Towards Upgrading Actively Replicated Servers on-the-fly

Marcin Solarski [†] Hein Meling [‡]

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.

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. 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) [3]. 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.

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. 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 describe the algorithm and concludes with a brief analysis. Section 4 discusses other work that relates to our upgrade algorithm. Finally, Section 5 concludes the paper and presents our ongoing work to validate the algorithm.

2 System Model and Assumptions

We consider a client-server architecture in an asynchronous distributed system augmented with unreliable failure detectors, 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 GCS [3] extended with a recovery mechanism [2]. 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.

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

[†]Fraunhofer Gesellschaft, FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany, Email: solarski@fokus.fhg.de

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

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 under the assumptions in Section 2.

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 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 replication level allowed.

Let 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. (3a) 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. (3b) 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.

3.1 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*.

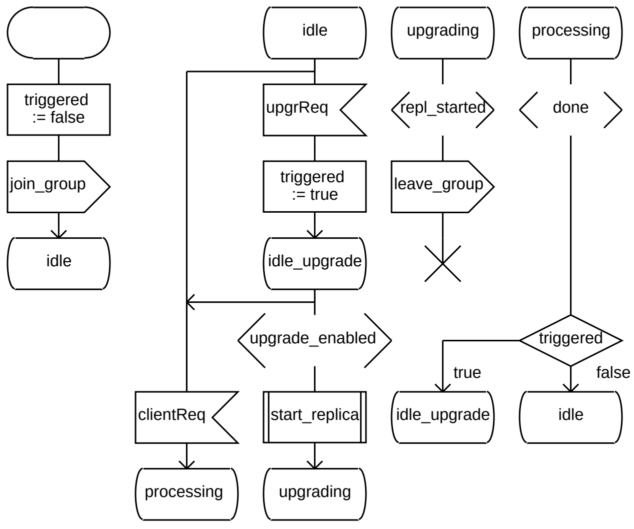

Figure 1. The upgrade algorithm from the viewpoint of a replica.

After initializing the replica (`triggered` flag is false) and joining the server group (`join_group`), the replica 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 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: (a) Is it this replica's turn to start the actual upgrade? (b) 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 replication 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, who is responsible for maintaining a given replication level. The upgrade process finally terminates and a new replica is successfully started 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, the new replica (version $v + 1$) enters a state in which can produce output identical to that of replica version v, given the same input.

3.2 Brief Analysis

The upgrade algorithm has the following features:

- 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 through the state transfer service provided by 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 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 replication level. To maintain a given replication level for the group, also outside the upgrade phase, we need to apply additional supervising mechanisms such as those provided by the Autonomous Replication Management (ARM) framework [2].

4 Related Work

The topic of upgrading software entities at runtime has been appearing in the literature from many perspectives [1, 4]. 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 [5] 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.

5 Conclusions

The presented algorithm supports upgrading an actively replicated server so that it remains 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.

Given the assumptions in Section 2, our upgrade algorithm allows a client to seamlessly communicate with the replicated server group, even during an upgrade. However, the assumptions also limit the algorithms applicability, since the same output must be provided, when given the same input to both version. Thus, if the output is incorrect, we cannot fix it without also replacing the client with a new version that makes use of a different method.

We are currently implementing the algorithm using the Jgroup [3] GCS in conjunction with a framework for Autonomous Replication Management (ARM) [2].

References

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

[2] 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.

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

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

[5] 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.