Towards Upgrading Actively Replicated Servers on-the-fly

Marcin Solarski
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10551 Berlin, Germany
Email: solarski@fokus.fhg.de

Hein Meling
Department of Telematics
Norwegian University of Science and Technology
N-7491 Trondheim, Norway
Email: meling@item.ntnu.no

Abstract

Change management is indispensable in most distributed software systems, which are continuously being modified throughout their life cycle. Managing the changes at runtime in highly available distributed systems is especially challenging as upgrade of a running system should not deteriorate its availability characteristics.

We present a distributed algorithm that allows to dynamically upgrade an actively replicated server so that the server is operational, even during the upgrade process. The algorithm makes use of the core functionality of an underlying Group Communication System that has been extended with a recovery mechanism. Its design enables dependable upgrades of replicated software in the presence of replica crashes. The presented mechanisms are part of the Dynamic Upgrade Management Framework aiming at supporting and managing dependable upgrades of distributed systems on the fly.

1 Introduction

Most distributed software systems evolve during their lifetime. The spectrum of software change is wide, and ranges from program corrections and performance improvements to complex changes of the overall functionality and structure of the system. Such changes may be necessary to adapt the system to new user requirements. In a conventional approach to system maintenance, the system runtime has to be interleaved with maintenance breaks in which the necessary changes are manually applied to the system. This approach, however, is not suitable in large distributed systems that have to be highly available.

Dynamic upgrade is a technique that allows the introduction of necessary changes into the system, so that the system remains operational even while being upgraded. Thus, system availability does not decline as a result of the system upgrade. Traditional techniques for increasing system availability have been based on masking hardware failures [8]. The idea is to introduce redundancy into the system by replicating certain system components. A common approach to provide object and process replication is based on the concept of a Group Communication System (GCS) [2]. Replicating system components eliminates the effects of transient hardware and software failures. However, replication cannot prevent system failures due to software design faults whose contribution to system unavailability grows sharply with the increasing complexity of software systems.

Dynamic upgrade, as presented in this paper, aims at shortening the recovery time for design faults. The formula characterizing system availability is often expressed as follows:

$$\frac{MTTF}{MTTF + MTTR}$$

In general, availability can be improved by increasing the Mean Time To Failure (MTTF) or decreasing the Mean Time To Repair (MTTR). Dynamic upgrade can improve system availability in two ways:

- The system's time to repair becomes shorter. There is no need to take the system down during its maintenance (upgrade).
- The system's time to failure is increased. A new version is typically less faulty. Thus, the system failure rate decrease after the upgrade.

In this paper we present an algorithm for upgrading an actively replicated server, i.e., a number of server instances (replicas) processing client requests in parallel [8]. The algorithm is self-stabilizing and it introduces only minimal additional load on the system, while maintaining continuous availability during the upgrade.

The rest of the paper is structured as follows. In Section 2, we describe the underlying system model and state the assumptions we have made for designing the upgrade algorithm. Section 3 gives an overview of the algorithm, followed by a detailed description and concludes with a brief analysis. In Section 4 we describe the implementation of the upgrade algorithm based on Jgroup/ARM [6]. Section 5 discusses other work that relates to our upgrade algorithm. Finally, Section 6 concludes the paper and presents our ongoing work to validate the algorithm.

2 System Model

We consider a client-server architecture in an asynchronous distributed system augmented with unreliable failure detectors [1], in which the basic unit of replication is the server. We assume server replicas fail only by crashing, and once crashed it does not recover. However, a replica that is considered to have crashed may be replaced by a new instance of the replica.

In this paper, we assume an actively replicated server, in which each server processes client requests deterministically. It is implemented using a view-oriented GCS [2, 7] extended with a recovery mechanism [5]. The recovery mechanism works by creating replacement replicas for the replicas that the GCS considers to have crashed. Clients issue requests through the GCS, using totally ordered multicast, to the server replicas (group) and receive replies from the servers.

2.1 Upgrade Assumptions

In this paper, we consider only upgrading the software of server replicas and not the clients. This puts certain restrictions on what can be achieved with respect to compatibility between client and server objects. Thus, in order to substitute a version v of a replica, we have made the following assumptions under which our algorithm will work:

- *Upgrade atomicity with respect to other upgrades of the server.* Server upgrades are atomic with respect to each other, i.e., two upgrade processes cannot interleave. Furthermore, the replica cannot process client requests while being upgraded.
- *Input conformance.* Replica version $v+1$ is replaceable with version v. In terms of input, the input accepted by version $v+1$, is a subset of the acceptable input to version v of the replica. In terms of interfaces, we assume that version $v+1$ offers a compatible interface to that of version v, possibly augmented with new functionality.
- *State mapping and output conformance.* There exist a mapping from the state of version v to the state of version $v+1$ of the replica, such that version $v+1$ produces the same output as version v, given some input acceptable to version v.
- *Upgrade atomicity with respect to client upgrades.* Clients provide input acceptable to version $v+1$, but not acceptable to version v, only after the upgrade algorithm terminates.

Furthermore, we assume that code for the new software version has been deployed to all the system nodes. The code can be started and its runtime instance can become a replica of the server after joining the server group.

3 The Upgrade Algorithm

In this section, we present a software upgrade algorithm whose purpose is to exchange the code of a running actively replicated server with a new version of the software. The algorithm is designed to avoid single points of failure and it is implementable given the assumptions in Section 2. First we state the requirements for the algorithm, followed by a brief overview and detailed description. We conclude this section with a brief analysis of the algorithm.

3.1 Algorithm Requirements

Given the assumptions in Section 2, we state the requirements that our algorithm should satisfy:

- *Continuous availability.* The replicated server should be capable of processing client requests during the upgrade process. In an active replication scheme, this requirement determines the minimum redundancy level, l, that is needed to provide service to clients. Typically, this would be $l \geq 1$.
- *Fault tolerance.* The algorithm should terminate in bounded time, after successful upgrade of all server replicas in spite of replica crashes during the upgrade.
- *Low resource usage.* The algorithm should exploit the least possible resources. In particular, it is not feasible to substantially increase the original redundancy level while upgrading the system.

3.2 Algorithm Overview

The algorithm is based on the following idea: to upgrade an actively replicated object it is enough to upgrade each of its replicas in a sequence of individual upgrades. However, the same algorithm may also be used to upgrade multiple replicas simultaneously. The number of replicas that can be upgraded in parallel depends on the availability requirements, i.e., the minimum redundancy level allowed.

Let the R denote the set of server replicas that is to be upgraded. Below we sketch the steps of the algorithm informally:

1. Reliably multicast an upgrade request to replicas in R.
2. Select a candidate replica, $r \in R$, to be upgraded next.
3. Check whether replica r can be upgraded.
    (a) If so, replica r is then stopped and replaced with its new software version. Otherwise, the replica may process client requests and its upgrade is postponed until it is possible. At the same time, the rest of the replicas are available to process client requests.

(b) After upgrading a replica, the state of the new replica, replacing r, must be initialized with the state of the running replicas.

4. The upgraded replica, r, is removed from R.

5. Repeat steps 2-4 until all replicas have been upgraded.

3.3 Algorithm Description

The algorithm is designed to have distributed control, that is there is no global coordinator. All the server replicas perform the same algorithm and are symmetric in this sense. Figure 1 illustrates a state-oriented representation of the algorithm, using SDL notation. The algorithm is described from the view point of a single replica, and it is referred to as *this replica*.

After the replica is initialized (triggered flag is false) and joins the server group (join_group), it enters its idle state, in which it is neither processing a client request nor being upgraded. Upon receiving a client request (clientReq), it enters the processing state and once processing the request is completed, i.e., the done condition is satisfied, it returns to the idle state. Upon receiving an upgrade request (upgrReq), the upgrade process is initiated by setting the triggered flag to true and entering the idle_upgrade state. While in this state, the replica awaits its turn to be upgraded, however it may also enter the processing state whenever a clientReq is received. Once the upgrade_enabled condition is satisfied, a new replica starts and this replica enters the upgrading state.

The upgrade_enabled condition is a conjunction of two basic tests:

1. Is it this replica's turn to start the actual upgrade?

2. Can this replica be upgraded at this moment?

The former test can be realized by ordering all the replicas in the server group and checking whether the replica is the smallest/greatest in this group. An example of such an order is an order relation defined on replica identities within the group. The second test is realized through checking whether the current redundancy level is greater than l, were $l > 1$ must be satisfied to perform an upgrade. The enabling condition is evaluated periodically and once satisfied, the replica continues with the upgrade procedure.

The replica creates another process, whose task is to start a new replica to replace this one, and then enters the upgrading state, awaiting the success of the operation. The start_replica operation is designed so that it always successfully starts a replica in bounded time, even in the presence of transient failures. To achieve this property, the operation uses the recovery mechanism which is responsible for maintaining a given redundancy level. The upgrade process finally terminates and a new replica is successfully started

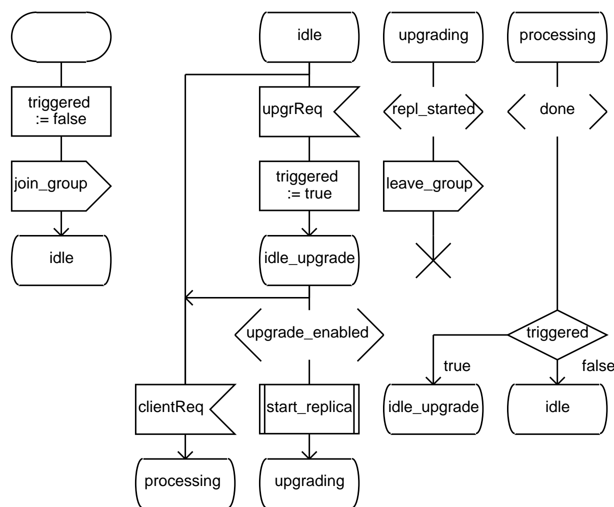

Figure 1. The upgrade algorithm from the viewpoint of a replica.

and joins the group. The GCS takes care of transferring the current state of the server group to the new replica, while this replica leaves the group and terminates. Considering the assumptions on input conformance and state transfer from Section 2.1, the new replica (version $v + 1$) enters a state in which it can produce output identical to that of replica version v, given the same input.

3.4 Brief Analysis

Below we reason informally, that the upgrade algorithm satisfies the requirements sketched in Section 3.1.

- The algorithm requires that there be a minimum allowed replication level $l > 1$, before a replica is replaced. Furthermore, if a replica cannot be upgraded it will continue to provide service using the old version. Thus, continuous availability is provided as there are replicas capable of processing client requests at any moment during the upgrade process.
- System consistency is maintained by the state transfer mechanism provided with the GCS. This is invoked for each upgraded replica. Note that we assume that state transfer can be achieved across different versions of the replica, as stated in Section 2.
- The algorithm is fault-tolerant in that the algorithm coordination is decentralized and it tolerates replica crashes. As there is no single entity that controls the progress of the algorithm, the upgrade continues even in presence of crashes of the replicas being upgraded. The recovery mechanism provided by the GCS allows

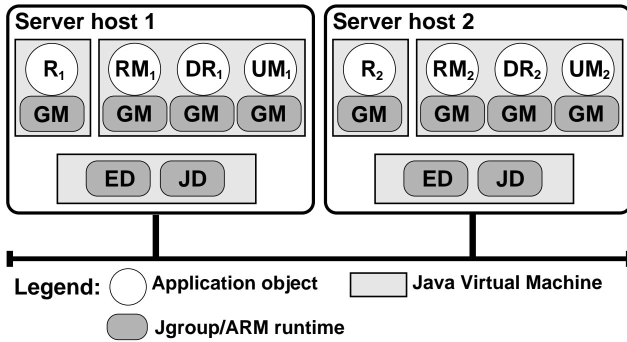

Figure 2. The architecture of Jgroup/ARM.

recovery from replica crashes by instantiating a new copy of the replica.

- At any time during the upgrade only one additional replica is added to the group, thus we keep the number of replicas in the system to a minimum.

Note that our algorithm by itself does not guarantee maintaining the redundancy level. To maintain a given redundancy level for the group, also outside the upgrade phase, we apply additional supervising mechanisms such as those provided by the ARM framework, discussed further in the following.

4 Implementation

The dynamic upgrade algorithm described in this paper has been implemented using Jgroup/ARM [6, 5], and aims at demonstrating its usefulness in terms of system availability. The implementation extends the ARM framework, that supports autonomous replication management, with a framework for dynamic upgrade. Figure 2 shows the core components of the Jgroup/ARM framework and its extensions to support dynamic upgrade. A brief description of the core components are given below.

- An *Execution Daemon* (ED) must be running on all hosts in the system that should be able to host application replicas. The execution daemon is used by the replication manager to create and remove replicas on remote hosts.

- *Replication Manager* (RM) is the main component of the ARM framework and its tasks include, replica distribution, failure recovery and interaction with client management applications through the replication manager interface. This component is, as shown in Figure 2, replicated for fault tolerance.

- *Upgrade Manager* (UM) effectuates upgrade group requests, communicated to it by an upgrade management client. It is naturally co-located with the RM to exploit its database of available groups.

- *Dependable Registry* (DR) is a replicated naming service. It enables a dynamic set of replicated remote objects to register themselves under the same name, forming an object group, which can later be retrieved by clients. This enables clients to communicate with the whole group as a single entity. Also the DR is co-located with the RM, since the RM depends on DR for bootstrapping.

- *Application Replica* (R) provides the actual service functionality that may be upgraded. The application replica may make use of various services provided by Jgroup by specifying a layer stack, as we discuss next.

```
<Application name="UpgradableServer" group="103">
  <Class name="test.upgrade.UpgradableServer" args=""/>
  <LayerStack order="PGMS:EGMI:Recovery:Upgrade"/>
  <RecoveryStrategy name="KeepMinimalInPartition">
    <Redundancy initial="3" minimal="1"/>
  </RecoveryStrategy>
</Application>
```

Figure 3. Example application specification for Jgroup/ARM.

4.1 The Jgroup Group Manager

The Jgroup *Group Manager* (GM) supports dynamic creation of group communication layer stacks, based on a layer stack ordering string associated with each application. The configuration of the layer stack can be expressed in XML, as shown in Figure 3, allowing each application to be configured according to its needs for various Jgroup services, such as recovery, upgrade, group membership and group method invocation services. Each GM layer may interact with any other GM layer, through an interface that each layer exports within the stack.

The *Jgroup Daemon* (JD) implements the basic group communication facilities such as failure detection, group membership and multicast, and each application specific GM layer may also communicate with the JD component to perform its tasks.

As shown in Figure 3, the UpgradableServer application use the PGMS, EGMI, Recovery and Upgrade layers. The PGMS is the group membership service provided with Jgroup; it supplies application replicas with information about the current view of the object group. The EGMI layer is an external group method invocation service, enabling clients to communicate with the entire object group as if it was a single entity. This means that the UpgradableServer will export an interface to its clients, enabling them

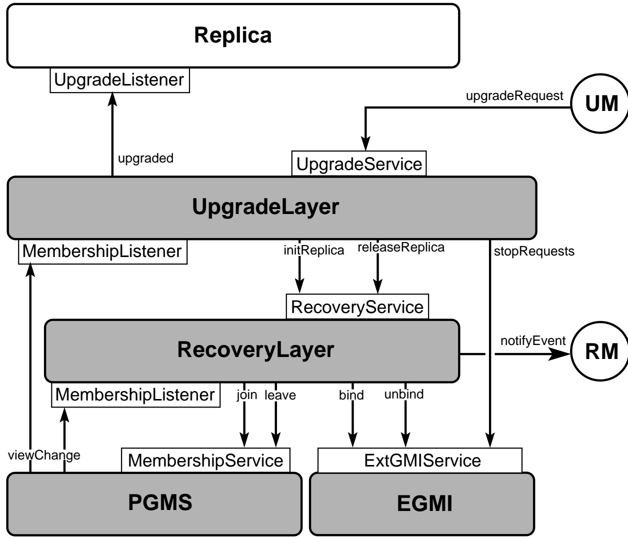

Figure 4. Upgrade layer stack.

to invoke the server group with what the client sees as a single method invocation. The RecoveryLayer also used in the example is a group manager layer that is part of the ARM framework. It is used in conjunction with the RM to ensure that all applications maintain a minimal redundancy level, as specified in Figure 3.

4.2 The Upgrade Layer and Interactions

The UpgradeLayer, the last component of the layer stack in the example above, implements the actual upgrade algorithm as described in Section 3.3. Figure 4 illustrates the layer composition and interfaces supported by the upgrade layer. For an application replica to be upgraded, it must implement the UpgradeListener interface (the upgraded() method.) The upgraded() method is used by the upgrade layer to notify the replica that a new version *has been* installed, and that the replica may now gracefully shutdown.

Prior to upgrading a particular application, it must first have been installed through the replication manager (RM). Figure 5 illustrate the main interactions of an upgrade. The actual upgrade is initiated by the *Upgrade Management Client* (UMC), by performing an upgradeGroup() invocation on the UM (①), which in turn leads to a upgradeRequest() (②) multicast invocation on the respective upgrade layers of the group to be upgraded. Next, the upgrade layers of the replicas decide if its their turn to be upgraded; in this case, R_1 is selected for upgrade and the UL performs a createReplica() (③,④) invocation on the local execution daemon. This in turn causes the newly created replica (new software version) to join the group, and thus all replicas (both new and old) install a new view (⑤). Once

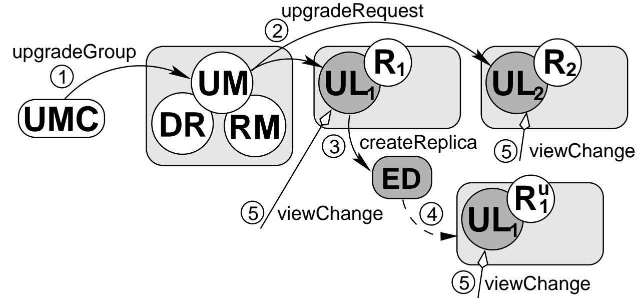

Figure 5. Interactions involved in an upgrade.

the UL representing the upgraded replica detect the new version (R_1^u), it will make the old replica leave the group; once it has left, the upgraded() method is invoked on R_1.

One of the main tasks of the upgrade layer is to determine which of the application replicas needs to be upgraded next, following the generic algorithm in Section 3.3. As shown in Figure 4, the upgrade layer will listen for viewChange() events from the PGMS (see also ⑤ in Figure 5). The replica to be upgraded next is determined on the basis of the replica positions in the view, e.g., the first member of the group will be upgraded first and so on. The view originated from the PGMS provides a list of the current group member identifiers. In order to implement the upgrade layer, we have extended the member identifier with a software version number. This is used by the upgrade layer to distinguish between replicas running the old software from replicas running the new software, within the same view.

To prevent client requests from being processed by the replica during an upgrade, the upgrade layer interacts with the EGMI layer, as indicated by the stopRequests() method. This is required to prevent returning results to clients while being upgraded.

5 Related Work

The topic of upgrading software entities at runtime has been appearing in the literature from many perspectives [3, 4, 9]. The unit of upgrade considered in this research ranges from a single operation to functions, programs and even distributed subsystems. The previous work differs from our algorithm, mainly in that they focus on upgrading non-replicated software entities. In our approach, the unit of upgrade is a replicated object and we focus on the availability characteristics and dependability aspects of the upgrade process. Eternal Evolution manager [10] supports live upgrades of actively replicated objects using an approach similar to ours. The target of an upgrade may comprise a set of CORBA objects, both clients and servers.

The upgrade proceeds by replacing single replicas in two phases, while the object group as a whole remains operational for the duration of the upgrade. The first phase involves an intermediate version, used to allow additional flexibility in the permitted changes. This, in contrast to our one-phase upgrade algorithm, is achieved through additional complexity.

6 Conclusions

We have presented an algorithm that supports upgrading an actively replicated server so that it is operational during the upgrade process. The upgrade process is transparent to the rest of the system and does not need human interaction as it is dependable. The algorithm makes use of an underlying GCS, in particular the group membership service and a reliable total-order multicast, to ensure dependable upgrade. Furthermore, our algorithm allow clients to seamlessly communicate with the replicated server group, even during the upgrade phase.

The upgrade algorithm has been implemented using the Jgroup [7] Group Communication System in conjunction with the Autonomous Replication Management framework [6]. The current implementation covers the actual algorithm, while it remains to implement a state transfer mechanism for our upgrade algorithm. This is required for stateful applications, to ensure that the newly upgraded replicas maintain the state of the old replicas, so that clients perceive a consistent state.

The presented mechanisms are part of the *Dynamic Upgrade Management Framework* aiming at supporting and managing dependable upgrades of distributed systems on the fly. Currently, we are working on the design of the management facility that will be responsible for managing multiple upgrades in the target system. Its task is to coordinate the upgrades in all the dimensions of the upgrade management space (time, range, multiplicity, atomicity and source of upgrade initiation) and validate that the upgrades were successful; and if not effectuate countermeasures to ensure continuous availability.

7 Acknowledgements

The authors would like to thank Prof. Oddvar Risnes (Telenor R&D Trondheim) for partial financial support to undertake this work. We would also like to thank Sune Jakobsson, Erik Berg and Prof. Bjarne Helvik for comments on our work.

References

[1] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[2] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33(4):1–43, Dec. 2001.

[3] D. Gupta, P. Jalote, and G. Barua. A Formal Framework for On-line Software Version Change. *IEEE Transactions on Software Engineering*, 22(2):120–131, Feb. 1996.

[4] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov. 1990.

[5] H. Meling and B. E. Helvik. ARM: Autonomous Replication Management in Jgroup. In *Proc. of the 4th European Research Seminar on Advances in Distributed Systems*, Bertinoro, Italy, May 2001.

[6] H. Meling, A. Montresor, Ö. Babaoğlu, and B. E. Helvik. Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management for Dependable Computing. Submitted for publication to IEEE Transactions on Computers, special issue on Reliable Distributed Systems, Feb. 2002.

[7] A. Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems*. PhD thesis, Dept. of Computer Science, University of Bologna, Feb. 2000.

[8] F. B. Schneider. Replicated Management using the State-Machine Approach. In S. Mullender, editor, *Distributed Systems*, chapter 7, pages 169–198. Addison-Wesley, second edition, 1994.

[9] M. Segal and O. Frieder. On-the-fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, pages 53–65, Mar. 1993.

[10] L. A. Tewksbury, L. E. Moser, and P. M. Melliar-Smith. Live Upgrade Techniques for CORBA Applications. In *Proc. of the 3rd Int'l Working Conference on Distributed Applications and Interoperable Systems*, Krakow, Poland, Sept. 2001.