Integration of a Dynamic Object Replication Framework in Java

Thomas Strauß
Darmstadt University of Technology
D-64283 Darmstadt, Germany
tstrauss@mainz-online.de

Oliver Theel
University of Oldenburg
D-26111 Oldenburg, Germany
theel@ieee.org

Abstract

By the use of dynamic *replication schemes – in comparison with* static *replication schemes – a tremendous increase of read and write operation availabilities can be achieved. Furthermore, object-oriented programming languages like Java offer excellent possibilities for the abstraction of complex problems. In this paper, we present our solution of combining dynamic replication and its use in object-oriented programming. More precisely, we show how Java objects can be made highly available through the use of dynamic replication in an almost transparent manner from the Java application programmer's perspective. This is done by using a general, highly flexible framework which enables the use of a large number of dynamic replication schemes. We show how the framework can be integrated, such that it is conveniently available for Java application programmers. We demonstrate its simple use, even when being introduced* a posteriori *in a Java program: one need only to modify a few lines of code.*

1 Introduction

The replication of objects in a distributed system is a viable means for improving the system's efficiency and for increasing the object's availability. A *replication scheme* manages multiple replicas of an object to continously provide the object even if some of the replicas fail. Furthermore, the replication scheme guarantees the *consistency* of the replicated object. In most settings, *1-copy-serializability* (1SR) is chosen as correctness notion [3]. 1SR assures that the execution of operations on a replicated object is equivalent to the execution of the same sequence of operations on a non-replicated object. Object replication can be divided in two categories. One category is called *active replication*. In active replication, multicast communication is used such that *all* replicas of an object receive and process calls from a client. But only a primary object provides results to the client. When the primary object fails, a backup object takes over the primary object's role of providing continuous services. The other category is called *passive replication*. In passive replication, only a dedicated replica – which may be a different one from call to call – out of the group of replicas representing the object receives and processes a call. The other replicas – or a dedicated subgroup thereof – passively wait for the state change provided by the invoked replica.

This paper focuses on the passive replication approach. Passive replication in the object-oriented context is as follows: Objects share two characteristics. They exhibit 1) a so-called *state* and 2) a so-called *behavior*. Informally, the state is what an object "knows" and the behavior is what an object "can do." Software objects like Java objects implement their behavior with *methods* and maintain their state in one or more *variables*. All methods and all variables of an object are defined in a so-called *class*. A particular object is called an *instance* of a class. The state of all variables of an object is called *object state*. Hence, two instances of the same class may only differ in their object state.

In our approach, the class definition of a replicated object must be installed on all sites that host a replica of an object in a mutually consistent manner. The methods must not be managed by a replication scheme since they do not change over time. The replication scheme only manages the object state of a replicated object in order to guarantee 1SR. This object state, from the perspective of the scheme, is treated like a *data image*: the data image must be read before a method can be applied and it must be written after the state has changed due to the execution of the method on the object. The advantage of only having a data image with a dedicated read and write operation allows the adoption of replication techniques originally intended for pure data. The corresponding replication schemes are therefore originally called *data replication schemes*.

During the last decade, researchers have spent tremendous efforts on trying to identify data replication schemes which offer read and write access operations on replicated data objects with both, high operation availabilities and low operation costs. It turned out that a good design strategy is to logically arrange the physical replicas according to some logical structures. Examples for successful structures are trees [1, 5], finite projective planes [8], grids [4], and acyclic graphs [14]. Although good solutions could be identified for some application scenarios (specified by the ratio of read to write operations), it became obvious that there is no single data replication scheme which is fairly good suited for *any* application. Thus, depending on the application, a particularly well suited data replication scheme must be chosen.

Since the data structures (and consequently the interpreting algorithms) vary substantially from one data replication scheme to another, *unifying frameworks* have been developed [2, 14, 15]. Such a framework allows the specification and usage of a large number of different data replication schemes by easy means and in a homogeneous manner. By simply redefining the specification, another data replication scheme can be adopted within an application scenario. It is this flexibility which allows to exploit the benefits of replication in an extremely time- and cost-saving manner.

Replication schemes with structures assume a certain placement of replicas within this logical structure. Once structure and placement have been both specified, they must remain unchanged throughout the lifespan of the replicated data. In other words: the specification (or *configuration*) does not change. The prerequisite of having an immutable configuration releaves the replication scheme from additional management overhead resulting in simpler and less costly algorithms, since, for example, "configuration version control" is not required. Allowing a configuration to change over time, on the contrary, embodies the potential of dynamically reacting to changing network characteristics, thereby leading to replication schemes with highly increased operation availabilities. In the following, we call replication schemes having immutable configurations *static* schemes, whereas schemes which allow configurations to change over time are called *dynamic*.

Dynamic General Structured Voting (dGSV) [16] is a framework that allows the modeling of a large variety of dynamic replication schemes whose non-dynamic counterparts are explicitly or implicitly based on coteries [6]. This includes coterie-based schemes exploiting priorities, like, e.g., the Tree Quorum Protocol [1]. This is done by a general method for systematically and automatically transforming static coterie-based replication schemes into their dynamic counterparts.

The emphasis of the work presented in this paper is to use the dGSV framework for Java object replication. In particular, we show how the framework manages replicated Java objects and how a Java program that exploits this technique generally looks like. It will turn out that the benefits of object replication can be exploited without the need for the programmer of explicitly coding a replication scheme within Java. A programmer and the object does not need to know anything about replication. In this sense, our approach is almost transparent. Additionally, object replication can be easily included in programs *a posteriori* by changing only a few lines of code.

The remainder of the paper is organized as follows. In the next section, a brief description of dGSV is given. In Section 3, we integrate the dGSV framework into Java. Section 4 presents a sample Java program that exploits object replication in the manner proposed. In Section 5, we present related work and Section 6 concludes the paper.

2 Dynamic General Structured Voting

2.1 Basic Idea

In the approach to dynamic replication of Rabinovich and Lazowska [10], the *dynamics* of a replication scheme must be specified on a per-replication-scheme basis. For instance when modeling the Grid Protocol [4], which logically arranges the N replicas of a data object in form of a $m \times n$ grid (if $m \cdot n > N$ then the m-th line of the grid is not complete), a rule must *a priori* be defined that states the shape of a grid to be used when $i \leq N$ replicas are available. Furthermore, it must be clear which replica resides in which position of the grid. We call a rule providing answers to both aspects a *structure rule*. Additionally, other rules must precisely state what subsets of available replicas, based on a particular grid, can be used for consistently performing read (write) operations. For the Grid Protocol, valid subsets of replicas for reading consist of at least one replica of every column of the grid (called *c-cover*), whereas for writing, a c-cover together with an entire column is required. For ease of description, we call the rule delivering subsets used for read (write) access *read (write) quorum rule*. All three rules together are referred to as *rule set*.

dGSV, on the contrary, uses a general fixed rule set that is able to model *every* replication scheme based on coteries. The core problem that had to be solved, was to come up with a rule set that leads to a good dynamic behavior of every modeled replication scheme without exploiting replication scheme-specific knowledge. This seemed to be a tough problem, since good dynamic behavior of, e.g., a Grid Protocol substantially differs from a good dynamic behavior of a Tree Quorum Protocol [1].

2.2 Replication Scheme Specification

Figure 1 shows the specification of the static Tree Quorum Protocol (TQP) [1] as used by dGSV. Such a specification is called a *master voting structure*. The master voting structure derives at run-time valid *read (write) quorums* as well as the sequence in which these quorums must be used for correct operation execution. A particular read (write) quorum is a set consisting of replicas which must *all* be contacted when consistently performing a read (write) operation. A voting structure is an acyclic connected graph with a unique root. The leaves of the voting structure represent *physical nodes* (shown as boxes), whereas inner knots represent *virtual nodes* (shown as ovals). Virtual nodes exist for grouping purposes. Nodes have an upper and two lower indices. The upper index is called a *vote*. Physical nodes, by default, host a replica. The left (right) lower index of a

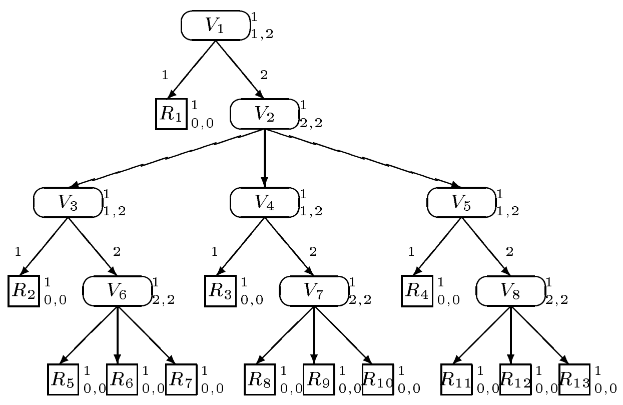

Figure 1. Master voting structure of TQP initially managing 13 replicas

node is the *read (write) operation quorum*. Starting from the root, a possible *quorum set* is obtained by including as many entities of the next lower level in the quorum set as necessary for satisfying the operation quorum. A quorum is *satisfied* if the sum of the votes of the chosen entities is at least as large as the operation quorum but no entity can be removed such that the sum of votes does not drop below the quorum. Weights applied to the arcs indicate *priorities*. Priorities define which quorum sets are used *prior* to other ones. The lower the weight, the higher the priority. If no weight is given, then the weight is assumed being ∞ (e.g., having minimal priority).

As shown in Figure 1, when given the master voting structure as input, 13 replicas are managed by TQP. The dGSV algorithm responsible for deriving quorums interprets the voting structure at run-time in correspondence with the original TQP specification: a read operation first tries to read the replica located at physical node R_1. If R_1 is not available, then it is tried to contact two of the three replicas located at physical nodes R_2, R_3, and R_4. Whenever it turns out that any of the chosen replicas of the former group is not available, then two out of three "spare replicas" must be contacted instead: if $R_i, i = 2, 3, 4$, is not available, then it must be replaced by two replicas of $\{R_{5+3\cdot(i-2)}, R_{6+3\cdot(i-2)}, R_{7+3\cdot(i-2)}\}$. In case of a write operation, R_1 together with two of the replicas at R_2, R_3, and R_4, as well as two out of three spare replicas for every chosen replica among R_2, R_3, and R_4 must be written.

Since dGSV may detect failures of network components leading to re-configuration actions, the voting structure actually used for deriving operation quorums may be different from the master voting structure. Based on the fixed, *a priori* known master voting structure and a *current epoch*, dGSV calculates a so-called *current voting structure*. An *epoch* is a set of sites holding replicas that were available at the time of an epoch formation. An epoch formation can only be performed under certain consistency-preserving conditions. The epoch obtained by the last epoch formation is called *current epoch*. The *current voting structure* includes exactly those replicas in the current epoch. Using the current voting structure, 1SR compliance is always guaranteed.

Note, that only when all replica-hosting sites are in the current epoch then the current voting structure is morphologically identical with the master voting structure. In all other cases the current voting structure is calculated by deleting all sites from the master voting structure that do not belong to the current epoch, followed by some reorganization of the structure. For further details, please refer to [17] where dGSV has been introduced in detail.

3 Integration into Java

In this section, we describe, how dGSV can be integrated into Java. The dGSV framework described before, supports convenient and flexible replication of data. To utilize dGSV for the replication of Java objects, we regard the object state as the data to be replicated at a dedicated set of sites. The class definition of the object is immutable and therefore must not be dynamically managed. Class definitions must only be installed at all participating sites. Furthermore, all classes required by the dGSV run-time system must be co-located at the replica-hosting sites. These sites are synonymously called *replica servers*. Whenever a client submits a method call to a replicated object, the call must be redirected to an arbitrary replica server of the object. Then, the replica server acts as *coordinator* for and during the method execution. In the course of such an execution, firstly, a read operation on the replicated data is performed via the dGSV method. As a result the actual object state is locally available. Secondly, the requested method is executed on the obtained, actual object state. Thirdly, the new object state is written to a qualifying set of replicas in the scope of a write operation under dGSV control. For consistency reasons, all replicas participating in the method invocation must be write-locked throughout all three steps.

Since the object state must be exchanged between the participating replica servers, it is necessary that the corresponding class be *serializable*. In Java serializability of a class indicates that the state of its objects can be written into and restored from e.g. a stream. A class is declared to be serializable by implementing the `java.io.Serializable` interface.

For communication among sites, we use the Java *RMI (Remote Method Invocation)* system [13]. In Java, the RMI system basically corresponds to an RPC (remote procedure call). It allows the invocation of an individual call in different address spaces and on different sites, thereby automatically and transparently encoding and decoding information transferred between the involved processes. Using RMI, a client process basically has the illusion of calling a local method. Another important issue in the scope of the envisioned integration is to provide a high degree of *replication transparency* from the Java application programmer's point of view. For this purpose, we integrate dGSV into the RMI architecture in a way, such that a call to a replicated object is performed analogously as a call to an RMI remote object. This well-known and successful technique allows an easy migration from an ordinary RMI object to a somewhat more sophisticated replicated dGSV object.

The RMI architecture consists of several layers. At the client side, the call of a method is taken from an *RMI stub object*. This RMI stub object sends the call – transparent for the caller – to the server. At the server side an *RMI skeleton object* receives the call and forwards it to the server object. The server object handles the call and provides the result, which is taken by the RMI skeleton and sent back to the RMI stub. Subsequently, the client receives the result from the RMI stub. RMI stub and RMI skeleton form the top layer: the protocol layer. It lies above the reference layer which contains a naming service. This naming service is used by the client to obtain the RMI stub object. Beneath the reference layer is the transport layer. Here, connections of the communication are managed and RMI communication is done. The transport layer, finally, lies on top of TCP/IP. As shown in Figure 2, the integration of dGSV is

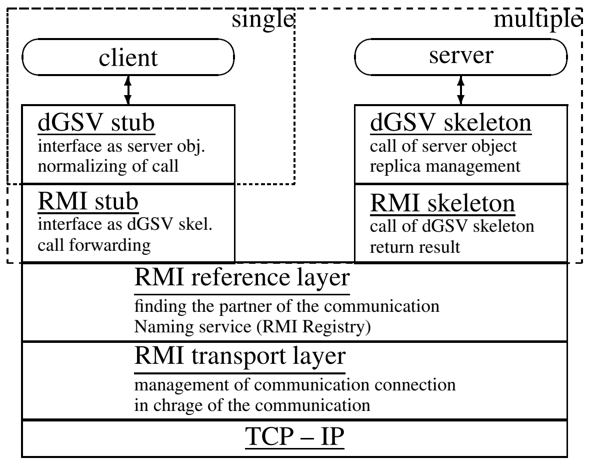

Figure 2. dGSV Integration

achieved by adding an additional *dGSV layer* between the client and server objects on top and the RMI protocol layer. The dGSV layer consist of a *dGSV stub* and a *dGSV skeleton*. It is important to note that at invocation time there is only a *single* client involved in communication but usually *multiple* server objects managing replicas. Consequently, there exist as many RMI skeletons and dGSV skeletons as server objects. A client object does not call the methods of the RMI stub but the methods of a dGSV stub, which contains exactly the same methods like the original RMI stub. The dGSV stub implements (like the original RMI stub) the same interface as the server object. Within the dGSV stub, a normalization of a call is done. Beyond this, in the dGSV stub, a particular replica server for the interaction with the client is selected. Then, the normalized call is sent over the corresponding RMI stub to the selected replica server. This corresponding RMI stub does not implement the interface of the server object but an interface of the dGSV skeleton. The dGSV skeleton is located above the RMI skeleton. It receives the normalized method calls from the RMI skeleton which are sent over the RMI stub to the RMI skeleton and prepares it to be sent to the "real" server object. Before sending the method calls to the server object, the dGSV skeleton triggers replica management. This replica management guarantees that the state of the server object is actual for the course of a method call. It is important that the RMI protocol layer with RMI stubs and RMI skeletons does not change from one particular class of server object to another class: only normalized method calls of dGSV are transmitted. Consequently, RMI stub and skeleton are always the same and the corresponding classes must not be generated each time anew. What has to done is that the dGSV stub must implement the sever object interface and furthermore, the stub is responsible for the normalization of the method call.

3.1 Normalization of Method Calls

The normalization of method calls is done in a dynamic *proxy* object. Being a well-known technique, this proxy is generated using the Java Dynamic Proxy API [12] at runtime. The proxy object implements the interface which contains the remote methods of the server object. All method

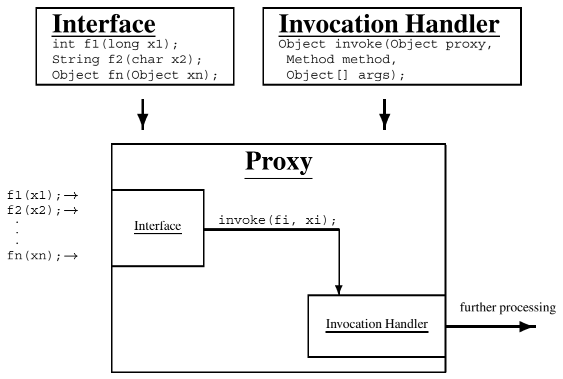

Figure 3. Use of the Java Proxy class

calls to the proxy are internally redirected by the proxy to a so-called *invocation handler* [12]. The invocation handler is assigned to the proxy at proxy creation time. It is called by the proxy only by means of a general invocation method called *invoke*. As the signature of the invoke method of the invocation handler remains constant, the invocation handler can be designed uniquely. Any further processing of a normalized method call is done by the invocation handler.

3.2 Selection of a Replica Server

There exists one dGSV stub (proxy object) per client. This dGSV stub contains multiple RMI stubs (as many as replicas exist). The selection of the replica server which should be used for the operation execution is done in the invocation handler. In principle, i.e. for correctness reasons, an arbitrary replica server may be selected. However, since a called method is always executed at the selected replica server, the potentially various clients should try to fairly balance the load. Suitable criteria for load balancing selections are, for example, 1) average response time, 2) mean availability or 3) communication costs. When a particular replication server is selected by the invocation handler then the normalized method call is sent to it by the RMI stub. This is always done by the same interface as the method call is normalized. This relieves of the burden to generate special RMI stubs and RMI skeletons for each server class anew. The dGSV skeleton which has received the method is responsible for further processing. In the course of this processing, the first step is replica management. This is described next.

3.3 Replica Management

For the management of the replicas, the selected replica server acts as coordinator. As described in Section 2.2, the coordinator tries to collect a valid write quorum according to the current voting structure. If this is possible and such a set of replicas could be write-locked then the coordinator determines the maximal version number among them. Note that write-locks imply read and write access rights and write quorums imply read quorums. Subsequently, the coordinator uses the data of a replica exhibiting the maximal version number as the actual state of the object whose method has been called by the client. Then, the method is called on the now available actual local state and the impact of the method invocation wrt. to the object state can be determined. In order to guarantee 1SR, the newly determined object state must be stored with a new, increased maximal version number at all write-locked replicas. Only then can the result be sent back to the calling client. If the object state could not be modified at all participating (e.g. write-locked) replicas then the operation is doomed to fail. It is important to note that despite of the initial reading of the data, it is necessary to use write-locks during the whole execution because the object state is about to be modified. In particular, it must be "written back to the replicas" and no operation can be allowed to be executed "in between" (i.e. concurrently) without putting 1SR at risk.

3.4 RMI versus dGSV Implementation

The difference between using a simple RMI remote object and a replicated dGSV object as described so far is as follows. The `dgsv` package must additionally be imported by the client and the server class. The server class must be declared as a replicated object instead of a "normal" remote object. This is done by extending `dgsv.DgsvRemoteObject` instead of `UnicastRemoteObject`. To bind the server object as a replica into the naming service, the naming methods of the `dgsv` package must be used. Additionally, the desired replication scheme must be specified. This is done through the use of a particular human-readable *configuration file*. It consist of 1) a description of the dedicated replication scheme to be used (i.e., by a voting structure specified in a formal language) and 2) all replica server site names (extending the specification of the voting structure). Configuration files and configuration file generators are available for several standard and non-standard replication schemes. A suitable configuration file can be identified or generated for a particular scenario by means of an automatic artifact [11]. Thus, even non-experts to replication can exploit the benefits of replication for their own interests quite easily. At the client side, the stub must be obtained by using the naming methods offered by the `dgsv` package and not by the naming methods of the `rmi` package. An additional parameter that specifies the participating hosts is needed. An application-specific generation of RMI stub and RMI skeleton is not necessary.

Additionally installation is quite easy. The server class together with the classes of the `dgsv` package and the configuration file must be installed at all participating sites. Then all server objects must be made available by executing the `dgsv.Naming.rebind()` method in the program. From now on, all a client needs is the `dgsv` package. It includes `dgsv.Naming.lookup()`. This method returns the dGSV stub. Using this stub, methods on the replicated object can conveniently be invoked.

Since a call to the replicated object using dGSV most likely leads to several calls from dGSV to the RMI system, a variety of exceptions can be thrown by RMI at several points. An appropriate handling is as follows. If dGSV tries to contact one replica, all exceptions are caught. In this case, the designated replica is not available and dGSV passes on to contact a different one. If not enough replicas could be contacted then the method call finally fails and the last exception caught is thrown back to the client. If enough replicas could be contacted and a method inside the replicated object throws an exception then this exception is also thrown back to the client. Thus, from the perspective of a client, the conceptional handling of the exceptions with dGSV is the same as using simple RMI.

4 An Example

The system as described has been implemented in Java. In the following, we show the procedure of creating a replicated Java object using dGSV. It is analog to the creation of a Java remote object using RMI. In order to illustrate its easy use, we subsequently present a simple example (obviously somehow motivated by the famous "Hello world" programming example that simply prints out the string "Hello World.") The difference to our example is that a replicated Java object stores the name of a so-called "counterpart." Clients may set the name using the method `setCounterpart()`. Thereafter, clients can get a complete welcome string like "`Hello` $<$ *counterpart* $>$`!`" using the method `sayHello()`. The name of the counterpart is stored until modified. The sample application will subsequently be used to illustrate the very few changes to "bare" RMI that are required to exploit flexible replication offered by the proposed system.

4.1 Remote Interface

The first step when writing an application that uses replicated Java objects is to create a remote interface for the replicated remote object. Figure 4 shows the remote interface for our "Hello" example. This interface must be public (line 2) and must extend the interface `java.rmi.Remote` (line 3). The `java.rmi.Remote` interface is used to identify all remote interfaces. The remote interface describes the methods to be accessed remotely (line 5, 7). All methods must be declared to throw at least `java.rmi.RemoteException` (line 6, 8) to handle failures during an invocation. The replicated object must implement this interface.

```
1: import java.rmi.*;
2: public interface Service
3: extends Remote
4: {
5:   public void setCounterpart(String name)
6:     throws RemoteException;
7:   public String sayHello()
8:     throws RemoteException;
9: }
```

Figure 4. Service.java

4.2 Server Class

As shown in Figure 5, the server class contains the implementation of our service. In order to use dGSV, the `dgsv` package must be imported (line 3), additional to the `rmi` packages (line 1, 2). The class must be a subclass of `dgsv.DgsvRemoteObject` (line 5) to declare the object as a replicated object and must implement our remote interface (line 6). In line 8, the variable to store the name of the counterpart is declared. It is initialized by the constructor in line 12. The two methods `setCounterpart()` (line 14) and `sayHello()` (line19) implement our remote interface. In line 17, the name of the counterpart is stored and in line 22 the "Hello" message is constructed and returned. In our example, we have also included a static `main()` method (line 24) to make a complete application which creates a single instance of our server class (line 28) and registers it with the name "Hello" at the naming service

```
 1: import java.rmi.*;
 2: import java.rmi.server.*;
 3: import dgsv.*;
 4: public class ServerImpl
 5: extends DgsvRemoteObject
 6: implements Service
 7: {
 8:   private String name;
 9:   public ServerImpl()
10:     throws RemoteException
11:   {
12:     name = "";
13:   }
14:   public void setCounterpart(String name)
15:     throws RemoteException
16:   {
17:     this.name = name;
18:   }
19:   public String sayHello()
20:     throws RemoteException
21:   {
22:     return("Hello " + name + "!");
23:   }
24:   public static void main(String[] argv)
25:   {
26:     try
27:     {
28:       ServerImpl server = new ServerImpl();
29:       dgsv.Naming.rebind("Hello", server,
30                    "strategyFileName.dgsv");
31:     }
32:     catch(Exception e)
33:     {
34:       System.out.println("Exception: " +
35                          e.getMessage());
36:       e.printStackTrace();
37:     }
38:   }
39: }
```

Figure 5. ServerImpl.java

(line 29). In order to integrate the dGSV skeleton between the server object and the RMI skeleton and furthermore, to declare the voting strategy which should be used, the use of the `dgsv.Naming.rebind()` method becomes necessary. This method constructs a new instance of a dGSV skeleton which 1) generates the dGSV stub, 2) stores the reference of the server object together with the declared voting strategy, and 3) registers itself at the naming service. Thus, the client object can obtain the dGSV stub from the naming service. The voting strategy is declared by means of a configuration file name (refer to Section 3.4).

4.3 Client Class

As in the server class, the `rmi` package must also be imported in the client class (line 1). So does the `dgsv` package (line 2). From within the client class, the service of the server class is called (line 17, 19). This requires the reception of the dGSV stub through the naming service (line 13, 14). This is done by the method `dgsv.Naming.lookup()` which receives the RMI stub from the RMI naming service. This RMI stub is a stub for

```
 1: import java.rmi.*;
 2: import dgsv.*;
 3: public class Client
 4: {
 5:   public static void main(String[] argv)
 6:   {
 7:     try
 8:     {
 9:       String[] hostname = new String[3];
10:       hostname[0] = "host1";
11:       hostname[1] = "host2";
12:       hostname[2] = "host3";
13:       Service service = (Service)
14        dgsv.Naming.lookup("Hello", hostname);
15:       if(argv.length > 0)
16:       {
17:         service.setCounterpart(argv[0]);
18:       }
19:       System.out.println(service.sayHello());
20:     }
21:     catch(Exception e)
22:     {
23:       System.out.println("Exception: " +
24                            e.getMessage());
25:       e.printStackTrace();
26:     }
27:   }
28: }
```

Figure 6. Client.java

communicating with the registered dGSV skeleton. Now, the dGSV stub (which was generated by the dGSV skeleton) can be received from the dGSV skeleton and is returned to the client object. As this dGSV stub can be received from any replica server, an appropriate data structure (*sic*: array) holding the names of all possible replica servers must be created (line 9-12). As a result, the lookup method is able to request any replica server if necessary (and available). After having received the dGSV stub, the service method is called. In line 14, it is checked whether a command line parameter is given. If affirmative then the counterpart is set to the server object (line 17). Subsequently the “Hello” message is received from the server object. This concludes the example.

5 Related Work

In [18], Wang and Zhou show two implementation approaches of primary-backup replication in Java. By using the primary-backup protocol, a client sends requests only to the primary. The primary propagates each request that leads to an object state change to the backup. The backup processes the changes and returns an acknowledgment to the primary. Finally, the primary replies to the client. If a client detects the failure of the primary then the backup is used instead.

Their first implementation approach is based on remote method invocation. Three interfaces must be defined. Firstly, a service interface which provides the methods of the service to be replicated. Secondly, a primary-backup interface with methods for the communication between primary and backup. And thirdly, a replica interface that is a combination of the former two interfaces. All communication is done through these interfaces via RMI.

The second approach is based on networking packages. The object that provides the service does not know anything about replication. This is done by the replica-proxy objects which communicates using the networking packages `java.net`. However, both approaches are less flexible in the sense that they do not support a change of adopted replication schemes.

In [19], a fault tolerance development framework based on replication is presented. The approach uses a primary object in order to provide service and a backup object that may take over in case the primary object fails. The reference of the backup object is maintained independently by three so-called replication managers. If a failure occurs then clients can get the reference of the backup object from one of these replication managers. For communication, RMI is used. As above, the approach is also fixed with respect to the replication strategy and therefore more than limited wrt. replication scheme change.

The Aroma system [9] enhances the RMI system for group communication and provides a strong consistency notion among multiple replicas by using an underlying multicast protocol. This is done by modifying both, the application and the RMI infrastructure. As a consequence, a method invocation to a server object can be intercepted and distributed to all replicas. The Aroma system supports active and passive replication but only a fixed replication scheme.

Another approach is given in [7]. RepliXa is a framework for transparent and adaptable object replication in Java. To bind a replication scheme to an object, RepliXa uses the special Java interpreter metaXa which offers behavioral reflection. Using metaXa , it is possible to bind so-called meta objects to classes, objects and references. Methods of these meta objects can be called if a previously defined event (i.e. a method call of an application object) has been caught. The meta objects manage replication. Since only few changes to existing programs are necessary, RepliXa reaches a high level of transparency. A major disadvantage is that the framework requires the special Java interpreter metaXa.

All of these approaches assume fixed replication schemes and are consequently less flexible wrt. replication scheme changes. If it is conceptionally possible to change the adopted replication scheme at all then the scheme must be explicitly programmed. This is in most cases very time-consuming and extremely error-prone. Furthermore, it requires expert knowledge about replication schemes. The dGSV framework is the only approach that enables the convenient use of a large number of dynamic replication schemes. The different schemes can simply be provided by a catalog consisting of scheme-specific characteristics, like run-time costs and operation availability with respect to a particular setting and a simple configuration file. Thus, replication non-experts can easily exploit the benefits of replication for their own interests. Furthermore, dGSV itself is able to automatically and dynamically modify the configuration of a selected replication scheme as failure patterns dictate at run-time. This is done in a general manner, i.e., without any scheme-specific knowledge, and thus, without the need of human intervention.

6 Conclusion and Future Work

In this paper, we showed the integration of the *Dynamic General Structured Voting (dGSV)* framework into Java and its convenient use. The framework automatically interprets the specification of a replication scheme and mimics its behavior. Furthermore, it is able to automatically and dynamically modify the configuration of the replication scheme. The use of this framework leads to highly available Java objects and methods on these objects by exploiting quite complex fault tolerance techniques. Thus, the dependability of a Java application can be highly improved. The replication schemes used for this purpose can be easily be configured and may be changed at almost any time. This allows to flexibly react to changing application requirements. Additionally, we demonstrated the nearly transparent use of dynamic replication techniques in existing Java programs.

With regards to the performance issues, it is important to carefully choose the object which should be replicated and its methods. For example, an object representing a large bitmap together with a method for setting a particular point within the bitmap is – in most cases – not a suitable candidate for replication, since at each operation execution the complete bitmap must be loaded from one replica, modified and subsequently be stored at multiple replicas. The overhead would be quite large and performance would suffer. In contrast, using the very same object but with methods for complex bitmap transformation will behave more efficiently. Thus, the relationship between the size of the object and the complexity of its methods must be well balanced. This may not be possible for every application, but on the contrary, not all applications require a high degree of fault tolerance. For those ones which does, we believe that replication in the manner described in this paper is a good choice.

We have recently implemented a first prototype of dGSV in Java. In ongoing research, we increase the robustness of the prototype implementation. In the near future, performance measurements derived from fully-grown Java applications will be performed. We plan to report these results as soon as possible.

References

[1] D. Agrawal and A. E. Abbadi. The tree quorum protocol: An efficient approach for managing replicated data. In *Proc. of the 16th VLDB Conference*, pages 243–254, 1990.

[2] M. Ahamad and M. H. Ammar. Multidimensional voting. *ACM Transactions on Computer Systems*, 9(4):398–431, 1991.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery*. Addison Wesley, 1987. ISBN 0-201-10715-5.

[4] S. Y. Cheung, M. Ahamad, and M. H. Ammar. The grid protocol: A high performance scheme for maintaining replicated data. In *Proc. of the 6th International Conference on Data Engineering*, pages 438–445, February 1990.

[5] B. Freisleben, H.-H. Koch, and O. Theel. Designing multilevel quorum schemes for highly replicated data. In *Proc. of the 1991 Pacific Rim Intern. Symp. on Fault Tolerant Systems, Kawasaki, Japan*, pages 154–159. IEEE, 1991.

[6] H. Garcia-Molina and D. Barbara. How to Assign Votes in a Distr. System. *Journal of the ACM*, 32(4):841–860, 1985.

[7] M. Jäger, M. Golm, and J. Kleinöder. Replixa - Ein Framework zur transparenten und anpassbaren Objektreplikation (in german). Technical Report TR-I4-99-11, University of Erlangen-Nürnberg, 1999.

[8] S. J. Mullender and P. M. B. Vitányi. Distributed match–making. *Algorithmica*, 3:367–391, 1988.

[9] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Transparent consistent replication of java rmi objects. In *Proc. of the International Symposium on Distributed Objects and Applications*, 2000.

[10] M. Rabinovich and E. D. Lazowska. Improving fault tolerance and supporting partial writes in structured coterie protocols. In *Proc. of the ACM SIGMOD*, pages 226–235, July 1992.

[11] H. Schinzel. genGSV: Ein Werkzeug zur evolutionären, automatischen Synthese von Replikationsstrategien (in German). Master's thesis, Dept. of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, 2002.

[12] Sun Microsystems, Inc. *Java 2 Platform, Standard Edition, v1.3.1, API Specification.*

[13] Sun Microsystems, Inc. *Java Remote Method Invocation Specification.*

[14] O. Theel. General structured voting: A flexible framework for modeling cooperations. In *Proc. of the 13th International Conference on Distributed Computing Systems, Pittsburgh, PA*, pages 227–236. IEEE, May 1993.

[15] O. Theel and H. Pagnia-Koch. General design of grid-based data replication schemes using graphs and a few rules. In *Proc. of the 15th International Conference on Distributed Computing Systems*, pages 395–403. IEEE, May 1995.

[16] O. Theel and T. Strauß. Automatic Generation of Dynamic Coterie-based Replication Schemes. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), Las Vegas, NV, U.S.A.*, pages 1606–1613, July 1998.

[17] O. Theel and T. Strauß. Automatic generation of dynamic coterie-based replication schemes. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada*, July 1998.

[18] L. Wang and W. Zhou. Primary-backup object replications in java. In *Proc. of the Technology of Object-Oriented Languages and Systems*. IEEE, 1998.

[19] X. Zhao, D. Chen, and L. Xie. An object-oriented developing framework of fault-tolerant system. In *Proc. of the 31st International Conference on Technologies of Object-Oriented Language and Systems*, 1998.