for highly efficient causally consistent software-only DSM. In *Lecture Notes in Computer Science 1586, IPPS/SDSP'99 Workshops*, April 1999.

[2] Randy Chow and Theodore Johnson. *Distributed Operating Systems & Algorithms*. Addison Wesley Longman, Inc., 1997.

[3] W. E. Speight and J. K. Bennett. Using multicast and multithreading to reduce communication in software dsm systems. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, pages 312–323, February 1998.

[4] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Ruehl, and M. Frans Kaashoek. Performance evaluation of the orca shared object system. *ACM Trans. on Computer Systems*, February 1998.

[5] Honghui Lu. *OpenMP on Networks of Workstations*. PhD thesis, Rice University, 2001.

[6] A comparison of alternative encoding mechanisms for web services. http://dblab.usc.edu/microsoft/.