A FLEXIBLE REPLICATION FRAMEWORK FOR SCALABLE AND RELIABLE .NET SERVICES

Hans P. Reiser
Dept. of Distributed Systems, University of Ulm, Germany
reiser@informatik.uni-ulm.de

Michael J. Danel
University of Erlangen-Nürnberg, Germany
michael.j.danel@informatik.stud.uni-erlangen.de

Franz J. Hauck
Dept. of Distributed Systems, University of Ulm, Germany
hauck@informatik.uni-ulm.de

ABSTRACT

We present a flexible framework for the development of reliable and scalable distributed services based on the Microsoft .NET framework. Scalability is obtained by load balancing among dynamically managed service replicas. Fault tolerance is achieved by redundancy among replicas and an active replication strategy to ensure replica consistency. The framework architecture features full replication transparency for client applications. True object-oriented programming is supported in the sense that remote references to replicated services may be passed transparently as serialised arguments in remote invocations. We compare our framework to other replication architectures and illustrate the performance with run-time measurements.

KEYWORDS

Replication, Middleware, .NET Remoting, Scalability, Reliability

1. INTRODUCTION

Distributed object middleware systems are highly popular for the development of network-based applications. Many of such applications demand system support to enhance their quality-of-service properties regarding reliability, scalability, and efficiency. The Microsoft .NET remoting framework is an increasingly popular infrastructure for the development of distributed applications. Its architecture is highly flexible, as it supports various communication mechanisms (e.g., binary TCP channels and SOAP-based HTTP channels) and offers various ways for extending the default remote interaction mechanisms. However, no substantial previous work can be found on how these possibilities are best utilised to create replicated services.

This paper surveys established approaches for implementing replication in other middleware systems and presents an overview of the .NET remoting architecture and its extension mechanisms. We present the architecture of our flexible replication framework for .NET services. It seamlessly integrates into the usual .NET environment. The architecture offers active replication for fault tolerance based on a group communication layer as well as load balancing among replicas to increase the scalability of services. Unlike many other replication infrastructures, it is fully transparent to clients and allows true object-oriented programming. A local smart proxy is automatically instantiated when a client binds to an existing replicated service, when it creates a new service instance, and when it receives a serialised remote reference.

The remaining paper is organised as follows. Section 2 summarises general approaches for a fault-tolerant middleware architecture. Section 3 outlines the extensibility features of the .NET remoting

framework. Section 4 describes our replication architecture in detail. Section 5 evaluates our approach and gives some performance measurements. Section 6 presents concluding remarks.

2. REPLICATION IN OBJECT-ORIENTED MIDDLEWARE

Replication may be used for load balancing, and thus increases the scalability of services in a distributed system. Client requests are forwarded to only one out of a set of replicas. This is mainly used for requests that do not modify the service state. Besides, replication adds redundancy to the system, which allows tolerating failures of some nodes and results in better service availability. Appropriate strategies to keep replicas consistent are required. In a passive replication scheme, only one primary executes requests and multicasts state changes to all replicas. This scheme avoids redundant computation of requests and copes with non-deterministic service behaviour. Active replication in contrast multicasts a client request to all replicas, which in turn execute it individually. Often, this requires less network resources than sending a state update; in addition, reaction to failures is much faster than in a passive scheme. However, replica consistency usually requires deterministic replica behaviour.

2.1 Replication with the CORBA Middleware

Several research projects have analysed ways for providing fault-tolerance in distributed object middleware. Many of them focus on the CORBA middleware. FT-CORBA provides a general standard for fault tolerance in CORBA systems, without specifying exact implementation details (OMG, 2000). Specific approaches to CORBA fault tolerance can usually be classified into the following categories: the interception approach, the service approach, and the integration approach.

The interception approach has first been implemented in the Eternal CORBA system (Narasimhan et al., 1997). It intercepts IIOP message at the interface between the CORBA ORB and the underlying operating systems. This way, any off-the-shelf ORB may be used, and replication is offered in a fully transparent way for both client and server. However, adequate support by the operating systems for such interceptions is needed. The AQuA framework (Krishnamurthy et al, 2003) uses a similar approach. It translates GIOP messages to group communication primitives of the Ensemble group communication system.

The service approach is used in OpenDREAMS (Felber, 1998). The fault-tolerance mechanisms are encapsulated into a CORBA service. Application objects directly interact only with a local Object Group Service (OGS), which then coordinates itself with other OGS instances and performs the operation on the replicas. This approach does not provide replication transparency (the client has to be aware of the OGS), but benefits from the fact that no proprietary extension to the ORB or the operating system are needed. The DOORS system is another example that similarly uses the service approach.

In the integration approach, the ORB is modified to provide the desired fault tolerance support. This usually means that interoperability with clients running on unmodified off-the-shelf components is not supported. Orbix+Isis (Birman&van Renesse, 1994) and Electra (Maffeis, 1995) are examples where the integration approach has been used.

A few projects do not fit exactly into one of the three classes, but rather use some hybrid approach. A typical example is FTS (Friedman&Hadad, 2002), which combines CORBA portable interceptors with a custom group object adapter (GOA) to provide fault tolerance in a non-FT-CORBA-compliant way.

2.2 Replication with the .NET Remoting Framework

The .NET remoting framework is a rather new middleware system, and no research project known to the authors has explicitly addressed replication strategies for .NET services. Some systems (e.g., Borland Janeva (Borland, 2003)) implement bridges from .NET to CORBA/IIOP that might allow accessing fault-tolerant CORBA services from .NET applications.

However, a native .NET-based replication framework has its benefits compared to such bridges, and also compared to other (e.g., CORBA-based) systems. First of all, from a .NET developer point of view, using the

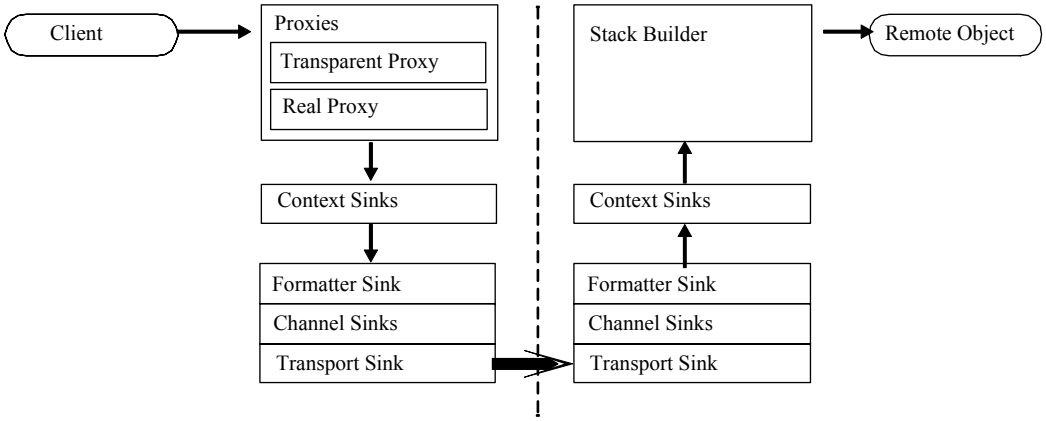

Fig. 1. The .NET Remoting Architecture

same technology—i.e., .NET—for client and service development is a significant simplification. More important, bridging mechanisms add some additional latency, which reduces service performance.

In addition, the .NET remoting framework offers flexible extensibility features that allow to achieve replication transparency for clients and a true object orient programming model, where remote references to replicated services may be passed transparently in the distributed system as reference parameters. Neither the operating system nor the middleware layer needs to be modified for such purpose, as we show with our replication architecture.

3. THE .NET REMOTING FRAMEWORK

3.1 Overview of the Remoting Architecture

Figure 1 illustrates the basic architecture of .NET remoting. A protocol stack composed of several components handles the interaction between client and remote object. With the exception of the transparent proxy and the stack builder, all components can be modified arbitrarily (McLean et al, 2003).

Transparent proxy and real proxy together constitute the proxy ("stub") of the remote object. The transparent proxy is automatically created from metadata (the interface signature) at runtime and delegates to the real proxy. The default real proxy forwards any client call to the sink chain. However, it may completely be replaced by a custom implementation. Because of this, it is a potentially good point for arbitrary extensions.

An arbitrary number of context sinks may be used to pre-process the invocation prior to serialisation. Its main purpose is to pass context information (e.g., thread priority) from client to server. Context data is added to the call data structure at the client and received and evaluated at the server side.

The channel consists of (de-)serialisation methods (the formatter sink), an arbitrary number of channel sinks that operate on the serialised data (e.g., to perform encryption/decryption), and a transport sink that is used to transport invocations over the network (e.g., a TCP transport or a HTTP transport).

3.2 Activation Strategies

The .NET remoting framework supports various *activation types* for remote objects. Each time a client calls the `new` operator or the runtime method `Activator.GetObject` for a *client activated object (CAO)* type, a new instance of the remote object is created. A *server activated object (SAO)* may either be a *single-call* object, which exists only for the duration of one method invocation. Or, it can be a *singleton* object, which has only one instance that is shared between all clients. The activation type of an object is defined in the local .NET runtime configuration.

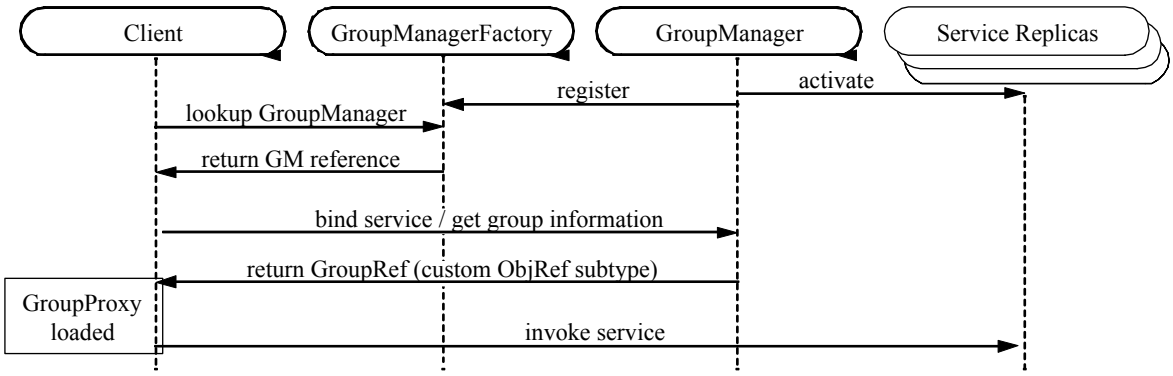

Fig. 2. Basic Structure of the Replication Architecture

3.3 Loading the Local Proxy and Sink Chain

The .NET remoting framework offers various methods for binding to a remote object and thus creating a local proxy for it. If an object type is registered as remote in the runtime configuration, a local proxy is transparently loaded when the `new` operator is called on that type (this does not work for interfaces or abstract types). Alternatively, the runtime method `Activator.GetObject` may be called explicitly to create a local proxy. In both cases, the transparent proxy is created automatically from the type signature in the service metadata that has been linked to the client. If an attribute of type `ProxyAttribute` is attached to the metadata, the attribute implementation is evaluated. It may provide a custom `CreateInstance` method that instantiates a custom proxy. One drawback of this approach is its strong link between a *service type* (class name) and a *service instance* (instance of that class). Remote references to two different instances of the same type using a different remoting configuration cannot be supported.

To avoid this problem, it is customary to use a factory approach for stub creation. This may be a local factory that loads the proxy and registers it with the remoting framework manually. It may also be a remote naming service, which maps a unique name to a service instance and returns a reference to this instance to a requesting client. This return value can be sent as a serialised `ObjRef` object (a default .NET remoting data type that encapsulates the service address and the object type information). Upon deserialisation of such an object, .NET remoting automatically creates a proxy for the referenced remote object, based on information in the `ObjRef` object. Custom subclasses of `ObjRef` can contain further information (e.g., all addresses of a replication group) and may overload a helper function that allows to load a custom real proxy at the client instead of the default proxy. In our replication framework, we support both ways for loading our local proxy.

4. ARCHITECTURE OF THE REPLICATION FRAMEWORK

4.1 Overview

In the design of our replication framework, extensions to the operating system—as in the interception approach—or proprietary modifications to the basic remoting architecture—as in the integration approach—have been discarded. The extensibility features of .NET remoting allow to create a framework that works with any off-the-shelf .NET environment and thus is easy to use and deploy. It is portable in the sense that it runs on any host that provides a .NET execution environment. With the availability of base systems like Mono (Novell, 2004)[], this may even include non-Microsoft platforms.

For a client, accessing a replicated service only requires linking against a service-specific library. In our current implementation, this library contains the service metadata (its interface description), our custom proxy implementation, and dependent classes. On the server side, the implementation of a replicated service differs only slightly from a non-replicated variant. If active replication is used for fault tolerance, modifying

operations need to behave deterministically. If replicas shall be automatically recovered after crashes or new replicas be added dynamically at run-time, mechanisms for state transfer between replicas are needed (see Section 4.3.4). In the following, we first give a general overview of the components of our replication architecture. After that, each component is described individually in more detail.

4.2 Building Blocks of the Replication Architecture

The basic structure of our replication architecture is outlined in Figure 2. A replication group is administered by a `GroupManager` component. It activates all service replicas as client activated objects and registers itself at a `GroupManagerFactory`. To locate the group manager of the replicated service, the client uses a statically configured `GroupManagerFactory` to obtain a reference to the group manager. The group manager itself can be replicated for high availability. We will explain the group management in more detail in Section 4.3.

With the help of the group manager, the client transparently loads an adequate group proxy. The configuration (i.e., location of replicas, replication strategy, etc.) is obtained from the group manager. Several variants of the group proxy exist for load balancing and for fault tolerant replication. These will be discussed in Section 4.4.

4.3 Group Management

The `GroupManager` is responsible for all tasks related to the administration of the replication group. It maintains information about the location of replicas and performs the activation of replicas on each host. Furthermore, it stores policies that select replication strategies. It interacts with the client, when the client needs to load an appropriately configured remoting proxy for accessing a replicated service. In addition, it supports dynamic group management. Based on service policies it may, for example, react to failures or high load by automatically creating new replicas on some available locations. In such situations, it also coordinates the state transfer necessary to add new replicas to the group.

4.3.1 Obtaining the Initial Reference to the Group Manager

Initially, the client has to bind to a remote group manager. Each replication group may have its own replication manager. To encapsulate the binding to the group manager, a factory design pattern is used. The `GroupManagerFactory` locates an adequate group manager for a requested remote object. The data returned by the factory allows the client to load a local custom proxy. A static configuration at the client defines which `GroupManagerFactory` is used.

A failure of the group manager, albeit less severe than the failure of an remote service, would reduce the functionality of the replication group: New clients are no longer able to bind to the replication group, and dynamic reconfigurations of the group are no longer possible. To alleviate this problem, the group manager itself may be replicated. This replication is possible with exactly the same mechanisms as for the replicated service itself. The only difference is that the group manager is an unmanaged replication group (it does not have a replication manager). Locating and binding is supported directly by the `GroupManagerFactory`. Reconfiguration of a group manager's replication strategy requires that the group manager implementation updates information stored in the `GroupManagerFactory` manually.

4.3.2 Locating and Activating of Replicas

The group manager knows all locations and contact addresses of replicas. Management may be performed either manually (the group manager is manually informed about the current location of replicas) or automatically. In the latter variant, the manager may detect the failure of replicas. Furthermore, it might know about potential execution locations and might be able to autonomously create new replicas at appropriate locations. The details of such automated management are, however, beyond the scope of this paper.

For dynamically adding new replicas as well as for recovering crashed replicas, mechanisms for state transfer (except for a stateless service) are required. Such state transfer is hard to implement in a generic, transparent way. Consequently, we require the developer of a replicated service to provide state transfer

mechanisms explicitely. An interface `IStateTransfer` with two methods `getState()` and `setState()` needs to be implemented by a replicated service.

Our framework supports all activation types of .NET remoting (cf. Section 3.2). The server-side service objects usually work as CAO. From a client point of view, the desired activation strategy is stored as a policy in the group manager. As soon as the client contacts the group manager to bind to some service, this policy is taken into account. For a CAO policy, the group manager contacts all replication hosts to create a new replica instance for the object whenever a client binds to a replicated object. For the SAO/singleton policy, all clients obtain a reference to the same replication group, which initially was activated by the replication manager. The SAO/single-call policy is only available in a load-balancing replication scheme. Here, the local custom proxy at the client is configured to forward each method invocation to a different replica; the replicas itself are implemented as single-call objects.

4.3.3 Implementing the Client

A client needs to contact first the group manager factory, then the group manager, to finally load a local group proxy to access the replicated service. All these steps may happen completely transparent to the client. There are only two prerequisites: The client has to be linked against a metadata assembly that contains the attributed interface of some remote service and the implementation of the attribute `GroupProxyAttribute`. Besides, it needs to be configured that the service is remote and that a certain `GroupManagerFactory` should be used.

```
public void Main(string[] args) {
       ChannelServices.RegisterChannel(new TcpChannel(0));
       RemotingConfiguration.RegisterActivatedClientType(typeof(Service), “”);
       RemotingConfiguration.RegisterWellknownClientType(typeof(
              GroupManagerFactory),“tcp://<ip>:<port>/GroupManagerFactory.rem“);
       // (a) Use service with custom remoting proxy, transparent binding
       Service a = new Service();
       a.MyMethod();
       // (b) Use replicated service with manual binding via lookup service
       GroupManagerFactory gmf = new GroupManagerFactory();
       GroupManager gm = gmf.GetGroupManager(“Service“);
       Service b = (Service)gm.GetService(“Service“);
       b.MyMethod();
}
```

In the variant (a), the operation `new Service()` detects that `Service` is registered as a remote object. The assembly contains an attribute that causes the corresponding implementation to be loaded and executed. This attribute code binds to the group manager factory to locate the group manager for `Service`. It interacts with the group manager to instantiate an appropriately configured group proxy at the client side. The new operation returns a reference to this group proxy.

The first variant does not differentiate between a type and an instance of this type. Variant (b) shows an alternative, where the group manager is created (via the group manager factory) and contacted manually. The method `GetService` returns a serialised `GroupRef` object (which is our own subtype of `ObjRef`). It contains all necessary information about the replication group (replica locations, replication policy, etc.) and a custom method to load the appropriate local group proxy.

4.3.4 Implementing the Server

The implementation of a replicated service is similar to a non-replicated service; it is however subject to the following constraints: For load-balancing, the remote methods cannot modify a global service state; for fault-tolerant active replication, the replica behaviour must be deterministic.

```
[GroupProxyAttribute()]
public class Service : ContextBoundObject, IStateTransfer {
       public Service() {...} // constructor
       public void MyMethod() { ,,,} // method implementation
       //     IStateTransfer interface implementation
       public State GetState() {
```

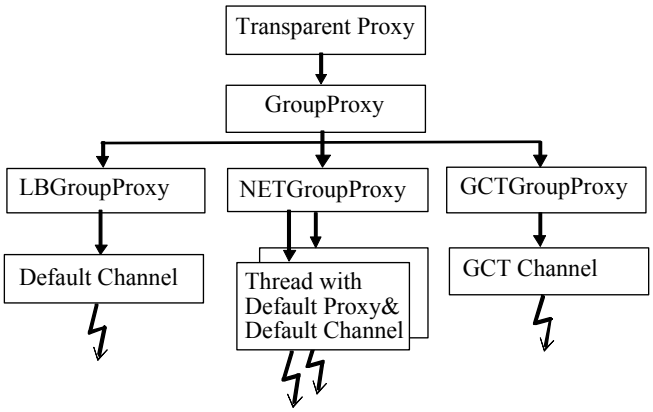

Fig. 3. GroupProxy variants

```
            Hashtable properties = new Hashtable();
            properties[“data“] = some_data;
            return new State(properties);
      }
      public void SetState(State state) {
            some_data = (type_of_some_data) state.Properties(“data“);
      }
}
```

The service has to implement the interface `IStateTransfer` (cf. Section 4.3.2), unless the service is stateless. The attribute `GroupProxyAttribute` is attached to the service implementation.

4.4 Group Proxy Alternatives

The group proxy is the key component of our architecture. All invocations from a client to a replicated remote service are locally passed to the group proxy via the default transparent proxy. The constructor of the group proxy is responsible for setting up all other components that are required for the remote interaction.

Three different variants of the group proxy are available in our replication architecture (Fig. 3). Which type is loaded depends on the requirements of the service application to be replicated. Typically, these requirements are stored as policies in the group manager of the replication group. When binding to a remote replication group, the appropriate proxy is transparently loaded at the client.

Variant 1: Using replication for load balancing

For load balancing purposes, one of all available replicas has to execute a non-modifying request. Our `LBGroupProxy` implementation currently offers two strategies for selecting an appropriate replica. First, a client may use a round-robin or random selection strategy. This way, no additional control information needs to be exchanged. As the real load of the replicas is not considered at all, the load distribution is however sub-optimal.

Second, a client may also query the replica group periodically to learn about the replicas´ load. The update period is selected via a group manager policy. The service object can provide an implementation of a `IPerformanceManager` interface. This allows custom specification of what kind of load to be considered (CPU load, network load, available memory, etc.). Better load distribution is achieved at the cost of additional communication overhead.

A third strategy, where the group manager monitors replica load and passes this information to clients, could also be investigated. Currently, this has not yet been implemented.

Variant 2: Active replication with default .NET channels

In this variant, a client directly connects to all replicas of the application service. Upon `NETGroupProxy` construction, the URIs of all replicas are obtained from a `GroupRef` object (our subclass of the `ObjRef`

# replicas	1	2	3	4	5
default .NET	1.0 ms				
NETGroupProxy	1.0 ms	1.2 ms	1.5 ms	1.7ms	2.0 ms
GCTGroupProxy	10 ms	31 ms	55 ms	87 ms	124 ms

# replicas	1	2	3
default .NET	120 ms		
3 clients, query load	370 ms	205 ms	145 ms
3 clients, random	370 ms	220 ms	180 ms

Fig. 4. Average execution times for fault-tolerant remote invocations of a no-op method (left) and for load-balancing with the LBGroupProxy (right)

class that is used for marshalling remote references). The proxy creates a standard .NET remoting connection to each replica, using the default real proxy and an out-of-the-box channel. Each connection is handled by an individual thread to enable parallel invocations of methods at each replica.

This variant has two serious drawbacks: First, total order of invocations is not guaranteed. Two invocation requests of different clients may get executed in an inconsistent order among the replicas. Second, a client crash may have the result that some replicas have executed that client's request, while others have not.

The advantage of this variant is that all available types of .NET remoting channels may be used. This is even possible in a mixed way. For example, some clients may access the service replicas via a binary TCP channel, while others use a SOAP-based HTTP channel.

This variant is suitable for replicated objects that are not accessed from more than one client simultaneously, or that are stateless. A typical example of such stateless application is a computation service that performs some complex calculations based on data provided by the client. The service needs to be replicated in case that the client requires the computation result in a timely manner even if one service node fails. Client requests are computed simultaneously in all replicas, and if one replica crashes, the client will still receive a response from another replica without delay.

Variant 3: Using a group communication system

For strong consistency among stateful replicas, a total order of all invocations at the replicas is obtained by a group communication system. In our implementation we use the Group Communication Toolkit (GCT), a C# implementation of the JGroups architecture [].

In this variant, the group proxy constructor has to obtain a channel of the group communication system. The group address is known to the client via information obtained from the group manager. Unlike the other two cases, the server has to be configured such that it is accessible via the GCT transport (Variant 1 and 2 use off- the-shelf .NET remoting channels).

This version is best suited for service applications that need to be highly reliable and are stateful. The group communication layer ensures total order of all method invocations, and thus provides strong consistency for replicas with deterministic behaviour.

5. PERFORMANCE EVALUATION

We have implemented several sample applications that make use of our replication framework. In Section 4.3 we already presented some code examples that show the necessary effort to create a client and a replicated service with our framework.

To analyse the performance of our architecture, two aspects have been investigated: First, what is the additional overhead created by the replication scheme? Second, what speed-up may be achieved with our load-balancing scheme? Two different test applications are deployed. All following measurements have been done on Windows XP-based PCs with 1.8 GHz P4 processors and a 100 MBit/s local network.

To evaluate the overhead of our active replication schemes, a simple remote service with only one method that immediately returns to the caller was used. A remote client calls this method repeatedly. We compare the remote interaction without replication (using a single server and the default .NET binary TCP channel) with a replicated service based on (a) the NETGroupProxy and (b) the GCTGroupProxy (see Figure 4, left table). For the NETGroupProxy, the round-trip invocation times equal those of a standard .NET

remoting channel with only one replica, and increase slightly with an increasing number of replicas. For a realistic application, this overhead (only 1 ms for 5 replicas) is negligible. Providing strong consistency with the GCTGroupProxy is significantly more expensive, and is thus feasible only for a small number of replicas. It needs to be investigated if other consistency protocols might yield better results.

The second scenario uses the LBGroupProxy to load-balance client requests on a replication group. The service computes an image; computation including default .NET marshalling from/to the client take 120 ms. Three clients use this service repeatedly. Figure 4, right table, shows the possible speed-up with several replicas. The first variant queries the load of all replicas prior to service invocations. This achieves well distribution of load with an almost constant cost of 20-25 ms for querying the current load. The second variant uses a random scheme for replica selection. This variant avoids the cost for the load query, however achieves a sub-optimal load balancing. In the example of Figure 4, it is always inferior to the load query model.

6. SUMMARY

We have presented an architecture for the replication of .NET service applications. It makes use of the flexible extensions mechanisms of the .NET remoting framework. This way, an integrated, portable solution is obtained. An important feature of our architecture is that service replication can be fully transparent to a client application. Furthermore, remote references to replicated services may be passed arbitrarily as remote method parameters, allowing true object-oriented programming. The management infrastructure offers two activation policies for replicated objects (server or client activation), serves as a replica location service for clients, manages the monitoring of replica availability and state transfer for replica recovery. Three replication variants offer load balancing, replication of stateless computation services, and consistent active replication based on group communication.

This .NET replication architecture was developed in the context of the AspectIX project. Additional information is available at `http://www.aspectix.org`. Our further research will address dynamic management of replication groups in more detail. A policy-based system will allow, e.g., automatic creation of replicas to fulfil QoS policies and an automated discovery of available resources for replica creation. Also, little effort was spent up to now on optimising the efficiency and latency of our prototype system, so we anticipate further improvement.

REFERENCES

OMG (2000). Fault Tolerant CORBA specification, v. 1.0. omg. OMG document ptc/00-04-04

P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith (1997). Exploiting the internet inter-orb protocol interface to provide corba with fault tolerance. In Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems

S. Krishnamurthy, W. H. Sanders, and M. Cukier (2003). An adaptive quality of service aware middleware for replicated services. IEEE Transactions on Parallel and Distributed Systems, 14(11):1112-1125

P. Felber (1998). The CORBA Object Group Service. PhD Thesis, Lausann, EPFL

K. Birman and R. van Renesse (1994), Reliable Distributed Computing with the Isis Toolkit, IEEE Computer Society Press

S. Maffeis (1995). Adding Group Communication and Fault-Tolerance to CORBA, Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS), Monetry, California

Friedman and E. Hadad (2002). FTS: A high-performance corba fault-tolerance service. In Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems

Borland Software Corp. (2003). Borland Janeva Overview. An innovative approach to platform interoperability using proven technology; A Borland White Paper, July.

C. Koiak, P. Nixon (2003). The Group Communication Toolkit (GCT). Available from <http://www.smartlab.cis.strath.ac.uk/Projects/ GCTProject/GCTProject.html> [Accessed 2004-12-01]

S. McLean, J. Naftel, K. Williams (2003). Microsoft .NET Remoting

Novell, Inc (2004). Mono: A comprehensive open source platform baesd on the .NET framework. Available at <http://www.mono-project.com/>. [Accessed 2004-12-01].