Using Object Replication for Building a Dependable Version Control System

Rüdiger Kapitza[1], Peter Baumann[1], and Hans P. Reiser[2]

[1] Dept. of Comp. Sciences 4, University of Erlangen-Nürnberg, Germany
rrkapitz@cs.fau.de
[2] LaSIGE, University of Lisboa, Portugal
hans@di.fc.ul.pt

Abstract. Object-oriented technologies are frequently used to design and implement distributed applications. Object replication is a well-established approach to increase the dependability for such applications. Generic replication infrastructures often fail to meet non-standard application-specific requirements such as support for client-side computing. Our FT*flex* replication infrastructure combines the fragmented object model with semantic annotations in order to customize and optimize replication mechanisms, and thus provides a more flexible replication infrastructure.

This paper presents DiGit, a replicated version control system based on the architecture of Git. DiGit is implemented with the help of the FT*flex* infrastructure for object replication. The contributions of this paper are twofold. First, the paper evaluates the fitness of our replication framework for a specific, complex application. We identify two advantages of the replication infrastructure: the ability to provide client-side code as a conceptually integral part of a remote service, and support for an optimized protocol for remote interaction. As a second contribution, the paper presents a powerful replicated version control system and shows the lessons learned from using object replication in such a system.

Keywords: Object Replication, Version Control System.

1 Introduction

Object-oriented technologies are frequently used to design and implement distributed applications. Such applications are faced with failures of various kinds. For example, nodes may crash or may even suffer from malicious intrusions, and communication between nodes may temporarily break down. Object replication is a well-established approach for coping with such kinds of failures. Many distributed object infrastructures provide support for object replication, either as an integral part or as an external add-on.

Generic object replication infrastructures frequently have some limitations. Being generic implies that the infrastructure implementation has to be suitable for many different kinds of applications. It is a hard challenge for a middleware to provide a range of configuration variants that include an ideal solution for every application, and make an automated selection of the best variant. Fragmented objects [14,11,19] are a technology that provides means for flexible adjustment to individual needs by the generation of custom code. This code substitutes the static stub and skeleton

R. Meier and S. Terzis (Eds.): DAIS 2008, LNCS 5053, pp. 86–99, 2008.

that is commonly used in distributed object middleware. The FT*flex* replication infrastructure [20] combines the fragmented object model with semantic annotations in order to customize and optimize replication mechanisms. In this paper, we use a specific complex application—a distributed version control system—to assess the replication mechanisms provided by that replication infrastructure.

Version control systems, such as Concurrent Versions System (CVS) [3] and Subversion (SVN) [17], are commonly used for software development. Most systems use a central server for storing all data and history information. This server may fail due to hardware or software problems. If developers use a third-party service (as it is the case for many open-source projects that use, for example, the SourceForge service), there is also the risk of intentional shut-down of that service. Replicating a repository on multiple hosts, potentially located in independent administrative domains, helps to maintain system functionality in spite of such failures. Usually, the repositories of large projects are mirrored using tools like rsync or FTP. This approach reduces the risk of data loss in case of failures and provides scalability for read-only access. However, manual intervention is necessary to determine the last valid state of the repository and to set up the a new primary repository to recover from a failure of the main site. During this time a coordinated exchange of development progress is hard to achieve.

In this paper, we present the design of DiGit, a replicated version control system based on the architecture of Git [4], which is the system that the Linux kernel project currently uses for version management. DiGit provides fault-tolerant replication mechanisms for a global repository in combination with decentralized client-side repositories. DiGit uses the FT*flex* object replication infrastructure and thus its implementation is based on the concept of fragmented objects. The replication infrastructure uses semantic information on operations to tailor the provided mechanisms. In essence, this paper makes two important contributions. First, it evaluates the fitness of the adaptable FT*flex* replication framework for a specific, real-world application. Second, it presents an architecture for a powerful replicated version control system that is able to transparently tolerate a limited number of faults without service unavailability and shows the lessons learned from using object replication in such a system.

This paper is structured as follows. First, we survey related work. In Section 3, we briefly describe the generic FT*flex* infrastructure. Section 4 presents the design of our DiGit distributed version control system. Then we outline evaluation results. In Section 6, we discuss the lessons learned from our implementation. Section 7 concludes.

2 Related Work

Object replication is supported in many distributed middleware systems. A prominent example for such support is the Fault-Tolerant CORBA (FT-CORBA) standard [16]. This standard is implemented in existing systems in various ways, for example using an interception approach in the Eternal system [15], an integration approach in Orbix+Isis [12], or a service approach in OGS [6]. The FT-CORBA standard is an example of a complex standard for fault-tolerant middleware infrastructures targeted at heterogeneous systems. Even after maturing for more than a decade, many deficiencies can still be identified [7]. An important observation is that replication cannot easily

be added to object-oriented applications in a transparent way. In this paper, we discuss some slightly different limitations of existing object-replication frameworks: The main challenges that arise from replicating a version control system are the local manipulation of files at the client side and efficient protocols for the transfer of large data (e.g., client operations on the repository).

The issue of customizing (and thus optimizing) remote communication for simple, non-replicated distributed objects has previously been investigated in various projects. Usual solutions provide a framework [18] or, as in Kurmann and Stricker [1], completely redesign the communication stack and the ORB. Such extensions to object middleware systems are incompatible with extensions for fault-tolerant replication. For example, efficient communication needs to be integrated into state transfer, and all client operations that modify the object state must be applied consistently to all replicas.

Fragmented objects have previously been used to construct flexible distributed systems. Replication with fragmented objects has previously been used in Globe [2]. Unlike the version control system that we propose in this paper, Globe does not consider client-side computing or custom communication protocols.

While traditional version control systems such as CVS and SVN use a central repository server, a decentralized approach has gained popularity in systems such as Arch [8], Darcs [5], Monotone [10], and Git [4]. These systems focus on a decentralized development model, in which each developer can have their own local repository. This approach makes off-line work under local revision control possible. Local changes can later be re-integrated into a central main branch, usually maintained by a head developer. The head developer may also allow authorized users to write directly to the central repository, thus allowing to mimic a centralized development model as well. If the central repository becomes unavailable, only local repository operations are possible, and interaction between developers is not possible. In DiGit, we also support developer-side local repositories, but the key advantage of our system is that it aims at increasing the availability of the central repository by fault-tolerant replication.

The Pastwatch [22] version control system is similar to our approach in the sense that it replicates a repository on multiple hosts, while providing a functionality similar to CVS. However, Pastwatch uses optimistic replication on a peer-to-peer-infrastructure and does not make strong guarantees about data consistency. Conflicting modifications by multiple developers are handled by creating branches that require manual conflict resolution. DiGit, on the other hand, uses a general-purpose replication infrastructure (FT*flex*) and provides strong consistency guarantees on the replicated repository.

3 Replication Infrastructure Based on Fragmented Objects

The FT*flex* object replication infrastructure [20] uses the fragmented-object model [14,11,19] provided by the Aspectix middleware to integrate replication mechanisms into distributed applications. From an abstract point of view, a fragmented object is defined by its identity, interface, functionality, and state. The implementation of these items is not bound to a specific location, but instead can be distributed arbitrarily over multiple fragments, located on multiple hosts. The FT*flex* infrastructure supplies a code generation tool that creates object-specific fragment code for client access and

for replica consistency management. The replication of an application remains fully transparent to clients. Semantic annotations at the interface level allow the developer to customize the provision of fault tolerance.

3.1 Replication with Fragmented Objects in FT*flex*

For realizing a replicated service, FT*flex* uses the fragmented object infrastructure of Aspectix to load service-specific fragment code at the client side and at replica locations, without requiring internal middleware modifications (see [9] for details). Fig. 1 shows a fragmented object that implements a fault-tolerant service; this object is internally composed of replica fragments and access fragments.

The development process of a replicated service in FT*flex* consists of defining the global object interface in CORBA IDL, implementing the functional parts of the service, and creating the fragment code. The creation of fragment code is done automatically by tools; these tools can make use of semantic annotations provided by the developer, as we will describe in Section 3.2. The annotations enable creating a customized layer between the client and the core framework and also between the framework and the replica implementation.

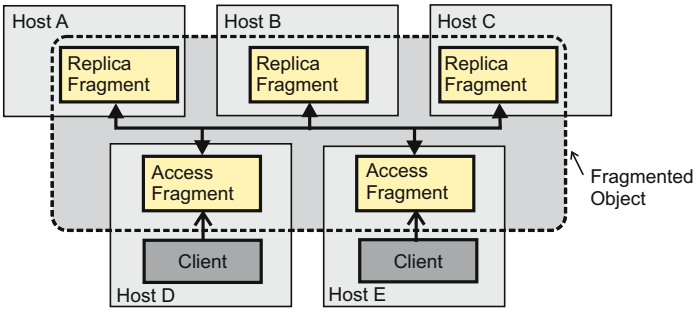

Fig. 1. A replicated service realized as a fragmented object

The generated fragment code handles most replication-relevant issues. It implements marshalling and unmarshalling, similar to an ordinary stub. Furthermore, it handles communication of clients with an available member of the replica group, ensuring at-most-once call semantics even in case of fail-over handling. With an active replication strategy, all client requests are distributed to the replicas using totally ordered group communication. In addition, each replica fragment contains semantic information about methods that has been extracted from the annotations. This information is used to improve the performance of request executions.

The Aspectix middleware uses CORBA IORs to reference fragmented objects via an APX profile. This profile contains a unique object ID, a specification of the initial fragment type to load and contact information of other fragments. The initial fragment type can be specified in a language-independent way; a code-loading service [13] is

used for obtaining the corresponding code for the local platform. The contact data in the APX profile contains information about the replica group.

3.2 Semantic Information and Code Generation

Similar to a traditional CORBA IDL compiler for stubs and skeletons, FT*flex* can generate client-side and replica-side fragments using IDL interface information. In addition, our architecture allows the developer to express semantic knowledge in order to improve and customize the replication mechanisms. Currently, FT*flex* supports several annotations on a per-method basis: annotations can be provided to specify if an object operation interacts with the replica and modifies its state, if it is a read-only operation, if it is parallelizable with other methods, or if it is a method that can be computed locally at the client side without interacting with the replica group.

The current prototype of the code-generation tool is based on IDLflex [21]. IDLflex parses CORBA IDL, evaluates an XML-based mapping specification, and uses this specification to create arbitrary output code. The tool supports semantic annotations in IDL files, expressed as `#pragma annotate` statements. A custom mapping specification defines how to evaluate these statements and defines the corresponding code-generation process.

If a method is marked as read-only, the communication with the replica group will be handled differently (using unicast instead of multicast). Information about parallelizability is passed from the replica fragment to the local application-level scheduler. Furthermore, annotations affecting the location of code have an influence on whether methods are part of the external service interface (and thus part of the client-side fragment interface), whether they are implemented in the client-side fragments only (but may be absent in the replica fragments), or whether they are implemented in replica fragments. Client-local method implementations (which are still part of the abstract service interface) are useful for methods that, for example, validate client data in a state-independent way or that provide static information to the client.

4 Replicated Decentralized Version Control

This section presents the design and implementation of DiGit, a reliable distributed version control system. DiGit uses FT*flex* for replicating a central repository service, and thus evaluates the FT*flex* infrastructure using a complex real-world application.

4.1 Background

A version control system records the development history and enables collaboration of multiple developers by supporting distributed and concurrent work on a source-code tree. To take part in the development, a developer checks out the sources, modifies them, and finally commits her changes. If the repository is provided via a central service (e.g., CVS), the unavailability of this service prevents coordinated development progress.

Our DiGit implementation aims at offering high availability of a virtual central repository. By virtual we mean that the repository is, in fact, replicated over multiple

physical hosts, but for clients it appears as a single central repository. Our prototype re-implements the Git system that was originally invented by Linus Torvalds. Git has a lean storage model that can be implemented in a straightforward way; it is proven to work well in large-scale projects such as the Linux kernel. Git already offers decentralized revision control of software. However, this support primarily targets the local availability of a repository and the possibility to later re-integrate local changes to a remote main repository. A public main repository, such as used in the case of the Linux project, still represents a single point of failure. Once down, no further check-outs and check-ins are possible. Thus, a centralized, prompt exchange of development progress is stopped. In addition, Git provides no out-of-the-box support for mirroring or replication of the repository on order to achieve scalability and fault tolerance. In the context of the Linux project this is solved by considerable big hardware efforts (see http://www.kernel.org) and replication via FTP or rsync. For this reason, the main goals of our reimplementation are to achieve fault tolerance and high availability without requiring external mechanisms or manual intervention.

Next, the basic concepts of Git and its storage format are explained. Then, the architecture and the API of our replicated Git repository are described. Our prototype implementation provides a command-line tool for repository management, but we will not focus in this tool.

4.2 Basic Concepts of Git

The storage model of Git is composed of four basic object types that are stored as ordinary files on disk. Each object has a header that identifies the type and the size of the object. The files are named after the SHA1 hash calculated over the content. Any modification to a file will result in a mismatch between the file content hash and the file name, and these modifications can easily be detected.

The four basic object types are Blob, Tree, Commit, and Tag. A *Blob* represents a file that is under version control. A Blob does not reference any other object. A *Tree* object contains of a list of names that reference Blob and Tree objects, together with meta data for each list entry, such as access time and permissions. A Tree object thus is similar to a directory. Third, a *Commit* object uniquely identifies a certain version of a branch. A Commit object consists of a changelog message, the names of the modified Tree objects, and the name of the direct ancestor Commit objects if there are any. A *Tag* object offers the additional support to uniquely identify and secure certain source code versions by a name and an optional certificate.

Fig. 2 illustrates the basic storage model of Git. The example shows the use of the three main object types, Blob, Tree, and Commit. Three files are represented by Blobs. There is one Tree object that represents the project directory and one Commit object that describes the initial version.

If a developer commits changes to the repository, all modified files have a different hash value than before and thus result in new Blobs. This leads to changes in the corresponding Tree objects, which results in different hash sums of the Trees and thus in the generation of new Tree objects. The Commit object documents the dependencies between the former version of the project and the new one. Fig. 3 shows three check-in

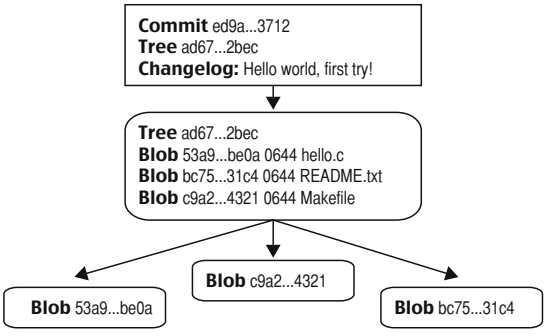

Fig. 2. Example of the basic Git storage model

operations on a repository with two files: the first operation creates the initial version, the second revision only modifies one file, and the third update changes all files.

This structure enables easy branching and merging of different development lines. The only things that are necessary are to store the modified files and to insert a new Commit object that references the joint branches.

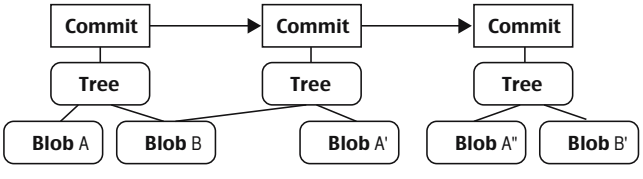

Fig. 3. Commit operation in Git

The Git storage model can be implemented in a straightforward way, and it prevents the undiscovered modification of files due to the rigorous usage of hashes. The drawback of the storage model is the redundancy it introduces. Versions that differ only slightly result in completely new Blob objects. This leads to an enormous waste of storage. Common version control systems like CVS or SVN usually store only the difference between two successive versions of files. To achieve a similar behaviour, Git supports custom archives, named *pack files*. A pack file is a collection of objects, individually compressed, with delta compression applied. These pack files are used to reduce the repository size by archiving older revisions and to enable an efficient network transfer for data-intensive repository operations, such as the initial checkout of a project.

4.3 Design of the DiGit Version Control System

Git uses a distributed development model that requires every developer to have their own local repository. The same basic model is used in our replicated variant. The goal of DiGit is to provide the same functionality as Git for the local repository operations,

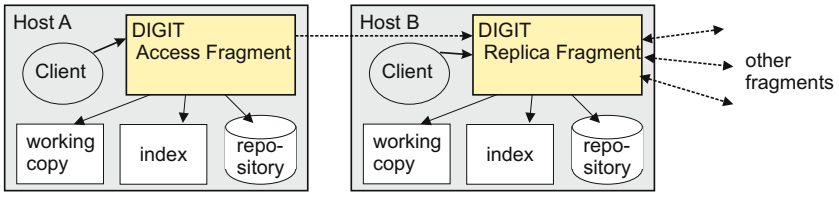

Fig. 4. A DiGit service is composed of access fragments and repository fragments, and both manage a working copy, an index, and the repository

and in addition support a replicated global repository that remains available in spite of the failure of some replicas.

DiGit is implemented as a fragmented object, as illustrated by Fig. 4. The object interface of the DiGit service offers an interface that includes methods to operate on the local as well as on the global repository. A single fragment handles both kinds of repositories. Operations on the local repository are executed locally, while operations on the replicated repository require communication with the group of replicas.

A client-side fragment may either work as a smart stub to access a remote group of replica fragments, or it may be part of the replica group itself. A local repository and a local replica of the global repository share the same storage. This means that a data object that is stored in both needs only a single entry in the storage under its hash. Furthermore the hash is registered in a shared index file for fast access.

The local repository can be synchronized to the global repository in the same way as a local Git repository is synchronized to a remote server. A developer does not operate directly on the global repository. Instead a copy of the global repository, called a clone, is created in the local repository, and all local modifications are first committed to a local branch of that clone. A `pull()` operation transfers updates from the global repository to the local clone, and these updates can then be merged to the local development branch. Conversely, `push()` applies changes in the local branch to the global repository.

Most of the DiGit operations operate on the local working copy, the index, or the local repository. These operations are implemented in a client-side fragment in a way that closely resembles the original Git implementation. The operations for cloning, pushing, and pulling repository data are the key operations that access the global repository. The main challenge that the DiGit implementation addresses is to make these operations work reliably and efficiently on a group of repository replicas.

4.4 DiGit Operations on a Local Repository

The DiGit methods for local revision management are annotated via the *local* keyword (see interface definition in Fig. 5). This tells the FT*flex* framework that the operations are implemented in the client-side fragment instead of at a remote server fragment.

At this point, we assume that the client-local repository is already populated by data from the global repository (how this is achieved is explained in the following section). A developer can `checkout()` a certain source version from the local repository into

```
#pragma annotate(local)
void checkout(in string ref) raises (NonExistingBranch, NonExistingObject);

#pragma annotate(local)
string add(in string filePath);

#pragma annotate(local)
string refresh(in string filePath);

#pragma annotate(local)
string commit(in string ref) raises (NonExistingObject);

#pragma annotate(local)
boolean merge(in string ref) raises (NonExistingBranch);

#pragma annotate(local)
void branch(in string name) raises (NonExistingBranch);
```

Fig. 5. Basic operations to manage the local repository

a local working directory, and can `add()` new files to the repository. The `commit()` operation creates a new version in the local repository, and the `merge()` operations updates the local working copy with data from the repository. If a certain source version should be used for subsequent independent development, the developer can use the `branch()` operation a new development branch.

Identical to Git, the data of files is internally stored in an index file. An `add()` operation automatically adds the content of the file at addition time to the index. The index content defines the updates that will be made to the local repository by the `commit()` operation. The `refresh()` operation updates the index data with the file data in the local working copy. With the explicit `refresh()` operation, a developer has the option to collect distinct changes for the next commit while keep on developing.

4.5 DiGit Operations on a Distributed Repository

The part of the DiGit interface relevant for operations on the distributed global repository is shown in Fig. 6. Several of the operations, such as the initial checkout of a repository and the update of large source trees, are data intensive. Such data intensive operations are a great challenge for replication infrastructures based on distributed object middleware. Many infrastructures provide only a plain remote invocation mechanism without dedicated support for efficient bulk data transfer.

The fragmented object model provides an alternative solution, as a service developer has the freedom of choice to use arbitrary object-internal communication. DiGit uses this approach to make a clear separation between control messages and file transfer, wherever it is possible. Control messages and state-modifying messages are handled by the implemented framework as standard method invocations, whereas data is transferred via an adapted custom protocol taken from the original Git implementation.

First of all, a decentralized Git service has to be initialized and started. This is achieved by creating an initial replica on some host, and this replica has an associated

```
#pragma annotate(local)
boolean clone(in string path);

#pragma annotate(local)
boolean pull();

#pragma annotate(local)
BranchArray push();

BranchArray pushPack(in ByteArray pack, in BranchArray old, in BranchArray new);

#pragma annotate(readonly,private)
StringArray getRepositoryServers();

#pragma annotate(private)
void addRepositoryServer(in string address);
```

Fig. 6. Operations to manage and synchronize the local with the global repository

IOR for a fragmented object. All service replicas provide not only a complete copy of the repository, but also run a TCP server for bulk data transfer. This server is started during the creation of a replica. As soon as an initial service replica has been successfully started, any client can use the repository service by binding to its IOR.

After binding to the repository service, the client has to populate the local repository using the sources of the replicated global repository. Only after this step, a developer can use the set of methods for local repository management described before. The `clone()` method populates the local repository with a copy of the global repository. The client-side access fragment selects one of the service replicas and requests a transfer of the current repository state. The TCP server of the contacted replica provides this state as a pack file. The client-side fragment uses the pack file to populate the local clone of the repository.

As the client-side IOR that contains contact addresses may partially be out of date, the replica group provides a `getRepositoryServers()` method that returns all data server addresses. This method is annotated as *private* and *read-only* method, which means that it is not visible on the outer interface of the DiGit services, and that it is invoked at only a single replica, instead of being distributed to all replicas.

After the repository of the access fragment has been initialized, it can be updated by calling `pull()`. The `pull()` method is also implemented as a custom local operation. It requests the current state of the repository from one of the replicas using the TCP server for bulk transfer. The bulk transfer implements an interactive process with the replica. First, the replica returns meta data about all branches and their current head versions (the last commits). The access fragment compares this information with the entries of the global part of the repository and then determines the missing commit objects. Finally, all modified objects are sent to the client as a pack file.

A developer commits local changes into the replicated global repository by calling the `push()` method. This local method communicates with the bulk transfer server of a replica to determine whether there is a conflict with the global repository. If the global state has been modified since the last comparison, the return values signals a conflict,

which has to be resolved before the changes can be committed to the global repository. Otherwise, the method generates a pack file that describes the updates to be made to the central repository. The pack file is then passed to the `pushPack()` method, which consistently modifies the state in the replicas. This global state modification is executed with the generic replication methods that the FT*flex* replication infrastructure provides. This implies that the update is consistently distributed to all replicas using totally ordered group communication.

Setting up a new replica requires a state transfer. The generic way for creating additional replicas in FT*flex* is first to bind to the IOR of the replicated service (thus instantiating a local access fragment), and then to upgrade the local fragment to a full replica. In DiGit, a local access fragment generally has a clone of the central repository. Thus, there is no need for a full state transfer from existing replicas to the new replicas. However, the local clone of the global replicated repository could be outdated, as there might have been commits to global repository since the last invocation of `pull()`. An extended state transfer implementation accounts for this fact and minimizes the transfer costs. The new replica submits revision information about its local copy of the global repository to all state providing nodes. Next, the replicas use this information to generate a pack file that is transferred to the joining replica. After this operation, the new replica adds the address of its data server via `addRepositoryServer()` to the set of state-providing replica servers.

5 Evaluation

This section evaluates the basic operations of our prototype and compares a DiGit repository with three replicas with the original non-replicated implementation of Git version 1.5.3.7. All measurements have been made in a 100 MBit/s switched Ethernet network on a homogeneous set of PCs with a AMD Athlon 2.0 GHz CPU and 1 GB RAM, using Linux kernel 2.6.17 and SUN Java SDK 1.5.0_09. The tests used a small project repository hosting 600 files with a history of 504 revisions and a size of 87 MB. For Git we used ssh as underlying remote protocol. We focused on operations with remote interaction. Tab. 1 summaries the results.

First we evaluated the time to initialize a client and clone the repository from a remote site or a remote global repository in case of DiGit. For DiGit this operation is independent from the number of replicas as clone represents a read-only operation and only one replica is accessed. Albeit using a similar protocol and performing almost identical operations Git is 6 times faster. The reason for this large performance gap lies in fact that the native Git implementation is highly optimized and makes heavy use of memory mapped files and uses faster compression routines to build the pack files, while our DiGit prototype does not yet use such techniques.

Next we measured the time to set up a new replica from an up-to-date client node. This operation is important if a replica crashed or was intentionally shut down and needs to be replaced. It took 793 ms on average to integrate a initialized client node.

This leads to the last two experiments. We updated the global/remote repository via push. The update consisted of 100 locally committed revisions. This time Git was twice as fast as DiGit. The reasons are similar to the first measurement, but in addition the

Table 1. Measurements DiGit vs. Git

	Initialize client fragment / Clone	*Become a replica*	*Push (100 revisions)*	*Pull (up to date)*
DiGit	90843 ms	793 ms	20866 ms	15 ms
Git	14613 ms		10271 ms	1665 ms

pushed revisions have to be distributed to all replicas via the group communication framework when using DiGit. The last measurement pulled the latest revisions from the remote/global repository. In the context of the measurement the local repository include already the current version so we measured only the request. This time DiGit performed much better than Git. The reason is that DiGit had an open connection from the client to the server, whereas for Git the connection process took most of the time.

6 Lessons Learned

We discuss the following three questions in order to analyse our replication strategy for the version control application:

- Is object replication suitable for providing a dependable version control system?
- What are the potential benefits from the fragmented object model?
- What are the potential benefits from semantic annotations?

In general, it is not easy to describe a version control system as a service object with an adequate interface, because there is an inherent lack of distribution transparency. Usually, the interface of a remote service is the basic contract for the interaction of a client with a remote service.

In the context of a version control system, however, the service functionality is not limited to the remote server, but also includes the protocol for applying modifications to client-local file. A typical object-oriented implementation would only specify the interface of the remote service, and let the client implement all the parts that are necessary at the client side. The fragmented object model makes a huge difference here, as client-side functionality can be implemented as a fragment of the version control service. This fragment is conceptionally part of the service itself, and it can be loaded automatically at the client, using a dynamic loading service if it is not available locally.

A second requirement is the need for efficient bulk data transfer for data-intensive operations. If the repository is implemented as an object-oriented service, this requirement is also hard to fulfil with standard replication infrastructures for objects. However, the concept of fragmented objects can again provide a large benefit: A local fragment at the client side can implement custom optimized communication protocols between client-side fragments and repository fragments.

The dynamic loading of fragments at the client side is, of course, faced with security considerations. These can be addressed by using digital signatures for code that is loaded automatically.

Finally, annotations are an important aspect to improve throughput. The DiGit implementation uses annotations mainly to specify that code shall be local to the client.

This is not only used for code that handles the local repository and working copy, but also for implementing custom mechanism for bulk transfer of data. For interaction between fragments, our prototype uses private methods, which are not visible to clients, but available for internal use in client-side fragments. The read-only annotation speeds up the method that queries the up-to-date replica list.

7 Conclusions

This paper presented the design of DiGit, a replicated version control system based Git. Our system provides a consistent, dependable central repository even in failure situation by using efficient object replication technology. Our approach differs from most current distributed version control systems, which typically use a single central repository, in combination with decentralized local repositories that allow the tracking of changes during times in which the central repository is not available. Our approach replicates the central repository in order to make it more available. DiGit can be combined with local repositories, which are still useful, for example, for clients with no network connectivity at all and for local development.

A useful future extension of DiGit could allow developers to replicate their local repositories as separate branches in the global repository. This way, snapshots of their work could be accessed by other developers and their work would be preserved if their local machine crashed. Furthermore, additional evaluation could focus on WAN scenarios and performance in failure situations.

The replication of DiGit has been realized with support from our FT*flex* replication infrastructure. This infrastructure uses fragmented objects and annotations-based code generation in order to provide a high degree of customizability. The DiGit system benefits from this architecture in a way that would not have been possible in a traditional object replication infrastructure. First, the fragments allow loading repository-specific code automatically at the client side for manipulating the client-local repository. Conceptionally, this code is still part of the remote central repository. Second, the annotations allow optimizing the performance of the replication strategies.

References

1. Kurmann, C., Stricker, T.M.: Zero-copy for CORBA - efficient communication for distributed object middleware. In: 12th IEEE Int. Symp. on High Performance Distributed Computing, pp. 4–13. IEEE Computer Society, Los Alamitos (2003)
2. Bakker, A., Amade, E., Ballintijn, G., Kuz, I., Verkaik, P., van der Wijk, I., van Steen, M., Tanenbaum, A.S.: The globe distribution network. In: Proc. of the USENIX Annual Conference, pp. 141–152 (2000)
3. Bar, M., Fogel, K.: Open Source Development with CVS, 3rd edn. Paraglyph (2003)
4. Baudis, P.: Git - fast version control system, `http://git.or.cz`
5. Darcs, `http://abridgegame.org/darcs`
6. Felber, P.: The CORBA Object Group Service: A Service Approach to Object Groups in CORBA. PhD thesis, EPLF, Switzerland, Number 1867 (1998)
7. Felber, P., Narasimhan, P.: Experiences, strategies, and challenges in building fault-tolerant CORBA systems. IEEE Trans. Comput. 53(5), 497–511 (2004)

8. GNU arch, http://www.gnu.org/software/gnu-arch
9. Hauck, F.J., Kapitza, R., Reiser, H.P., Schmied, A.I.: A flexible and extensible object middleware: CORBA and beyond. In: Proc. of the Fifth Int. Workshop on Software Engineering and Middleware. ACM Digital Library (2005)
10. Hoare, G., Smith, N., Scherger, D.: Monotone - A distributed version control system, document version 0.35 (2006)
11. Homburg, P., van Doorn, L., van Steen, M., Tanenbaum, A.S., de Jonge, W.: An object model for flexible distributed systems. In: Proc. of the 1st Annual ASCI Conference, pp. 69–78 (1995)
12. IONA and Isis. An introduction to Orbix+Isis. IONA Technologies Ltd. And Isis Distributed Systems, Inc. (1994)
13. Kapitza, R., Schmidt, H., Bartlang, U., Hauck, F.J.: A generic infrastructure for decentralised dynamic loading of platform-specific code. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, Springer, Heidelberg (2007)
14. Makpangou, M., Gourhant, Y., Narzul, J.-P.L., Shapiro, M.: Fragmented objects for distributed abstractions. In: Casavant, T.L., Singhal, M. (eds.) Readings in distributed computing systems, pp. 170–186. IEEE Computer Society Press, Los Alamitos (1994)
15. Moser, L.E., Melliar-Smith, P.M., Narasimhan, P.: Consistent object replication in the eternal system. Theor. Pract. Object Syst. 4(2), 81–92 (1998)
16. Object Management Group (OMG). Common object request broker architecture: Core specification, version 3.0.2. OMG document formal/02-12-02 (2002)
17. Pilato, C.M., Collins-Sussman, B., Fitzpatrick, B.W.: Version Control with Subversion, 1st edn. O'Reilly Media, Sebastopol (2004)
18. Pyarali, I., Harrison, T.H., Schmidt, D.C.: Design and performance of an object-oriented framework for high-speed electronic medical imaging. Computing Systems 9(4), 331–375 (1996)
19. Reiser, H.P., Hauck, F.J., Kapitza, R., Schmied, A.I.: Integrating fragmented objects into a CORBA environment. In: Proc. of the Net.ObjectDays, pp. 264–272 (2003)
20. Reiser, H.P., Kapitza, R., Domaschka, J., Hauck, F.J.: Fault-tolerant replication based on fragmented objects. In: Eliassen, F., Montresor, A. (eds.) DAIS 2006. LNCS, vol. 4025, pp. 256–271. Springer, Heidelberg (2006)
21. Reiser, H.P., Steckermeier, M., Hauck, F.J.: IDLflex: a flexible and generic compiler for CORBA IDL. In: Proc. of the Net.Object Days, pp. 151–160 (2001)
22. Yip, A., Chen, B., Morris, R.: Pastwatch: a distributed version control system. In: Proc. of the USENIX/ACM 3rd NSDI (2006)