©2006 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE. This is the author's version. The original publication is available at: http://dx.doi.org/10.1109/DEXA.2006.146

Using Replication to Build Highly Available .NET Applications

Johannes Osrael[1], Lorenz Froihofer[1], Georg Stoifl[2], Lucas Weigl[2], Klemen Zagar[3], Igor Habjan[3], and Karl M. Goeschka[1]

[1]Vienna University of Technology, Argentinierstrasse 8/184-1, 1040 Vienna, Austria, {osrael|froihofer|karl.goeschka}@tuwien.ac.at

[2]University of Applied Sciences Technikum Wien, Höchstädtplatz 5, 1200 Vienna, Austria, georg@stoifl.com, lucas.weigl@gmx.at

[3]Cosylab, Teslova ulica 30, SI-1000 Ljubljana, Slovenia, {klemen.zagar|igor.habjan}@cosylab.com

Abstract

Replication is a well-known technique to achieve fault-tolerance in distributed systems, thereby enhancing availability. However, so far, not much attention has been paid to object replication using Microsoft's .NET technologies. In this paper, we present the lessons we have learned during design and implementation of a .NET based replication framework that allows building dependable, distributed .NET applications. Our framework does not only support traditional replication protocols like primary-backup replication or voting but also a new protocol for explicit balancing between data integrity and availability. Based on our experiences, we recommend to use a state-of-the-art group communication toolkit (e.g., Spread) and .NET Remoting as basis for object replication in a .NET environment.

1. Introduction and contribution

Replication, the process of maintaining different copies of an entity (data item, object), is used to enhance performance and availability of distributed systems. A plethora of replication techniques has been proposed and thoroughly investigated since the 1970's for many different domains such as databases, tightly-coupled distributed systems (objects, middleware components), and more recently large-scale, wide-area systems such as peer-to-peer and Grid environments. However, applying the well-understood principles of replication to new paradigms (such as service-oriented computing) or new technologies is nevertheless a challenge. For instance, Microsoft's .NET platform is a relatively new technology where object-level replication is still in its infancy. In this paper, we contribute by closing the gap between research and engineering and present the lessons we have learned when we designed and implemented a .NET-based framework that supports different replication protocols such as primary-backup replication [1], voting [2], and the newly designed primary-per-partition protocol [3].

Our replication framework has been developed within the DeDiSys (Dependable Distributed Systems) research project. Among others, DeDiSys aims at enhancing availability by temporarily relaxing data integrity. To control this trade-off, explicit run-time management of data integrity constraints and consistency threats is required in combination with a new kind of replication protocols (e.g., the primary-per-partition protocol). Target applications that benefit from this approach range from control engineering (e.g., the Advanced Control System [4]) to air traffic control (e.g., the Distributed Telecommunication Management System [5, 6]).

In this paper, we do not focus on the specifics of the DeDiSys research project but concentrate on the replication part of the middleware system, that can be used for any kind of general-purpose distributed system.

2. System model and architecture

System model We focus on tightly-coupled, data-centric, object-oriented distributed systems [7] with up to about 30 server nodes and an arbitrary number of client nodes. Server nodes host objects which are replicated to other server nodes in order to achieve fault-tolerance. We consider both node and link failures (partitioning), i.e., the crash failure [8] model is assumed for nodes and links may fail by losing but not duplicating or corrupting messages.

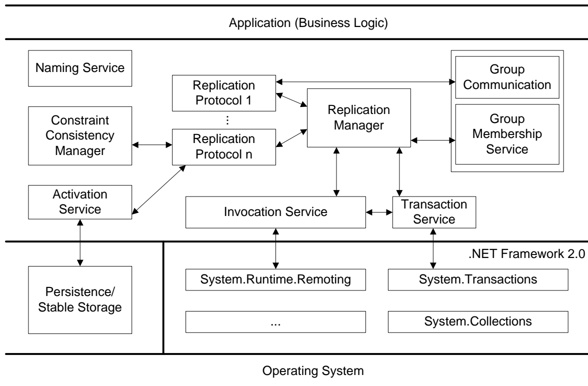

Figure 1. DeDiSys .NET system architecture

System architecture We have presented the platform-independent DeDiSys system architecture in [3]. Figure 1 shows the mapping of the architecture to a .NET environment. Our framework is based on .NET 2.0 using Windows as underlying operating system. The C# programming language has been chosen for implementation.

The core components of the system architecture are the replication manager and its associated replication protocols and the group membership and group communication services. Further components are the invocation service, naming service, activation service, transaction service, and the DeDiSys-specific constraint consistency manager [9].

3. Lessons learned

In this section we shortly describe each component and present the most important lessons learned during design and implementation. Finally we evaluate two options for point-to-point communication.

3.1. Group communication and group membership service

A group membership service is used to keep track of which nodes are operational, taking into account voluntary group changes (join or leave) as well as node and link failures. Group communication provides reliable multicast to groups with configurable delivery and ordering guarantees.

Group membership and group communication services can be treated as separate components which interact with each other. However, in practice, both components are usually integrated in one toolkit which is referred as view-oriented group communication system [10].

From the variety of available state-of-the-art toolkits like Appia [11], Spread [12], Ensemble [13], JGroups [14], and GCT [15], to our knowledge three of them provide a C# API: Spread, Ensemble, and GCT.

Any of the three toolkits could be used in a .NET replication framework from a technical point of view. However, Spread is our first choice since a) the Spread community seems to be more active than the other communities, b) commercial support for Spread is available and c) Spread is also used in our J2EE and CORBA implementations of the framework which eases cross-technology comparison.

3.2. Replication manager and protocols

Replication is the primary means to achieve fault-tolerance in our middleware infrastructure. The replication core system is divided into two sub-parts: The replication manager and the replication protocol.

The replication manager keeps track of object replicas in the system. Thus, it maintains a mapping between the logical object and its replicas. Furthermore, depending on the replication model it supports, the roles of the replicas need to be stored as well, e.g., primary replicas vs. secondary replicas in a primary-backup scheme. The replication manager has to be built from scratch since no previous work on a .NET replication manager for partitionable environments exists to our knowledge.

Our replication architecture supports both traditional replication protocols (e.g., primary-backup replication [1], voting [2]) and novel protocols for balancing data integrity with availability (e.g., the primary-per-partition protocol (P4) [3]).

The main lesson we have learned during implementation of the replication manager and the protocols is that .NET does neither offer specific features that would ease implementation nor does it aggravate it. Custom implementation is required with respect to the invocation service. Comparing our .NET implementation of the manager and the protocols with a previous implementation in a Java framework [16] yields to the conclusion, that the existing Java implementation could have been recoded rather easily in C#. With .NET Remoting providing the invocation logic it is best to have a replication manager and protocols that are capable of handling `IMessage` as invocation context.

3.3. Invocation service

The role of the invocation service is to intercept all method invocations upon a logical object and to ensure that the invocation is properly delivered to the respective replicas of that object. Also, the invocation service must properly convey the transaction context and other contextual information (e.g., authentication principal) from the client's to the replica's process.

The invocation is the only interaction between the user

and the middleware; thus, further involvement of the middleware is made transparent after the call is issued.

There are several ways to implement the invocation service in Microsoft .NET:

- **Custom invocation API.** This would typically be an implementation of a command design pattern [17]. Though this approach is most flexible and provides an elegant solution to issues such as atomicity and isolation, it might be cumbersome to use for the application developers, as it involves an implementation of a *command object* for every method.
- **Aspect-Oriented Programming (AOP) [18].** Interceptors are injected in the call chain using cross-cuts at method invocations of replicated objects. Several AOP toolkits for .NET are available ([19][20][21]). The drawback of this approach is that application developers have not yet adopted the AOP methodology and that the toolkits have not yet reached production-level maturity.
- **.NET Remoting interceptors.** The .NET framework provides mechanisms for injecting interceptors at the client and server side of the invocation chain. The advantage of this approach is that the infrastructure is readily provided by the .NET framework itself. A disadvantage is that all the replicated objects must extend the `MarshalByRefObject` class, thereby limiting the designer's options for inheritance.

Our effort so far has concentrated on the .NET Remoting interceptors, which has also been used for replication purposes in the past [22]. Since at a later time we might also investigate other approaches, we have abstracted the interceptors with an API introducing `before` and `after` operation delegates that hide the interception details (.NET Remoting, AOP, synchronous, asynchronous, ...) from the interceptor implementation. Thus, the implementation of the invocation service can be replaced without affecting the other components. For instance, it would be easy to weave an AOP *advice* code snippet into the begin of method invocations.

3.4. Transaction service

We have not yet made use of transactions, therefore this subsection only presents findings of our technology investigation and prototyping.

Before .NET 2.0, transactions were available through the `System.EnterpriseServices` namespace. `EnterpriseServices` is essentially an API that exposes the COM+ infrastructure to code written for .NET. Thus, in order to make use of transactions, the .NET code had to be deployed as a COM+ component. Unfortunately, COM+ applications are not so easy to deploy (e.g., using the `xcopy` deployment commonly found in .NET). Also, they impose a significant overhead, as .NET applications must be exposed to the COM+ container via a COM/.NET interop bridge.

With .NET 2.0, the `System.Transactions` library was introduced that provides a native .NET implementation of transactions. Transactions can be either lightweight (inside an application domain) or they might employ the Microsoft Distributed Transaction Coordinator (MSDTC).

Transactional resource managers are relatively easy to implement by extending the `IEnlistmentNotification` or the `ISinglePhaseNotification` interface. The resource manager is then enlisted in a current transaction (`Transaction.Current`), whereas the transaction manager uses its `Prepare`, `Rollback` and `Commit` methods to coordinate the two-phase (or single-phase) commit protocol.

3.5. Naming service

The naming service maps human-readable object names to logical object IDs. We have investigated two ways of achieving this mapping:

- **Active Directory.** Active Directory is Microsoft's implementation of the *Lightweight Directory Access Protocol* (LDAP) that allows access to a distributed naming hierarchy. Active Directory is typically used to store administrative information about entities such as computers, users and printers.
- **Custom implementation.** If only functional requirements (name-to-object binding and resolution) need to be implemented, and non-functional requirements (distribution, replication, ...) are provided using other means, a naming service is not difficult to implement – essentially, it is a simple `IDictionary` implementation.

Because Active Directory incurs a lot of overhead due to the non-functional requirements it addresses, and is non-trivial to set-up and administer, we have decided to implement the naming service using a dictionary implementation provided with .NET. This is sufficient since we use our framework's replication logic to replicate the naming service.

3.6. Activation service

The activation service is responsible for creating instances of objects.

With `Enterprise Services`, the COM+ framework provides the just-in-time activation (JITA) service,

where a method on an object is called when it is activated or deactivated, even though the object is not actually taken out of memory. Through this call, application developers can handle the activation and deactivation logic. However, this approach is not directly applicable to our purposes, as it is useful for improved performance and resource utilization of service-like objects, whereas it is inapplicable to data objects.

Thus, we have considered the following alternative approaches to activation:

- **Manual activation.** The host of the object's replica instantiates an instance of the object using the `new` keyword. Then, it ensures that the instance is available via .NET Remoting through the use of the `RemotingServices.Marshal` method. Also, the host process must notify the replication manager of the new replica, and associate it with a logical object.

- **Interception of `IConstructionCallMessage`.** For client-activated objects, the invocation chain interceptors can intercept an `IConstructionCallMessage`, create the instance of an object (or fetch it from a cache), and return the result.

So far, we have been working on the first approach, mainly due to its simplicity. The second approach will be pursued later on to relieve the application developers from having to create replicas by themselves and register them with the replication manager.

3.7. Point-to-point communication

Invocation logic and parts of the replication logic (e.g., the reconciliation phase in the primary-per-partition protocol) require point-to-point communication. To evaluate whether we should use the group communication toolkit Spread even for point-to-point messages, we have compared the execution time of a .NET Remoting call with a message transfer using Spread. The .NET Remoting call was issued as *call by value*, i.e., the remote object was transferred from one machine to the other. Since a .NET Remoting call requires two message transfers (invocation and return), equivalent behavior has been achieved by acknowledging the receipt of each Spread message.

The measurements were conducted on a 100MBit switched network on two machines (3GHz, 1024MB RAM, Windows XP SP2). Thousand test iterations have been performed with different message sizes. The average transfer times, given in ms in the below figure, show that - independent of the message size - .NET Remoting using the binary formatter over TCP is fastest but Spread is still faster than .NET Remoting using the SOAP formatter over HTTP. Hence, we have used the first variant in our replication middleware for point-to-point calls.

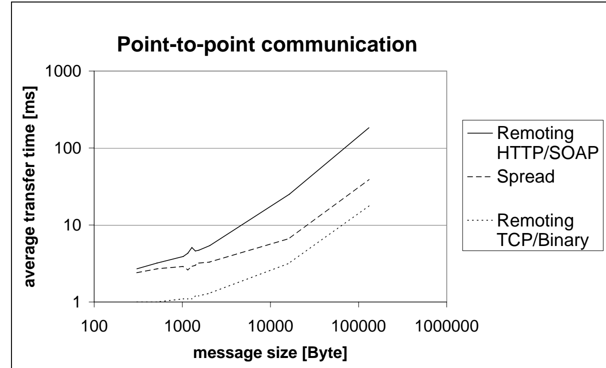

Figure 2. Spread vs. .NET Remoting

4. Related work

Replication has been used in several middleware systems for distributed objects. However, most of the research projects have been focused on CORBA (Common Object Request Broker Architecture) [23] or J2EE middleware [24].

Seidmann [22] implemented object replication in .NET environments; however, in the context of a distributed shared memory, while we focus on a flexible replication middleware for partitioned environments.

Reiser et al. [25] developed a framework that offers three forms of replication: First, a load-balancing algorithm for read requests (based on a round-robin or random selection strategy) has been implemented. Second, active replication using .NET Remoting has been implemented. However, this approach is only suitable for stateless objects or objects that are not accessed by more than one client simultaneously since total order of the invocations is not guaranteed. The third variant is comparable to our approach since it implements active replication using the GCT [15] group communication toolkit. However, in contrast to our framework, it is not targeted to partitioned environments. Furthermore, beyond traditional replication, our framework allows enhancing availability even more by trading data integrity for availability.

5. Conclusions and future work

We presented the lessons we have learned during design and implementation of a middleware architecture that offers object-level replication in .NET environments. First of all, we recommend to use an existing group communication

toolkit for reliable group communication in .NET environments. Among the variety of toolkits we have considered, Spread, Esemble, and GCT offer C# APIs. Furthermore, we favor the mechanisms provided by .NET Remoting for interception of method invocations (compared with a custom invocation API or AOP toolkits). However, a custom implementation of a naming service is more appropriate than to use Active Directory. Finally, we compared .NET Remoting with Spread for point-to-point communication. .NET Remoting using a binary formatter over TCP is faster than Spread - independent of the message size.

Future work includes support of transactions and implementation of further replication protocols.

6. Acknowledgments

This work has been partially funded by the European Community under the FP6 IST project DeDiSys (Dependable Distributed Systems, contract number 004152, http://www.dedisys.org).

We thank Almir Kazazic for many in-depth discussions about our solution.

References

[1] N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. The primary-backup approach. In S.J. Mullender, editor, *Distributed systems*, chapter 8. ACM Press, Addison-Wesley, 2nd edition.

[2] D.K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proc. of the seventh ACM symposium on Operating systems principles*, pages 150–162, 1979.

[3] J. Osrael, L. Froihofer, K.M. Goeschka, S. Beyer, P. Galdámez, , and F. Muñoz. A system architecture for enhanced availability of tightly coupled distributed systems. In *Proc. of 1st Int. Conf. on Availability, Reliability, and Security*. IEEE, 2006.

[4] K. Zagar. Fault tolerance scenarios in control engineering. In P. Cunningham and M. Cunningham, editors, *Innovation and the Knowledge Economy - Issues, Applications, Case Studies*, volume 2, pages 1389–1395. IOS Press, 2005.

[5] R. Smeikal and K.M. Goeschka. Fault-tolerance in a distributed management system: a case study. In *ICSE '03: Proc. of the 25th Int'l Conf. on Software Engineering*, pages 478–483. IEEE CS, 2003.

[6] K.M. Goeschka and R. Smeikal. Using replication for increased availability of a distributed telecommunication management system. *e&i*, 121(5):187–193, 2004.

[7] J. Osrael, L. Froihofer, H. Kuenig, and K.M. Goeschka. Scenarios for increasing availability by relaxing data integrity. In P. Cunningham and M. Cunningham, editors, *Innovation and the Knowledge Economy - Issues, Applications, Case Studies*, volume 2, pages 1396–1403. IOS Press, 2005.

[8] F. Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2), 1991.

[9] L. Froihofer, J. Osrael, and K.M. Goeschka. Trading integrity for availability by means of explicit runtime constraints. In *Proc. 30th Int. Computer Software and Applications Conference (COMPSAC'06)*. IEEE CS, 2006.

[10] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[11] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *Proc. of the 21st Int. Conf. on Distributed Computing Systems*, page 707. IEEE CS, 2001.

[12] J. Stanton Y. Amir, C. Danilov. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proc. of The Int. Conf. on Dependable Systems and Networks*, pages 327–336. IEEE, 2000.

[13] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[14] Jgroups project. http://www.jgroups.org.

[15] Gct project. http://www.smartlab.cis.strath.ac.uk/Projects/GCTProject/GCTProject.html.

[16] S. Beyer, A. Sánchez, P. Galdámez, and F. Muñoz-Escoi. Dedisys lite: An environment for evaluating replication protocols in partitionable distributed object systems. In *Proc. of the 1st Int. Conf. on Availability, Reliability and Security*. IEEE CS, 2006.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[19] Aspect#. http://www.castleproject.org/index.php/ AspectSharp.

[20] Runtime assembly instrumentation library (rail). http://rail.dei.uc.pt/.

[21] Aspectdng. http://www.dotnetguru.biz/aspectdng/.

[22] T. Seidmann. Distributed shared memory using the .net framework. In *Proc. of the 3rd IEEE/ACM Int. Symp. on Cluster Computing and the Grid*, pages 457–462, 2003.

[23] Object Management Group (OMG). Common Object Request Broker Architecture: Core Specification, v3.0.3, 2004.

[24] Ö. Babaoglu, A. Bartoli, V. Maverick, S. Patarin, J. Vuckovic, and H. Wu. A framework for prototyping j2ee replication algorithms. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, pages 1413–1426. Springer Verlag, 2004.

[25] H.P. Reiser, M.J. Danel, and F.J. Hauck. A flexible replication framework for scalable and reliable .net services. In *Proc. of the IADIS Int. Conf. on Applied Computing*, volume 1, pages 161–169, 2005.