Transparent and Adaptable Object Replication Using a Reflective Java

Jürgen Kleinöder, Michael Golm

September 1996 TR-I4-96-07

Technical Report

Computer Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University Erlangen-Nürnberg, Germany

Transparent and Adaptable Object Replication Using a Reflective Java

Jürgen Kleinöder[1], Michael Golm
University of Erlangen-Nürnberg, Dept. of Computer Science IV
Martensstr. 1, D-91058 Erlangen, Germany
{kleinoeder, golm}@informatik.uni-erlangen.de

Abstract

Adaptability to special requirements of applications is a crucial concern of modern operating system[2] architectures. Reflection and meta objects are means to achieve this adaptability. This paper describes a reflective architecture and the realization of different object-replication techniques within this architecture.

The reflective architecture is an extension of the Java run-time environment. We show how reflection can be used to implement replication that is transparent to the replicated object and its clients. The architecture allows a fine-grained adaptability of the replication protocols to special object requirements. The problems of replication in object-oriented systems and their solutions are discussed.

1 Introduction

Replication of objects becomes an important issue when object-oriented systems meet distributed systems. In this paper we discuss techniques of implementing replicated objects using *MetaJava*, a reflective programming system based on Java. Our run-time reflective architecture enables the transparent use of replicated objects and user customizability of replication protocols. Customizability is realized at a very fine-grained level. The replication mechanism of each object can be adapted to the special requirements and usage patterns of that object.

We do not discuss security issues in this paper. A description of implementing security with reflection can be found in [Rie96]. We also do not consider migration of objects, although this could be a valuable extension to our system.

The contribution of this paper is to show how existing replication techniques can be combined with a reflective system to enable replication that is transparent to the programmer and to the user of the replicated object and that can be adapted to application requirements and to properties of specific environments.

The paper is organized as follows. Section 2 describes replication concepts for object-oriented systems. Section 3 introduces reflection and metaprogramming. Section 4 describes the computational model of MetaJava. Section 5 explains how replication can be implemented transparent to the application program using the reflective features of MetaJava. Problems of ensuring this transparency are discussed in Section 6. The trade-offs between the solutions are explained and it becomes apparent that only a system that allows different coexisting replication strategies can satisfy all users' needs. Section 7 describes our implementation of the transparent and adaptable object replication system with MetaJava. We conclude with a comparison with related work and a short note about the current project status and future work in Sections 8 and 9.

2 Replication in object-oriented systems

The benefits of replication are twofold. In fault-tolerant computing systems replication is used to increase availability and reliability of substantial system components. In other distributed systems replication is used to mask communication latency when accessing remote objects and thus to raise performance.

The increasing use of object-oriented languages for software development requires a new look at replicated systems because objects are more complex entities than just plain data.

Two categories of object-replication techniques can be distinguished: the primary-backup and the state-machine approach.

Primary-backup approach. In primary-backup replication [BMS+93], also called passive replication, methods are executed at one node — the primary. The state of the primary is mirrored to the backup replicas at certain points (checkpointing). A usual checkpoint is the end of a method execution. This approach has some drawbacks. Electing a new primary after failure is difficult and time consuming,

1. This work is supported by the *Deutsche Forschungsgemeinschaft DFG* Grant *Sonderforschungsbereich SFB 182*, Project *B2*.

2. When we talk about operating systems we include system libraries, run-time systems, and virtual machines.

and may even be not tolerable for real-time applications. Between two checkpoints the state of the backups is not coherent with the state of the primary.

State-machine approach. The state-machine approach (active replication) [Sch90], executes the method at all replicas and thus avoids the single point of failure. Nevertheless this approach also has some problems. All methods of the replicas must be deterministic. Hence they must not use random functions or node specific properties, such as input devices (sensors, real time clocks). Another disadvantage of active replication is the replication of computation and thus the waste of computational resources. As described in [Sch90] the transition from one state to the next is an atomic operation. The object is used as a monitor, which limits useful and sometimes necessary concurrency.

3 Reflection and Metaprogramming

In the past programs had to fulfill a task in a limited computational domain. Today's applications have become more complex: they demand support for multithreading (synchronization, deadlock detection, etc.), distribution, fault tolerance, mobile objects, extended transaction models, persistence, and so on. These demands make it necessary for an application program to observe and adjust its own behavior. Many ad hoc extensions to languages and run-time systems have been implemented to support particular features such as persistence. Reflection is a fundamental concept for a uniform implementation of all these different demands.

Smith's work on 3-LISP [Smi82] was the first that considered reflection as essential part of the programming model. Maes [Mae87] studied reflection in object oriented systems. According to Maes *reflection* is the capability of a computational system to "reason about and act upon itself" and adjust itself to changing conditions. *Metaprogramming* separates functional from non-functional code. *Functional code* is concerned with computations about the application's domain (*base level*), non-functional code resides at the *meta level*, supervising the execution of the functional code. To enable this supervision, some aspects of the base-level computation must be reified. *Reification* is the process of making something explicit that is normally not part of the language or programming model.

As pointed out in [Fer89] there are two types of reflection: structural and behavioral reflection (in [Fer89] termed computational reflection). Structural reflection reifies structural aspects of a program, such as inheritance and data types.
A common technique for structural reflection is to let a meta class control a number of base classes. Run-Time Type Identification (RTTI) of C++ [Str92] is an example of structural reflection. Behavioral reflection is concerned with the reification of computations and their behavior. Our architecture deals with behavioral reflection at run time. In [KlG96] we show how a run-time meta architecture can be built that is nearly as efficient as one that operates only at compile time. A programming environment that incorporates a meta architecture gains the following advantages:

Increased productivity. Like object-oriented programming, metaprogramming is a new paradigm which leads to more structured and easily maintainable programs.

Separation of concerns. In conventional programming the application program is mixed with and complicated by policy algorithms. This makes it difficult to understand, maintain, debug, and validate the program.

Separation of the reflective from the base algorithm makes reusability of policies feasible [HüL95], [Kic96b]. The use of a separate meta space allows the application programmer to focus on the application domain. The additional functionality is supplied as a library of meta components or is developed together with the application program. We regard reflective decomposition as a new structuring technique in addition to functional and object-oriented decomposition.

Configurability. The meta level establishes an open system architecture [Kic96a]. New policies can be implemented without changing the application code. This is especially useful for class libraries, where the library designer can only guess the demands of the library user.

Not only application developers can profit from metaprogramming but also application users may replace meta components to tailor the application to their particular needs. Typically, a system administrator will tailor the meta space, so that the application can take care of local resources such as processors, printers, network connections, or hard disks.

Transparency. Transparency and orthogonality of base system and meta system are desirable features but cannot always be guaranteed in real applications. Consider a meta object that implements a bounded buffer synchronization scheme. It does not make sense to attach it to a base object that does not have a bounded buffer semantics (i.e., two operations *put* and *get* to access the buffer).

4 Computational model of MetaJava

Traditional systems consist of an operating system and, on top of it, a program which exploits the OS services using an application programmer interface (API).

Our reflective approach is different. The system consists of the OS, the application program (the *base system*), and the *meta system*. The program may not be aware of the meta system. The computation in the base system raises events (see Figure 1). These events are delivered to the meta sys-

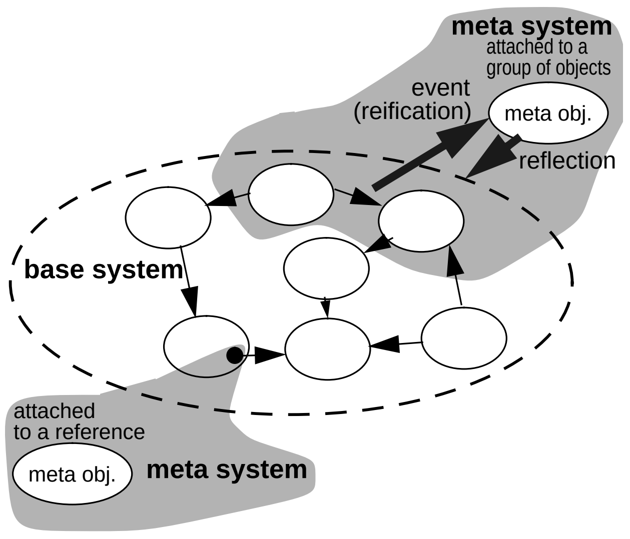


Figure 1: Computational model of behavioral reflection

tem. The meta system evaluates the events and reacts in a specific manner. All events are handled synchronously. Base-level computation is suspended while the meta object processes the event. This gives the meta level complete control over the activity in the base system. For instance, if the meta object receives a *method-enter* event, the default behavior would be to execute the method. But the meta object could also synchronize the method execution with another method of the base object. Other alternatives would be to queue the method for delayed execution and return to the caller immediately, or to execute the method on a different host. What actually happens depends entirely on the meta object used. The replication-relevant events are listed in Figure 2.

method-enter(object,method,arguments)
: method was called at object with arguments.

aquire-object-lock (object)
: Aquire the lock of object.

release-object-lock(object)
: Release the lock of object.

read-field (object, field)
: Read the field of object.

write-field (object,field, value)
: Write value into the field of object.

Figure 2: Events generated by base computation

A base object also can invoke a method of the meta object directly. This is called explicit meta interaction and is used to control the meta level from the base level.

Not every object must have a meta object attached to it. Meta objects may be attached dynamically to base objects at run time. This is especially important if a distributed computation is controlled at the meta level and arbitrary method arguments need to be made reflective. As long as no meta objects are attached to an application, our meta architecture does not cause any overhead. So applications only have to pay for the meta-system functionality where they really need it.

void ***attachObject*** (MetaObject meta, Object base)
: Bind a meta object to a base object.

MetaObject ***findMeta*** (Object base)
: Find the responsible meta object for a base object.

Object ***continueExecutionObject*** (EventMethodCall event)
: Continue the execution of a base-level method. This calls the non-reflective method. No event is generated, otherwise the reflection would not terminate.

Object ***doExecuteObject*** (EventMethodCall event)
: Execute a method. Contrary to the previous method, this one calls the method as if it were called by an ordinary base object.

Object ***createNewInstance*** (EventObjectCreation event)
: Create a new instance of a class. The class name is passed as String (as part of the event parameter).

Object ***cloneReference*** (Object ref)
: Create a new reference which points to the same object as object ref. This method is used to attach meta objects to references.

Class ***cloneClass*** (Object ref)
: Clone the class of the object given. The clone becomes the class of this object. This method is used when a meta object is attached to an object to avoid interference with other instances of the same class.

void ***installBytecode*** (Object ref, String method, byte code[])
: Installs code as new method bytecode. Together with addConstantPoolItem this method is used to generate a stub method in the place of the original method.

String ***retrieveObjectLayout*** (Object ref)
: Returns the types of all fields of object ref. This method is used together with getFieldObject, getFieldInt, getFieldFloat, setFieldObject, etc., to access fields of arbitrary objects.

Object ***getFieldObject*** (Object ref, String fieldName)
: Returns the contents of field fieldName of object ref. Name and type of all fields (object layout) can be retrieved with retrieveObjectLayout.

void ***setFieldObject*** (Object ref, String fieldName, Object obj)
: Sets the contents of field fieldName of object ref to object obj.

int ***addConstantPoolItem*** (Class c, CPItem i)
: Adds an item to the constant pool of class c.

Figure 3: Selected methods of the meta interface of the MetaJava virtual machine

Currently meta objects can be attached to references, objects, and classes. If a meta object is attached to an object, the semantics of the object is changed. Sometimes it is desirable only to change the semantics of one reference to the object — for example, when tracing accesses to the reference, or when attaching a certain security policy to the reference [Rie96]. Attaching a meta object to a class makes all instances of the class reflective.

To fulfill its tasks the meta object has access to a set of methods which can manipulate the internal state of the virtual machine. These methods are called the *meta-level interface* (MLI) of the virtual machine[3]. Only the MetaObject class, its subclasses, and members of the package meta can invoke these methods. A list of the most important methods of the MLI is given in Figure 3. The MLI methods continueExecutionObject and doExecuteObject are also available for the return types int, float, and void. The methods getFieldObject and setFieldObject are also available for the field types int and float. All methods of the MLI are implemented as native methods and dynamically linked to Sun's Java interpreter as shared library.

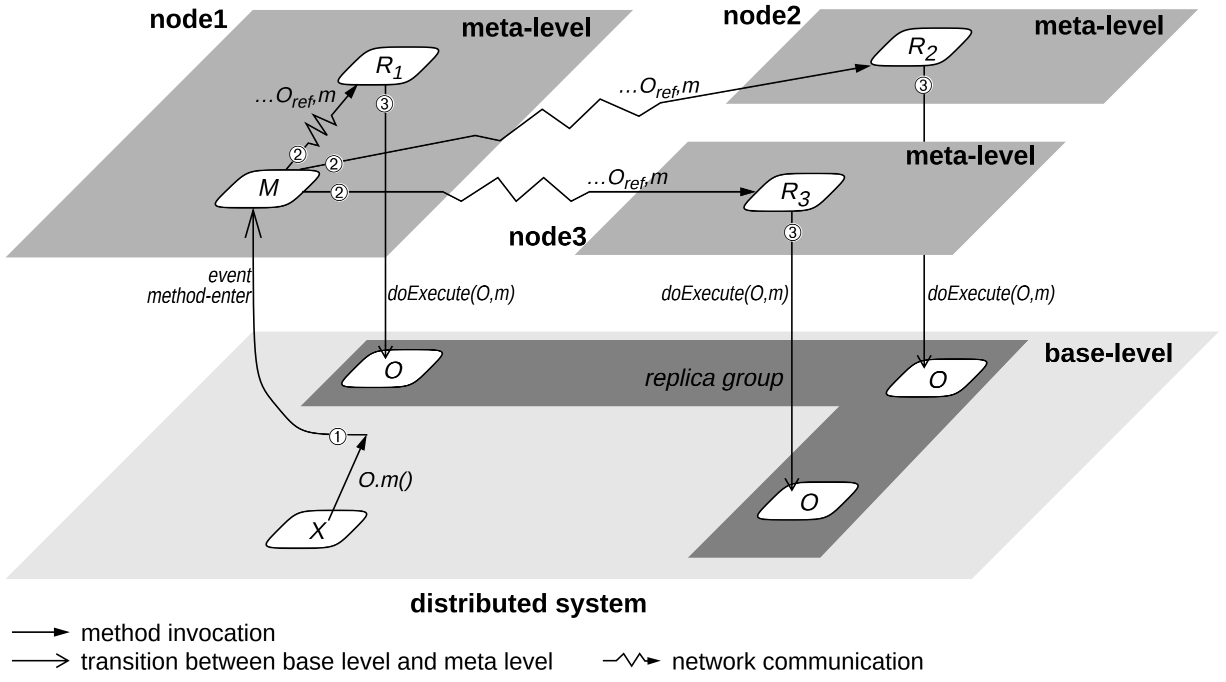
3. Architecture and terminology of the Java VM are described in detail in [LiY96]

Above the MLI lies the event generation layer — a set of methods inherited by every meta object from the base class MetaObject. It is completely written in Java using the MLI for structural reflection. The event generation layer provides the abstraction of a base computation that is generating events as described in this section.

5 Transparent replication using MetaJava

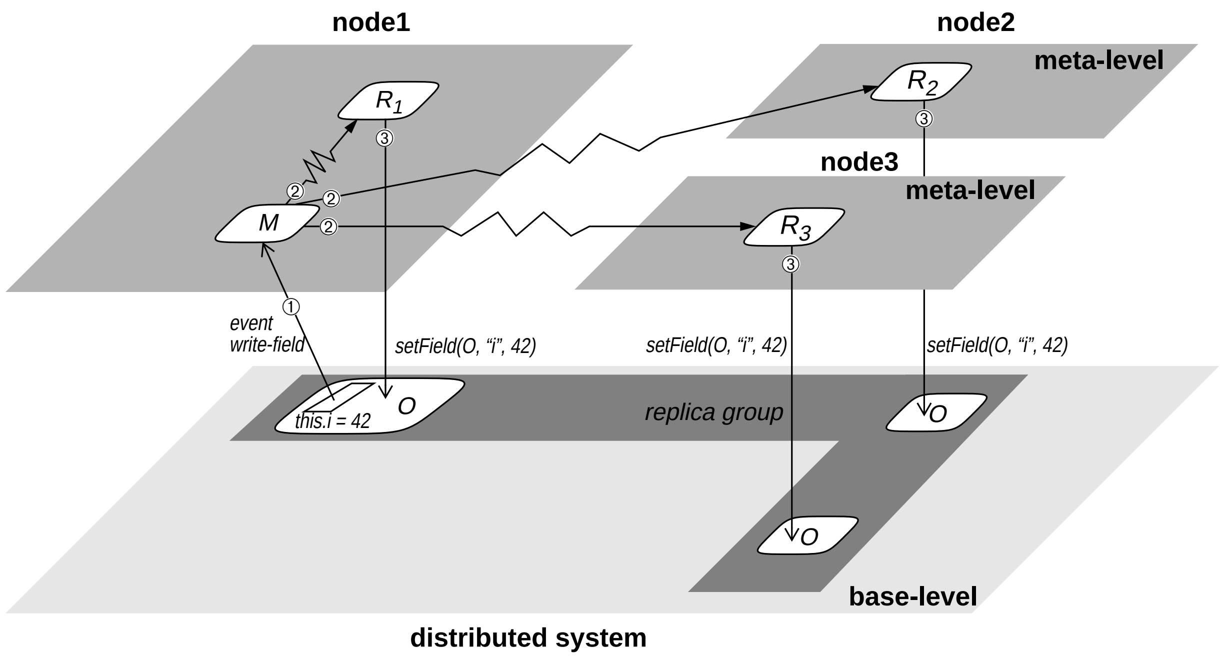
All replicas belonging to one replicated object are called a *replica group*. We use a slightly generalized definition of active and passive replication than introduced in Section 2. The term *active replication* is used for every protocol that duplicates base-level threads between nodes. Every protocol that replicates the state of an object and ensures a certain consistency of the replicated state is classified as *passive replication*. Thus active replication protocols are concerned with method executions while passive replication protocols deal with read/write operations on the object state and give guarantees about their order.

A scenario of active replication is sketched in Figure 4. It shows the involved objects at base and meta level and their interaction when a method at a replicated object is



Object X calls method m of the replicated object O. This base-level action is implemented at the meta level as follows: ① Meta object M is attached to O and thus it receives the *method-enter* event. ② M requests all replication-protocol meta objects ($R_{1..3}$) of the replica group to execute method m at object O. ③ $R_{1..3}$ call the base-level object O using the meta-interface method doExecute( ). The return value of method m is given back to $R_{1..3}$ as return value of doExecute( ). $R_{1..3}$ deliver this value back to M, which collects all results and returns one to X.

Figure 4: Active replication: Method invocation at replicated object



A method of object *O* is executed and tries to write field (instance variable) *i* . This base-level action is implemented at the meta level as follows: ① Meta object *M* is attached to *O* and thus it receives the *write-field*-event. ② *M* requests all replication-protocol meta objects of the replica group to set field *i* of object *O* to the new value. ③ $R_{1..3}$ set the field to the new value using the meta-interface method setField( ).

Figure 5: Passive replication: Write access at replicated object

invoked. All method invocations at *O* are reified by *O*'s *replication-control meta object M*. The *replication-protocol meta objects* $R_{1..3}$ together with *M* are responsible for a consistent execution of the replicated methods.

A possible implementation of passive replication is shown in Figure 5. The replication-control meta object *M* reifies the access to all *fields* (instance variables) of the replicated base-level object *O*. The replication-protocol meta objects $R_{1..3}$ are responsible for a consistent state of the replicated object.

We use the term *replication meta objects* for all replication-control and replication-protocol meta objects together.

6 Problems of transparency and adaptability

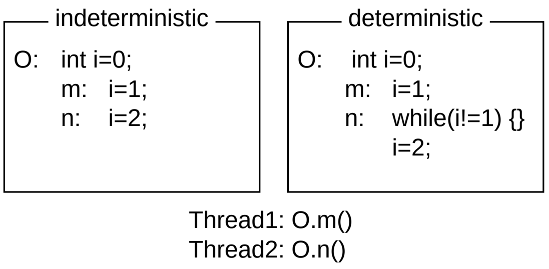
The previous section described how active and passive replication can be implemented in MetaJava, so that it is transparent to the base-level program. Related work in replicated RPC and distributed shared memory (DSM) identified several problems when replicating computation or data. In this section we discuss those problems and their solutions in our architecture. Contrary to other existing replication systems, it is possible to use different replication protocols transparently to the base level. It is even possible for an application programmer to use own object-specific protocols.

The binding between meta objects and base-level objects can be described either in a separate configuration or with statements in the application program. Since the first approach entails a lot of unsolved problems outside the focus of this work, we used the second approach for our prototype implementation although transparency is lost at the points where the binding of meta objects is established.

To simplify the following discussion we call all remote objects replicas, even if there exists only a single incarnation of the object.

6.1 Active replication

Problems with indeterminism. The notion of determinism enforced by the state-machine approach is very restrictive. Considering replicated objects as state machines would make replication non-transparent to the Java programmer. The programming model of Java is the time-shared uniprocessor; we assume preemptive, time-sliced scheduling. We require programs to be deterministic in this environment. Figure 6 shows two versions of an object O, both containing an instance variable i and two methods m and n. Both methods are invoked concurrently, method m is called in thread 1 and method n is called in thread 2. The value of i is 1 or 2 in the left definition and always 2 in the right definition of object O. The deterministic behavior is achieved without



```
O:   int i=0;
     m:   i=1;
     n:   i=2;
```

```
O:   int i=0;
     m:   i=1;
     n:   while(i!=1) {}
          i=2;
```

```
Thread1: O.m()
Thread2: O.n()
```

Figure 6: Example for indeterministic and deterministic executions

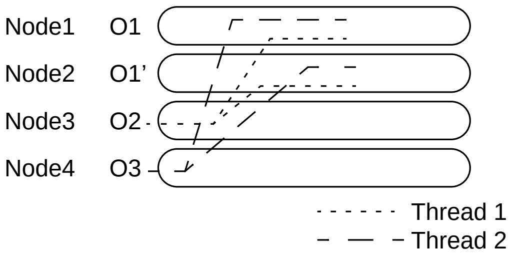


Figure 7: Two replicas with different method execution orders

using special synchronization constructs, such as mutexes or barriers. This behavior of O must remain the same when O is being replicated. Object replication systems that treat method executions as atomic events (such as ORCA [BBH+96]) and thus transform every object into a monitor are not able to replicate the deterministic object of Figure 6. In MetaJava this is not an issue, because the execution of normal Java methods is not atomic. The programmer has to synchronize explicitly where it is necessary. The replication meta objects have to handle the synchronization events (see Figure 2).

The message order problem. Method invocations may be executed in any sequential order that respects the specification of the program. However, if replication has to be transparent, the invocation order has to be the same at each replica of a replica group.

To achieve this order, some ordered atomic broadcast protocol has to be used [JoB89][Bir85][Coo85][BSS91]. These protocols are commonly classified as centralized, token based, and distributed protocols. In the centralized and token based protocols the message order at one designated replica is used as global order (case 3 of Figure 8). In MetaJava one of the replication-protocol meta objects ($R_{1..3}$ in Figure 4) can be used to determine this order. In the distributed protocol (case 1 of Figure 8), the client agrees with all replicas upon a unique sequence number for the invocation. To achieve this in MetaJava the replication-protocol meta objects have to commit upon a global order of method invocations.

The centralized protocol introduces a single point of failure but limits network contention. It is used in [YJT88] for the implementation of replicated RPC from a group of m replicas to a group of n replicas. The designated replica receives m calls for one invocation and distributes the invocation to the (n-1) follower replicas and thus reduces the number of messages from m*n to m+n-1.

The call duplication problem. If a *n*-replicated object calls a different replica group, the callee receives *n* invocation requests (case 1 and marked object in case 3 of Figure 8) but must execute only one invocation.

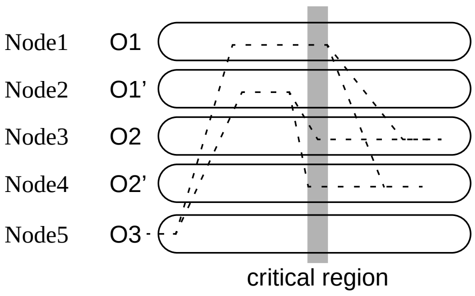


Figure 9: Call duplication problem

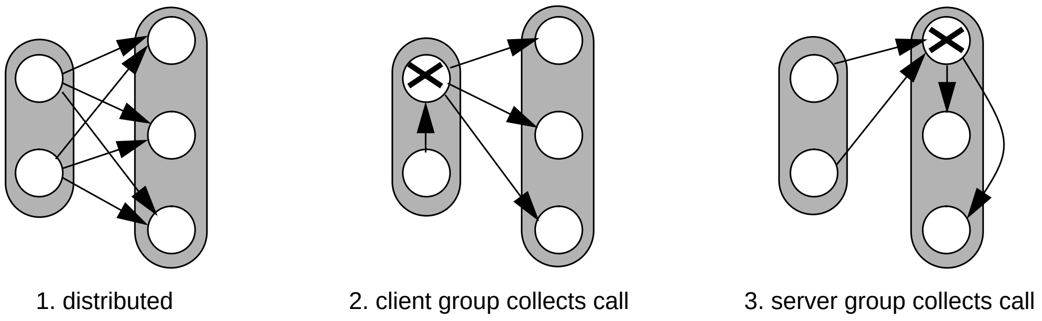


Figure 8: Communication structures when calling from one replica group to the other

This conflict can be resolved at client side (replicas O1 *in Figure 9*, *case 2 of Figure 8*) or at server side (replicas O2 *in Figure 9*). If the server does the work, it has the choice to invoke the method after receiving the first request or to wait for the last one (the server must be able to identify invocations that belong to one group, this can be done using a global thread ID). Cooper [Coo85] distinguishes first-come (using the first request) and unanimous (waiting for the last request) execution of a many-to-one or many-to-many call. Waiting for the last request limits parallelism. The slowest replica dictates the speed of the whole computation.

```
O1: int i=0;                    O2:
    m:   i=1; O2.m();                m:   O1.n();
    n:   i=2;


                Thread1: O1.m()
```

Figure 10:Program fragment that emphasizes the problems of first-come

Figure 10 illustrates the problems of the plain first-come strategy. The final state of i is 2. The state of i in the slower replica can be 1 if n is called before m sets i to 1. This can happen if the execution of O2.m() is started in the critical region of Figure 9 and the invocation request O1.n() is received by O1 at Node1 within the critical region. A protocol implementing first come has to ensure that replicated calculations do not execute concurrently on the same node.

In a revised version of GARF [MaG95b] the call of only one replica of a caller group is transmitted to the callees and only one of the results of the callee group is sent back. Filter objects decide which call and which result are transmitted. This scheme is termed *prefiltering*.

How to solve the call duplication problems depends on the protocol for method invocations from one replica group to the other and the way of solving the message order problem. The described protocols can be implemented by the replication meta objects. Furthermore it is possible to select the optimal protocol on a per-method basis if the programmer provides additional semantic information about the methods of the base object when binding the replication-control meta object. For example first-come is the optimal protocol for all methods which do not invoke further methods.

The object duplication problem. Objects that are created by methods of a replicated object are automatically replicated, because they are created by every replica. A reference to such an object can be propagated to a different replica group as parameter of a method invocation. Depending on the protocol to handle call duplications the callee receives either one call with one reference (case 2 of Figure 8) or multiple calls with different references R_i. In the second case the replication-protocol meta object installs one local proxy for all R_i on the callee side. This proxy is controlled by a meta object X. Instead of the references R_i the callee gets a reference to the proxy. All invocations using this reference are caught by the meta object X and redirected to the references R_i.

Using semantic information. The replication meta objects control one replicated base object. They can use programmer-supplied information about the base object's semantics to optimize the protocol. In our implementation of an active replication protocol the programmer can, for example, name read-only (idempotent) methods when establishing the binding between base and meta object. Such methods may be performed at only one replica, for example the local replica.

Choosing the protocol. A centralized protocol limits network contention and thus it may improve communication performance. On the other hand it introduces a single point of failure and possible performance bottleneck which in turn could limit scalability.

The choice of the protocol depends also on the intended use of the replicated object and the available hardware: What hardware properties can the protocol rely on (e.g. FIFO network, atomic hardware broadcast)? What requirements has the user, (e.g. fault-tolerance, high performance with no fault-tolerance, real-time guarantees)? If fault-tolerance is required, which failure model matches closest with reality (e.g. fail-stop or byzantine failures, network-partitions)? How is the "sharing pattern" of the object in the specific application? Must the protocol be non-blocking in case of node failure?

6.2 Passive replication

When implementing a passive replication protocol one must consider two problems. Objects contain references to other objects. These references are only valid in the address space of the same virtual machine. The second problem is the bad performance of DSM systems compared to message passing systems.

References in a replicated object. DSM systems use identical placement of pages in the virtual address space of the different processes. This requires a homogeneous hardware architecture. Zhou et. al. [ZSL+92] describe a heterogeneous page-based DSM system and the problems of different representations of basic data types, like string, int, and float. Pointers to items on the same page are converted by adding the difference between the virtual start addresses of the pages. They do not consider pointers to other, possibly non-replicated, data.

It is possible to solve the reference problem by maintaining a bijective mapping of local references and global identifiers at every node. References are translated using this mapping when they are copied between replicas. We have not implemented this solution yet and thus currently exclude objects that contain references from passive replication.

Performance problems. The unit of sharing in most DSM systems is one page. This large granularity leads to the problem of false sharing — independent modifications in different sections of a page. Another reason for bad performance are unnecessarily strong consistency models.

In our replication architecture the unit of sharing is one object and consistency can be implemented on a per-field basis. Thus false sharing is eliminated. The adaptability in the MetaJava architecture allows a flexible choice of the appropriate consistency model and consistency-preserving protocol for each object or even for each field of each object.

Memory models. In non-distributed memory a read of a memory cell returns the value of the most recent write to the cell (strict consistency). This behavior is not possible in distributed memory because of the limited speed of information propagation in distributed systems. Sequential consistency [Lam79] preserves the illusion of non-distributed, consistent memory without referring to the notion of global time. Unfortunately consistency can be achieved only at the cost of bad performance. Lipton and Sandberg [LiS88] showed that a replicated system cannot be consistent, fast, and scalable at the same time. Adaptability enables the programmer to decide on a per-object basis which properties are mostly needed for the object. If the object's semantics allows a non-coherent memory model, performance may profit. There has been a lot of work done in developing weak consistency models and investigating which classes of algorithms do tolerate them. Maya [ACL+93] is a simulation platform to demonstrate the performance benefits of weak consistency models.

Selecting a protocol. A memory model is merely a contract between the programmer and the memory system. How the memory system fulfills this contract, that is, what protocol (write-invalidate, write-update, write-shared, etc.) it uses to ensure a certain consistency, may vary.

7 Implementation

7.1 Distributed Virtual Machine

Our distributed system consists of a number of Java virtual machines (JVM) running on several hosts, which are connected through a network. We currently use UDP for communication. A *port-server* object distributes incoming messages to the appropriate replication-protocol meta objects or to the installation server. The messages are marked with a protocol ID and an instance ID of the replication-protocol meta object. The *installation-server* object is used to install the base-level replicas and their replication-protocol meta objects.

7.2 Messages

The first 32 bits of a message contain the protocol ID. At the time of this writing there are three protocol IDs:

- PROT_INSTALL: This protocol is used for the installation and deinstallation of replicated objects and their meta objects.
- PROT_ACTIVE_CENTRAL: Active replication, described in Section 7.5.
- PROT_PASSIVE_SEQUENTIAL: Passive replication, described in Section 7.6.

The remaining message data is protocol dependent.

7.3 Replica installation

Replica installation is done by attaching a replication-control meta object to an ordinary base object (see Figure 11). The meta object sends an installation request to the installation servers at all nodes where a replica has to be installed. A unique transaction sequence number is used to identify replies to an installation request. The installation server understands three different requests:

```
installObject("classname"," protocolname") → reference
installWithName("classname", "symname", "protocolname")
                      → reference
getReferenceFromName("symname") → reference
```

Call-by-replication. If an object is passed as argument to a method of a replicated object, the argument object can be replicated at the callee hosts, or just passed as remote reference. The replication-control meta object of the invoked replica controls this. The programmer can reconfigure the attached replication-control meta object, for instance before calling a method of the base object, to change the parameter passing semantics of a single method invocation.

```
import meta.replication.*;

class Database {
    private String names[ ], values[ ];
    public void add(String name, String value) {
        ...   // insert tuple in arrays
    }
    public String query(String name) {
        ... // search for name and return value
    }
}
class Client {
    public void go (Database db) {
        db.add("USA", "Washington");
        db.add("France", "Paris");
        System.out.println("Capital of France: " + db.query("France"));
    }
}
```

```
class Main {
    public static void main (String args[ ]) {
        Database db = new Database( );

        MetaReplication m = new MetaReplication("host1", "host2");
        MethodSemantics s = new MethodSemantics( );
        s.add(READONLY, "query",
"(Ljava.lang.String;)Ljava.lang.String;" );
        m.attach(db, s);

        Client c = new Client( );
        c.go(db);
```

The code example implements a replicated database. The main method instantiates the database object and a replication-control meta object. The replication-control meta object is initialized with the intended location of the replicas. When attaching the meta object to the database object, additional semantic information about the method query is passed to the meta object. Finally the client gets a reference to the database object and can use this reference like any other object reference. Invocations of the query method can be optimized by the replication meta objects because they have the information that it is a read-only method.

Neither the Database class nor the Client class have to know anything about replication — it is fully transparent!

Figure 11:Application example

7.4 Object locking

Every Java object can be viewed as a binary semaphore. Duplicating an object allows distributed synchronization using the object as a distributed lock. The replication-control meta object detects the attempt to acquire the lock with the MetaJava event mechanism (see Figure 2). Every replication protocol must implement a mechanism to maintain the lock.

7.5 An active replication protocol (PROT_ACTIVE_CENTRAL)

We have implemented an active replication protocol using one designated replication-protocol meta object in the callee replica group to solve the message order problem. If this replica group acts as caller, the same replication-protocol meta object is used to solve the call-duplication and object-duplication problems.

When receiving a method-enter event, the replication-control meta object sends the invocation request to the nearest replication-protocol meta object, which in turn distributes the request if necessary.

If every method invocation is distributed to all replicas, one can hardly expect any performance benefits from replicating the object. Thus the replication meta objects use the supplied by the programmer when binding the replication-control meta object to the base object. If a method has no side effects and is read-only, the method is executed only at the local replica.

7.6 A passive replication protocol (PROT_PASSIVE_SEQUENTIAL)

Our passive replication protocol ensures sequential consistency with a write-invalidate protocol. We implemented a slightly modified version of the write-invalidate protocol used in IVY [TSF90]. In IVY the unit of sharing is one page and the consistency-preserving protocol works on whole pages. In our implementation the unit of sharing is one object and the consistency-preserving protocol operates on fields. The replication meta objects keep track of the current state of each field of each replica (*current owner, writable, readable, invalid*).

If a field in one replica is to be written, control is passed to the meta level with the write-field event. If the field is not marked as writable, the current owner is requested to transfer the ownership to the writer. After receiving the ownership the field in all other replicas is invalidated. Finally the field is marked writable and the meta object sets the new value.

Read operations cause a read-field event. If the field is marked invalid, the owner is requested to grant read access. The owner marks the field in its own replica as readable, transfers the value to the requesting meta object and grants the read rights. The meta object marks the field as readable and returns the value as result of the read operation.

8 Related work

Munin, TreadMarks, Midway. Munin [CBZ91] and its successor TreadMarks [ACD+96] are DSM systems that provide multiple consistency preserving protocols. The programmer classifies shared variables according to a certain sharing pattern (read-only, migratory, write-shared, conventional). The system selects the best-fitting protocol for the specific type of sharing. Midway [BZS93] is a DSM system similar to Munin. It introduced entry consistent memory. Concerning adaptability Midway comes closest to our system. It allows multiple consistency models within the same program. But whereas Midway provides only a limited set of consistency models (entry, release and processor consistency) and protocols, we provide an infrastructure to implement any protocol.

Linda. Linda [CaG89] is a language independent framework for sharing data objects, called tuples, in a tuple space. The tuple space provides two operations *in* and *out* and can be considered as a distributed database using pattern matching as query mechanism. It would be interesting to implement the Linda tuple space and the pattern matcher using the replication mechanism described in this paper. The result could be a *replication transparent implementation* of Linda.

Orca. Orca [BBH+96] is an object-based DSM system. The compiler collects information about the read/write ratio of objects. Based on this information it decides whether to replicate an object or not. The Orca system differs from our approach in several points. We do not rely on compiler support to detect information about object use. Instead we assume that the programmer has more information about the intended use of the object and can make a more competent decision when binding the replication-control meta object to the base object. Furthermore the Orca system does synchronization by considering every object as a monitor. We rely on the programmer to correctly use synchronization as he would do in a non-distributed, multithreaded program.

GARF. GARF [GGM95] realizes fault-tolerant Smalltalk machines using active replication. Similar to our system, GARF uses meta objects, called *behavioral objects*. These behavioral objects are not as general as our meta objects as they can reify only method invocations. GARF relies on ISIS [BiJ89], a toolkit for fault-tolerant Unix processes, to solve the message order problem. Using ISIS enhances stability but limits extensibility.

Replication with OpenC++. Fabre et. al. [FNP+95] describe a reflective fault-tolerant system using active and passive replication. The system is built using the reflective language OpenC++ [ChM93]. They do not mention the call-duplication and object-duplication problems described in this paper and it is unclear whether their architecture can handle these problems. Passive replication is done using the primary-backup approach described in Section 2. This approach leads to incoherent backups between two checkpoints, but this is not a problem if replication is used exclusively for fault-tolerance.

9 Project status

We have implemented one protocol of the active and one of the passive replication model. The reification of load/store operations at base objects is currently only possible with a modified Java interpreter. This affects global computation and thus decreases the performance of all setfield/getfield operations. We have not yet considered static variables and static methods in replicated objects.

We are also working on several other applications of the MetaJava architecture. Meta objects for remote method invocation and just-in-time-compilation have already been implemented. Further proposals for meta objects include support for migration control, object clustering, security policies, active objects, atomic objects, synchronization support, and extended transactions [KlG96].

Information about the current project status is available at http://www4.informatik.uni-erlangen.de/Projects/PM/Java/.

10 References

ACD+96 C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, Vol. 29, No. 2, Feb. 1996, pp. 118-128.

ACL+93 D. Agrawal, M. Choy, H. V. Leong, A. K. Singh. *Maya: A simulation platform for parallel architectures and distributed shared memories.* Technical Report TRCS-93-24, Univ. of California at Santa Barbara, Dept. of Computer Science, 1993.

BBH+96 H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl. *Orca: a Portable User-Level Shared Object System.* Technical Report IR-408, Vrije Universiteit Amsterdam, June 1996.

BiJ89 K. Birman, T. Joseph. Exploiting Replication in Distributed Systems. S. Mullender (ed.). *Distributed Systems*. ACM Press, New York, 1989, pp. 319-367.

Bir85 K. P. Birman. Replication and Fault-Tolerance in the ISIS System. *Proc. of the 10th Symp. on Operating Systems Principles*, Orcas Island, WA, Dec. 1 - 4, 1985, Operating Systems Review, Vol. 19, No. 5, pp. 79-86.

BMS+93 N. Budhiraja, K. Marzullo, F. B. Schneider, S. Toueg. The Primary-Backup Approach. S. Mullender (ed.). *Distributed Systems.* ACM Press, New York, 1993, pp. 199-216.

BSS91 K. Birman, A. Schiper, P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, Vol 9, No. 3, 1991, pp. 272-314.

BZS93 B. Bershad, M. Zekauskas, W. Sawdon. The Midway Distributed Shared Memory System. *Proc. of the COMPCON Spring '93*, San Francisco, CA, Feb. 22-26, 1993, pp. 528-537.

CaG89 N. Carriero, D. Gelernter. Linda in Context. *Communications of the ACM*, Vol. 32, pp. 444-458, 1989.

CBZ91 J. B. Carter, J. K. Bennett, W. Zwaenepoel. Implementation and Performance of Munin. 1*3. ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, Oct. 13-16, 1991, pp. 152-164.

ChM93 S. Chiba and T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. *Proceedings of ECOOP '93, the 7th European Conference on Object-Oriented Programming,* Kaiserslautern, Germany, LNCS 707, Springer-Verlag, pp. 482–501.

Coo85 E. C. Cooper. Replicated Distributed Programs. *Proc. of the 10th Symp. on Operating Systems Principles*, Orcas Island, WA., Dec. 1 - 4, 1985, Operating Systems Review, Vol. 19, No. 5, pp. 63-78

Fer89 J. Ferber. Computational Reflection in class based Object-Oriented Languages. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '89*, New Orleans, La., Oct. 1989, pp. 317–326.

FNP+95 J. Fabre, V. Nicomette, T. Perennou, R. J. Stroud, Z. Wu. Implementing fault tolerant applications using reflective object-oriented programming. *Proceedings of the 25th IEEE Symposium on Fault Tolerant Computing Systems,* 1995.

GGM95 B. Garbinato, R. Guerraoui, K. Mazouni. Implementation of the GARF Replicated Objects Platform. *Distributed Systems Engineering Journal*, March 1995.

HeS92 A. Heddaya, H. Sinha. *Coherence, Non-coherence and Local Consistency in Distributed Shared memory for Parallel Computing.* Technical Report BU-CS-92-004, Boston Univ., May 1992.

HuA90 P. W. Hutto, M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. *Proc. of the 10th International Conference on Distributed Computing Systems*, Paris, France, May 28 - June 1, 1990, pp. 302-311.

HüL95 W. L. Hürsch, C. V. Lopes. *Separation of Concerns.* Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.

JoB89 T. A. Joseph, K.P. Birman. Reliable Broadcast Protocols. S. Mullender (ed.). *Distributed Systems*. ACM Press, New York, 1989, pp. 293-317.

Kic96a G. Kiczales. Beyond the Black Box: Open Implementation. *IEEE Software*, Vol. 13, No. 1, Jan. 1996, pp. 8-11.

Kic96b G. Kiczales et al. *Aspect-Oriented Programming*. Position Paper for the ACM Workshop on Strategic Directions in Computing Research, MIT, June 14-15, 1996 (http://www.parc.xerox.com/spl/projects/aop/).

KlG96 J. Kleinöder, M. Golm. MetaJava: An Efficient Run-Time Meta Architecture for Java. *Proc. of the International Workshop on Object Orientation in Operating Systems - IWOOOS '96*, October 27-18, 1996, Seattle, Washington, IEEE, 1996.

Lam79 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), Sept. 1979, pp. 690-691.

LiS88 R. J. Lipton, J. S. Sandberg. *PRAM: a scalable shared memory.* Technical Report CS-TR-180-88, Princeton University, Sept. 1988.

LiY96 T. Lindholm, F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Sept. 1996.

Mae87 P. Maes. *Computational Reflection*. Technical Report 87_2, Artificial Intelligence Laboratory, Vrieje Universiteit Brussel, 1987.

MaG95a K. Mazouni, G. Garbinato. Building reliable client-server software using actively replicated objects. *Proc. of the TOOLS95 (Technology of object oriented Languages and Systems)*, Versailles, Mar. 1995.

MaG95b K. Mazouni, G. Garbinato, R. Guerraoui. Filtering Duplicated Invocations Using Symmetric Proxies. *Proc. of the International Workshop on Object Orientation in Operating Systems - IWOOOS '95*, August 14-15, 1995, Lund, Sweden, IEEE, 1995.

Mos93 D. Mosberger. Memory Consistency Models. *Operating Systems Review*, Vol. 17, No. 1, Jan. 1993, pp. 18-26.

RaP93 K. Ravindran, S. Pichai. *Replication Protocol Structures for Fault-Tolerant Remote Procedure Calls.* Technical Report TR-93-18, Kansas State Univ., 1993.

Rie96 T. Riechmann. *Security in Large Distributed, Object-Oriented Systems.* Technical Report TR-I4-02-96, University of Erlangen-Nürnberg, IMMD IV, Mai 1996.

Sch90 F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, Vol. 22, No. 4, Dec. 1990, pp. 299-319.

SkS83 D. Skeen, M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, SE-9, No. 3, 1983, pp. 219-228.

Smi82 B. C. Smith. *Reflection and Semantics in a Procedural Language*. Ph.D. Thesis, MIT LCS TR-272, Jan. 1982.

Str92 B. Stroustrup. Run Time Type Identification for C++. *USENIX C++ Conference proceedings*, Portland, Or., Aug. 1992.

TSF90 M. Tan, J. M. Smith, D. J. Farber. A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems. *Operating Systems Review*, Vol. 24, No. 3, Jul. 1990, pp. 40-67.

YJT88 K. S. Yap, P. Jalote, S. Tripathi. Fault Tolerant Remote Procedure Call. *Proc. of the 8th International Conference on Distributed Computing Systems*, 1988, pp. 48-54.

ZSL+92 S. Zhou, M. Stumm, K. Li, D. Wortmann. Heterogeneous Distributed Shared Memory. I*EEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 5, Sept. 1992, pp. 540-554.