DYNAMIC EVOLUTION IN A SURVIVABLE APPLICATION INFRASTRUCTURE

Haraldur D. Thorvaldsson[1] Kenneth J. Goldman[1]
Department of Computer Science and Engineering
Washington University in St. Louis, St. Louis, MO 63130
{harri, kjg}@cse.wustl.edu

ABSTRACT
We describe a highly scalable algorithm for secure system evolution in an infrastructure for widely distributed Byzantine fault-tolerant applications. To maintain high availability, the system and its applications evolve on-line, providing uninterrupted service during installation of upgrades. Installations are made to appear atomic with respect to other installations and application execution steps. Our algorithm guarantees safe installation despite Byzantine faulty replicas and replica groups. An initial phase prepares replica groups for an upgrade, while a second phase triggers the installation of the upgrade by gossip among groups. A simple but novel scheme using secret sharing and Byzantine quorums prevents faulty replicas and replica groups from disrupting or maliciously exploiting installations. Installation message complexity and computational complexity grow linearly with the number of replicas.

KEY WORDS
Fault tolerant systems, Dynamic evolution, Configurable distributed systems, Byzantine fault tolerance

1 Introduction

Fault-tolerant systems use replicated servers to provide correct service even if a bounded number of their replicas fail. A *Byzantine faulty* [1] replica may behave arbitrarily, for example due to errors or attacker intrusion. We have proposed Survivable Workflow Transaction Infrastructure (SWFTI) [2], a decentralized architecture supporting Byzantine fault-tolerant application execution. Using recent algorithms for efficient Byzantine fault-tolerant replication of both passive [3] and active [4] application components, our infrastructure is designed to confer survivability on *long-running applications*, enabling them to make progress despite Byzantine faulty replicas. Survivability greatly benefits critical infrastructure such as financial, health-care and avionics systems, but also business processing automation services of increasing importance, such as supply chain management and freight tracking.

A fault-tolerant system usually needs to provide high availability as well. SWFTI supports *dynamic evolution* [5], whereby service is maintained during the upgrading of applications and system software. This reduces maintenance downtime and expedites error repair. Since the execution model of SWFTI is based on atomic execution steps, an evolution execution step must appear atomic with respect to other evolution execution steps, as well as ordinary application execution steps. However, it is critical for the system's evolution mechanism to not afford intruders new opportunities for attack. Our challenge, therefore, is to *dynamically and atomically update a large number of distributed replicas in the presence of Byzantine faults*.

The contribution of this paper is an efficient and scalable solution to this problem that ensures safety, liveness and security for dynamic upgrade installations. Even if the bound on faulty replicas tolerated has been exceeded in some number of replica groups, the algorithm guarantees safe installation in the correct groups. This form of *fault isolation* is important, since we expect fault-tolerant and non-fault-tolerant (i.e. non-replicated) groups to coexist in our infrastructure. The algorithm is decentralized and does not depend on a single coordinator, increasing its robustness and scalability.

The remainder of the paper is organized as follows. Section 2 discusses related work in dynamic evolution. Section 3 presents the system model and a simple, baseline solution. Section 4 gives a scalable algorithm that safeguards upgrades if each group is correct, even if some of the upgrade coordinators are faulty. Section 5 augments the algorithm so that upgrades are safeguarded in correct groups even if the bound on the number of faulty replicas has been exceeded in other groups.

2 Related Work

Existing work in dynamic evolution falls into two main categories: (1) upgrading of local processes, through transformation or migration of their volatile state [6, 7, 8, 9, 10] and (2) upgrading of distributed systems [5, 11, 12, 13, 14, 15, 16]. Our scope is the latter, although many techniques for local processes are applicable to distributed settings. Aside from communication and language integration issues, the distributed work focuses on safeguarding the consistency and correctness [5, 17] of applications from race conditions during upgrade installations. A common theme in these early approaches and many later ones [18, 19, 20] is to *quiesce* the servers or communication links affected by an upgrade, by halting them and rejecting or delaying operation calls until installation is complete. However, quiescing a large number of servers is disruptive, since most or all ap-

[1]Supported in part by National Science Foundation grant number 0305954.

plication processing is halted on all servers at least until the final server enters its quiescent state.

Karamanolis and Magee [21] describe a group-multicast-based protocol for adding and removing replicas from replica groups. Our algorithm is more general, but simpler, leveraging our infrastructure's support for atomic operations. Eternal [22] dynamically evolves replicated (non-Byzantine) fault-tolerant CORBA servers. Replicas are restarted one by one in a hybrid version of the old and new software, after which they atomically switch to the new. Replicas are quiesced before the switchover using a protocol that resembles distributed commit [23]. A significant drawback is that distributed commit does not scale well (see Section 3). Since we want to support atomic evolution of large numbers of replica groups, a different approach is required.

Our algorithm does not quiesce nodes, but rather executes installations in a way that makes them appear atomic [24, 25, 26]. Aborted installations have no effect, so an installation attempt never leaves the system in an intermediate state. Many of the non-transactional approaches do not adequately address failures during installation. To our knowledge, no prior dynamic evolution system tolerates Byzantine failures.

In addition to being robust, atomic dynamic evolution also makes it easier for developers and system administrators to reason about distributed upgrades, since they appear like steps in a simple, sequential execution. Furthermore, when the caller(s) and implementer(s) of an evolving interface are upgraded together atomically there is no need to develop "wrapper" software [19, 27] to translate between old-version and new-version calls. SWFTI does allow such wrappers for continued support of external system clients, if only to inform them about the deprecation of old interfaces. However, using them as a means of obtaining logically atomic installation [25] requires significant development effort and ingenuity.

A limitation of the atomic approach is its need for application co-operation to define transactional boundaries. We note that the non-transactional systems also require some co-operation to determine execution points at which installations may take place. These may be explicit reconfiguration points in code [9, 14] or implicit ones such as object instance creation time [28] or run-time detection of states of stack unwinding [7, 8, 10], with the concomitant overhead.

A second limitation is the scalability of atomic operations, in particular when implemented using locking for concurrency control. This paper addresses that issue, by eschewing distributed transactions in favor of an algorithm that commits installations using Byzantine fault-tolerant diffusion between replica groups. Our basic approach is related to that of Boyapati et al [26], for lazily propagating installation between objects, but is novel in the methods used to efficiently tolerate faults.

When an upgrade requires application state to be mapped from an old version to a new one, we assume the upgrade provides *transformation functions* for that purpose, which may operate on persistent state [10, 25], volatile state [26, 9, 22] or both. The transform functions may be executed eagerly or lazily. While our algorithm doesn't simultaneously halt the entire set of evolving replica groups, each individual group has to halt while transforming its local state, if it does so eagerly. Therefore, a lazy approach may be suitable for replicas with large persistent states. The details of how our upgrades are carried out locally are similar to the existing work.

3 Baseline approach

This section briefly describes the relevant parts of our system model [2]. It then describes a baseline evolution protocol using distributed transactions. While this protocol is workable for small installations, its main purpose is to serve as a correctness definition for our scalable algorithm.

3.1 System model

SWFTI consists of groups of deterministic replicas [29] that use a Byzantine agreement protocol [4] to ensure that all correct replicas take the same execution steps and maintain consistent states. We assume asynchronous communication with the relatively weak condition that the delay for message delivery does not grow faster than real time [3]. A replica group of size n tolerates the Byzantine failure of no more than $f = \lfloor (n-1)/3 \rfloor$ replicas, the known lower-bound [30]. A *correct* replica executes its application and agreement protocol according to specification. A replica that is not correct is *faulty* and may behave arbitrarily. For example, it may return erroneous results or even seek to undermine system function, possibly in collusion with other faulty replicas. Similarly, a replica group whose bound of faulty replicas has not (has) been exceeded is called a *correct* (*compromised*) group.

The application components in a SWFTI infrastructure are passive, persistent *objects* and active *transactions*. Objects are deterministic state machines that passively wait for their operations to be invoked, whereas transactions actively observe and update the state of objects, by invoking non-mutating *accessor* operations and mutating *mutator* operations on objects. Each component belongs to an *application*, a logical collection of objects and transactions owned and controlled by some principal, the application's *deployer*. Each application is executed by a set of replica groups. Each replica belongs to exactly one group and each replica of a particular group runs on a separate physical host machine. Each group executes (a part of) a single application. Two groups are *neighbors* if a transaction running in one of the groups reads or writes an object in the other group.

The state of the distributed system can be roughly partitioned into *application state*, that changes as application transactions execute, and *configuration state*, that describes the components of each application: their connections, replica groups, software implementations and physical hosts. The configuration is changed through the *installation* of *upgrades*, that specify changes to any of the above (including, possibly, changes to application state). The SWFTI execution model stipulates that an installation occur as an execution step that is atomic with respect to other installation execution steps and ordinary application execution steps. Thus, application transactions in the distributed system never "observe" a partially completed installation; it appears to start and complete in an indivisible instant of time. Furthermore, installations that abort appear as if they never took place.

3.2 Evolution steps and correctness

Let Δ be an upgrade that affects some subset of applications in the system. Let G denote the set of replica groups running these applications which are affected by Δ and R the set of replicas in the groups of G. An upgrade is *valid* if it is well-formed and duly authorized for the changes it contains. We omit the details here, but assume that (1) for each replica $r \in R$ there is a public / private [31, 32] key pair (r_u, r_p), where r_p is known only to r but r_u is known to everyone, and that (2) each correct replica r of each group $g \in G$ is able to securely verify the validity of Δ_g, the *piece* of upgrade Δ that affects g. We require the following for the installation of Δ:

Safety. *A correct replica of a correct group in G installs Δ if and only if all correct replicas of all correct groups in G install Δ.*

Liveness. *If Δ is a valid upgrade then eventually at least one correct replica in a correct group in G will install Δ.*

3.3 Transactional Installation

A straightforward way to satisfy our safety and liveness conditions is to store the configuration within the system itself in special system objects. Evolution execution steps can then execute as ordinary atomic Byzantine fault-tolerant operations on these objects, with safety and liveness guaranteed by our Byzantine agreement protocol. We designate for each group $g \in G$ a *group object* o_g, that stores the configuration of g and its application. Object o_g has an operation $Install(\Delta_g)$, accepting an upgrade piece for g as its parameter. An additional benefit of this approach is that evolution can be secured using the SWFTI distributed security system (not discussed here), the same as ordinary application operations.

If multiple groups are affected by an upgrade, the installation could be executed within a distributed transaction, leveraging the system's support for atomic execution. This would simplify installation considerably, obviating the need for node quiescence or complex synchronization protocols. However, a distributed evolution execution step that upgrades a large number of components may fail to ever commit, suffering aborts at some group(s) due to time-outs, before acquiring exclusive access to other groups. Even for a single group, upgrading a large data object in an atomic transaction may render the object unavailable for a long time, delaying or forcing the abort of a large number of other transactions. This applies in particular to operations that add new replicas to a group, since the replica's state must be included in the upgrade and/or acquired from other replicas, which may be time-consuming. Therefore, in the next two sections, we describe an installation algorithm that scales to large numbers of replicas and maintains high application availability while still ensuring safety and liveness.

4 Scalable, diffusing installation

This section describes our algorithm for overcoming the limited scalability of transactional installation. It proceeds in two phases. In the *preparation* phase, groups are "primed" for their upgrade by providing them with their upgrade pieces. Once all groups acknowledge their readiness, the installation is triggered in a *commit* phase, that diffuses through the system, piggybacked on messages between groups, such that the installation appears atomic.

4.1 Installers and pieces

The installation of an upgrade is coordinated by *installers* executing as survivable applications within SWFTI. We opt not to trust installers, though, since that would make them a highly valuable target for attackers. A compromised installer could, for example, only prepare a subset of the groups and then trigger the commit phase, violating safety. Our algorithm ensures this cannot happen. However, it cannot preclude a faulty installer from violating liveness, by not completing the preparation phase or not triggering the commit phase. For this reason, we allow multiple installers to work concurrently on an installation, guaranteeing liveness as long as at least one installer is correct. A secondary advantage is that concurrent installers can potentially complete an installation in a shorter time than can a single installer.

The deployer of an upgrade Δ splits the upgrade into upgrade pieces, one for each group $g \in G$. It encrypts piece Δ_g of group g using a freshly generated symmetric key k_g and gives the encrypted piece to the installers. This encryption prevents installers from gleaning information from pieces. For each replica r of each group g, the deployer provides the installers with $\langle k_g \rangle_{\sigma r_u}$, i.e. key k_g encrypted with r's public key, which enables r (and only r) to later decrypt k_g using private key r_p. The deployer grants the installers authorization to call $o_g.PrepareInstall(\Delta_g)$ on each group object o_g, which is an operation whose return value indicates whether group g *accepts* piece Δ_g for installation or *rejects* it. It also authorizes installers to call $o_g.CommitInstall()$ on at least one group object o_g, which is the operation that commits the installation on group g. Both operations are defined to be idempotent: if a group has already accepted, rejected or committed a piece, it returns the same value as the first call for that piece, without any change in the group's state. This allows multiple

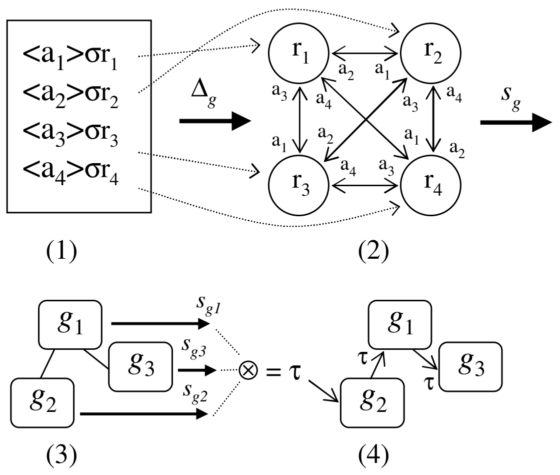

Figure 1. The main steps in installation, (1) group replicas accept their piece and decrypt shares, (2) group replicas exchange shares and reconstruct the group secret, (3) after receiving all group secrets, installers compute trigger code τ, (4) installers trigger installation, τ diffuses between groups.

installers to attempt to prepare the same piece at a group without ill effect. The installation completes without further involvement from the deployer's host, which is not a single point of failure and does not need to be survivable.

4.2 Installation safety

We must ensure that an installer can trigger the commit phase only after all groups have accepted their pieces. A straightforward method would be to have installers collect digitally signed responses from all $\mathit{PrepareInstall}()$ calls, and give these as proof of readiness to $\mathit{CommitInstall}()$ calls. This is burdensome, however, as each replica of each group would need to know about all other groups in G and their replica public keys, in order to verify the response signatures. Furthermore, the diffusion of commits would be hampered by having to include (and verify) on the order of $|R|$ signatures per message.

Our solution is for the deployer to create a *trigger code* that acts as a "password" for the commit phase which installers can only obtain once all groups are prepared. Figure 1 summarizes how correct replicas and installers recover the trigger code and then use it to trigger an installation. The deployer creates the trigger code, τ, as follows. For each group $g \in G$, it generates a large[2] random integer s_g, called the *group secret code* of g, and includes it in upgrade piece Δ_g. The deployer computes τ for the overall installation as $\bigoplus_{g \in G} s_g$, where $\oplus$ denotes the binary exclusive-or operator. Finally, the deployer computes a cryptographic digest $D(\tau)$ of τ, and includes it in every installation piece.

Group object o_g returns group secret s_g from a $\mathit{PrepareInstall}()$ call only if it accepts the piece offered, and only executes $\mathit{CommitInstall}()$ for that upgrade if the correct trigger code τ is passed in as a parameter. It verifies an incoming trigger code by computing its digest and comparing with the $D(\tau)$ value from its piece. Since an installer can only compute τ once it has obtained the secrets of all groups, it can only trigger the commit phase once all groups are ready, as required.

The algorithm as described has a vulnerability: a faulty installer can collude with a faulty replica r' of some group g to obtain s_g directly from r'. Hence, it can trigger the commit phase without ever preparing g, violating safety. Therefore, instead of placing s_g directly into Δ_g, the deployer breaks s_g into *shares*, one share a_r for each replica r of g, using a (k, n) secret sharing scheme [33]. The scheme is set up so that at least $k = f + 1$ shares out of $n = 3f + 1$ are needed to reconstruct s_g, where f is the maximum number of faulty replicas tolerated by g. The (at most f) faulty replicas of a correct group cannot conspire to erroneously reconstruct and release s_g, nor can their refusal to release shares prevent the (remaining $2f + 1$) correct replicas from reconstructing s_g and accepting a piece.

The deployer includes $\langle a_r \rangle_{\sigma r_u}$ in each piece Δ_g for each replica r of g, that is: the share of r encrypted with the public key of r. To prevent a faulty replica from passing forged shares to its peers, the deployer also includes in each Δ_g a mapping from each replica r of g to the digest of that replica's share. Correct replicas compute the digests of shares offered to them and discard shares whose digests don't match those from the map. We could alternatively use cheating prevention schemes [34, 35], but they are not strictly required since we trust the deployer (dealer) with respect to creating shares for the deployer's own upgrades.

An installer can only commit upgrade Δ in a group after obtaining all group secrets, meaning that all groups will commit Δ. If at least one installer is correct, then eventually all groups will prepare (through the actions of the correct installer and possibly some of the other installers as well) and the correct installer may then trigger the commit. Recall that in this section we assumed all groups in G are correct, since a compromised replica group *can* violate safety in this version of the algorithm. We enhance the algorithm to handle that case in Section 5.

4.3 Atomic, diffusing installation

Once a correct installer has computed the trigger code τ for an upgrade Δ, it calls $o_g.\mathit{CommitInstall}(\tau)$ for at least one $g \in G$. The next time a group g that has just committed Δ communicates with a neighboring group g', it diffuses the installation to g' by piggybacking τ on its message, to prove that Δ has committed (g' simply ignores τ if $g' \notin G$).

The pseudo-code in Figure 2 illustrates the behavior of a group when making and fulfilling requests. If a group learns about the commit of one of its pending upgrades via an incoming request, it immediately commits the installation of the upgrade and subsequently services the request

[2]Large enough to resist guessing, e.g. 128 bits.

```
$T_g \equiv$ trigger codes destined for group $g$, initially $\emptyset$
$P \equiv$ prepared trigger codes in this group, initially $\emptyset$
$C \equiv$ committed trigger codes in this group, initially $\emptyset$
**MakeRequest(**to-group $g$**)**
    $\Theta \equiv$ this transaction (about to make a request to $g$)
    Send trigger code set $T_g$ with request to $g$; $T_g \leftarrow \emptyset$
    Wait for $g$'s response, containing a set $R$ of trigger codes
    $R_{new} \equiv (R \cap P) \setminus C$
    If $R_{new} \neq \emptyset$ then
        For each neighboring group $g' \neq g$: $T_{g'} \leftarrow T_{g'} \cup R_{new}$
        Abort($\Theta$); Commit piece of each $\tau \in R_{new}$; Restart($\Theta$)
**ReceiveRequest(**from-group $g$, with trigger code set $R$**)**
    $R_{new} \equiv (R \cap P) \setminus C$
    If $R_{new} \neq \emptyset$
        For each neighboring group $g' \neq g$: $T_{g'} \leftarrow T_{g'} \cup R_{new}$
        Commit piece of each $\tau \in R_{new}$
    Execute request; Return with $R = T_g \setminus R_{new}$; $T_g \leftarrow \emptyset$
```

Figure 2. Group making and serving requests.

through execution of the new, upgraded software. Since our algorithm aborts transactions that learn about installation commits from request return values, the execution appears as if that case never occurs.

Our algorithm ensures that commits are linearizable [36] with one exception: an external client $c \notin G$ could invoke requests on two groups in G where only one of them has committed the upgrade. To prevent this, groups include in all their request responses a *group version* v, that changes with each committed group upgrade. Clients store the version of each group they call and pass it along with their requests to that group. A group receiving an outdated version v' returns back the current version and the trigger codes of all upgrades that the group has installed since v' was current. The external client passes the codes to other groups it calls (including those it has called already, e.g. as a part of the client transaction commit message), ensuring they commit their pending installations (if any) or abort the client transaction in time to ensure linearizability. Since a faulty external client can only trigger installations that are ready to commit, it can only violate the linearizeability of its own executions, by not forwarding trigger codes.

To reduce the delay from the beginning of a commit to its completion, we allow groups to proactively diffuse trigger codes by calling $CommitInstall(\tau)$ on their neighboring groups, in addition to the passive piggybacking on existing messages, as described.

4.4 Aborts, replay attacks and conflicts

A large organization deploying many applications and replica groups may delegate evolution authority to multiple sub-deployers. Unless these coordinate among themselves, they may attempt to prepare upgrades on overlapping sets of groups. An upgrade Δ_1 by deployer d_1 may be rejected by a group g if, unbeknowst to d_1, another upgrade Δ_2 has been prepared at g and the two pieces *conflict* at g, so that Δ_1 would be invalid[3] if installed after Δ_2. If some piece of Δ_2 is rejected by a different group $g' \in G$, then neither upgrade can complete and the installation deadlocks.

To resolve this, installers must be able to abort installations. There may be other legitimate reasons for a group to reject a piece, for example, if the deployer created the upgrade based on outdated information about the group's configuration. However, installers must be prevented from violating safety, by committing installations in some groups while aborting them in others. To ensure safety, the deployer of an upgrade Δ creates and includes in all pieces a randomly chosen *cancellation code* c, broken into shares using the secret-sharing techniques of Section 4.2. It also includes the cryptographic digest of c in all pieces, to permit cancellation verification. A group returns c from $PrepareInstall()$ to reject a piece. This causes installers to call $o_g.AbortInstall(c)$ for each group $g \in G$, which makes the group abort the installation, discard its piece and diffuse the cancellation code to its neighbors. Since we assume in this section that all groups are correct, each group releases its group secret or the cancellation code, *but never both*. Therefore, the trigger code and cancellation code of an upgrade cannot both be reconstructed and all groups will either commit or abort a particular upgrade, preserving safety.

To prevent attackers from re-submitting old upgrades to groups, we require a deployer d to include in each piece Δ_g the trigger-code τ_d of the most recent piece d committed at g (or cancellation code c_d, if the upgrade was aborted), or else g will reject the piece. The sequence of these *version codes* establishes a total order on the upgrades submitted by a particular deployer to a particular group.

Aborts can break deadlocks, but installations of a pair of conflicting upgrades can still become *livelocked* if both upgrades keep getting aborted and deployers keep resubmitting them without ever being first to prepare in all the groups in the upgrade's intersection. The symmetry can be broken and livelock avoided if there exists a total[4] order $\prec$ on upgrades, and groups are allowed to postpone previously accepted pieces, as described in the pseudo-code of Figure 3.

If a group g receives a pair of conflicting upgrade pieces d_1 and d_2 in that order, g rejects d_2 right away if $d_1 \prec d_2$. Else, if $d_2 \prec d_1$, it waits until either one of them gets committed. There are two cases; either (1) d_1 is prepared first in every other group, so τ_1 gets reconstructed and Δ_1 installed, or (2) d_2 is first in at least one group, so c_1 is released and τ_1 cannot be reconstructed, while τ_2 does get reconstructed after all groups learn about the cancellation of Δ_1. An installer that receives a *conflict* result for an upgrade Δ continues with the installation but periodically retries $PrepareInstall()$ on the *conflict* groups, until either all replica groups have accepted Δ or some replica

[3]If two pieces do not conflict (e.g. they affect a disjoint set of objects), then they can be prepared concurrently and committed in either order.

[4]It only needs to be total for the set of upgrades currently being installed, not for all upgrades ever installed.

```
$A \equiv$ pieces this group has accepted, initially $\emptyset$
$C \equiv$ pieces this group has cancelled, initially $\emptyset$
$X(d) \equiv \{ d' \in A \mid d' \mathit{conflicts\ with}\ d\}$
**PrepareInstall(**valid piece $d_2$ of $\Delta_2$**)**
    If $d_2 \in A/C$ then return secret of $d_2$ / $c_2$, respectively
    For each $d_1 \in X(d)$
        If $d_1 \prec d_2$ then $C \leftarrow C \cup \{d_2\}$;
    If $\exists\, d_1 \in X(d) : d_1 \prec d_2$ then return $c_2$
    $A \leftarrow A \cup \{d_2\}$
    If $X(d) \neq \emptyset$ then return *conflict*
    Else return secret of $d_2$
**CommitInstall(**$t$**)**
    If $t$ is the trigger code for some $d \in A$ then
        Commit $d$
        $A \leftarrow A \setminus X(d)$; $C \leftarrow C \cup X(d)$
    Else if $t$ is a cancellation code for an upgrade $d \in A$ then
        $A \leftarrow A \setminus \{d\}$; $C \leftarrow C \cup \{d\}$
```

Figure 3. Group preparing and committing pieces.

group rejects it. Since we assume that at least one installer of each upgrade makes progress, either event is guaranteed to eventually occur.

One way of establishing $\prec$ is for deployers to obtain signed upgrade sequence numbers from a central service endorsed by the top-level deployer. One could imagine a decentralized alternative, where large numbers would be randomly self-assigned by deployers. To make selfish assignments of low numbers computationally expensive, the "random" number could be defined as the deployer's set of current version codes at each group in G, encrypted using the deployer's private key. Each group would decrypt the set and verify that its own code was present in it.

5 Tolerating compromised replica groups

Recall that a replica group whose bound of faulty replicas has been exceeded is called a *compromised* group. Relaxing the assumption that all groups in G are correct allows, for example, a compromised group to release both the cancellation and group secret codes for an upgrade. A colluding installer could then trigger the commit of the upgrade's installation at some groups while aborting it at others, violating safety.

If it is neccessary to tolerate compromised groups, the following modification to our approach can be made. The deployer does not give the code shares of a group g directly to the replicas of g. Instead, it distributes the shares to a group K_g of *guardian* hosts that it assigns to g, such that only one of the codes can be reconstructed as long as a bounded number of guardian hosts are faulty. The guardians can be drawn from the set R of evolving replicas, some other set of hosts at the deployer's disposal or both.

A group of guardians forms a type of Byzantine quorum system [37], which has low message and computational complexity. A guarded group can get the shares it needs in a single message round, since all guardians can be contacted concurrently and the guardians do not communicate amongst themselves. The per-group message complexity of $O(|K_g|)$ and minimal addition to piece data (roughly 20-30 bytes per guardian) means that *large guardian groups are practical*, larger than what may be practical for application replica groups in general. Assuming an overall proportion p of evenly distributed faulty hosts, the probability that a guardian is faulty and its guarded group is compromised is p^{F+1}. We note that the number of guardians in each guardian group can be tailored to the fault-tolerance level of the guarded group, e.g. by assigning a greater number of guardians to less fault-tolerant groups. We also note that if a deployer only releases the guardian identities at the start of an upgrade, the time period for an attacker to compromise a sufficient number of them is limited, as compared to compromising a well-known and slowly changing application replica group.

In our modified approach, the deployer chooses for each group $g \in G$ a set K_g of guardians, such that $|K_g| = 3F+1$, where F is the number of faulty guardians to tolerate. The deployer breaks the group secret code s_g of g and cancellation code c into shares using a (k, n) secret-sharing scheme with $k = 2F+1$ and $n = 3F+1$. The deployer includes in each piece Δ_g of each group g, for each guardian k of K_g, a tuple $(a_k, D(s_g^k), D(c^k))$. Here a_k is the network address of guardian k while s_g^k and c^k are k's share of s_g and c, respectively. A guardian k gets, for each group g that it guards, the pair (s_g^k, c^k) of its shares of the group secret and cancellation code of g, respectively.

When a group g wishes to accept or reject an upgrade, it sends out an operation request $\mathit{GetShare}()$ to each $k \in K_g$, passing in $D(s_g^k)$ or $D(c^k)$, respectively. If a guardian k has a share corresponding to the digest, it takes it as proof that g is calling and returns the share, otherwise it ignores the request. A correct guardian, however, only ever returns a group's group secret code share or cancellation code share, *never both*. If at most F of the $3F+1$ guardians are faulty, then (1) a group can always get enough shares of one type of code, since there are at least $2F+1$ correct guardians and (2) a compromised group cannot get enough shares to reconstruct both codes, since it would have to contact some two guardian sets of size $2F+1$. These would have at least one correct guardian in common, which would refuse to release both types of shares.

Guardians can also be used to accelerate commit or abort diffusion. Groups pass their reconstructed codes to their guardians which pass them on to their guarded groups. The topology and redundancy in connectivity of guardians and groups can be arranged as to ensure rapid and robust diffusion of codes throughout G.

6 Conclusions

We have presented an algorithm for dynamically, atomically, securely and efficiently upgrading a large number of distributed replica groups in SWFTI, an application infrastructure that tolerates Byzantine faulty replicas and compromised groups. Our algorithm confers the same level of survivability on upgrade installations as is provided for applications running within the infrastructure, while maintaining high availability.

We are currently implementing our algorithm, as part of the prototype implementation of SWFTI. This will enable experimental validation of its performance and scalability.

References

[1] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[2] H. D. Thorvaldsson and K. Goldman. Architecture and Execution Model for a Survivable Work Flow Architecture Infrastructure. Technical Report WUCSE-2005-61, 2005.

[3] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *3rd USENIX OSDI*, 1999.

[4] S. Pallemulle, I. Wehrman, and K. Goldman. Byzantine Fault Tolerant Execution of Long-running Distributed Applications. In *18th IASTED Paralell and Distributed Computing and Systems*, 2006.

[5] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[6] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *2nd ICSE*, pages 470–476, 1976.

[7] I. Lee. A Dynamic Modication System. PhD thesis, Department of Computer Science, University of Wisconsin, 1983.

[8] O. Frieder and M. E. Segal. On dynamically updating a computer program: from concept to prototype. *Journal of Systems and Software*, 14(2):111–128, 1991.

[9] M. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *ACM SIGPLAN*, pages 13–23, 2001.

[10] D. Gupta and P. Jalote. On-line Software Version Change Using State Transfer Between Processes. *Software - Practice and Experience*, 23(9):949–964, 1993.

[11] T. Bloom. Dynamic Module Replacement in a distributed programming system, PhD thesis 1983. Technical Report MIT/LCS/TR-303.

[12] M. E. Segal and O. Frieder. Dynamically updating distributed software: supporting change in uncertain and mistrustful environments. In *IEEE Conf. on Software Maintenance*, pages 254–261, 1989.

[13] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Trans. Softw. Eng.*, 11(4):424–436, 1985.

[14] J. Purtillo and C. Hofmeister. Dynamic reconfiguration of distributed programs. In *11th IEEE ICDCS*, pages 560–571, 1991.

[15] B. Swaminathan and K. J. Goldman. Dynamic reconfiguration with I/O Abstraction. *IEEE Symposium on Parallel and Distributed Proceessing*, pages 496–501, 1995.

[16] K. Moazami-Goudarzi and J. Kramer. Maintaining Node Consistency in the Face of Dynamic Change. In *3rd IEEE ICCDS*, page 62, 1996.

[17] D. Gupta, P. Jalote, and G. Barua. A Formal Framework for On-line Software Version Change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996.

[18] I. Warren and I. Sommerville. A model for Dynamic Configuration which Preserves Application Integrity. In *3rd Intl. Conf. on Configurable Distr. Systems*, page 81, 1996.

[19] H. Evans and P. Dickman. Zones, contracts and absorbing changes: an approach to software evolution. In *15th ACM OOPSLA*, pages 415–434, 1999.

[20] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A Dynamic Reconfiguration Service for CORBA. In *4th Intl. Conf. on Configurable Distr. Systems*, pages 35–42, 1998.

[21] C. Karamanolis and J. Magee. A Replication Protocol to Support Dynamically Configurable Groups of Servers. In *3rd Int'l Conf. on Configurable Distributed Systems*, pages 161–168, 1996.

[22] L. A. Tewksbury, L. E. Moser, and P. M. Melliar-Smith. Live Upgrades of CORBA Applications Using Object Replication. In *IEEE ICSM*, page 488, 2001.

[23] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the Eternal system. *Theor. Pract. Object Syst.*, 4(2):81–92, 1998.

[24] T. Bloom and M. Day. Reconfiguration and module replacement in argus: Theory and practice. *IEE Softw. Engineering Journal*, 8(2), 1993.

[25] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and Simulation: How to Upgrade Distributed Systems. In *USENIX HotOS-IX*, 2003.

[26] C. Boyapati, B. Liskov, L. Shrira, C. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *18th ACM SIGPLAN*, pages 403–417, 2003.

[27] T. Senivongse. Enabling Flexible Cross-Version Interoperability for Distributed Services. In *Intl. Symposium on Dist. Objects and Applications*, Edinburgh, UK, 1999.

[28] G. Hjálmtýsson and R. Gray. Dynamic C++ classes - A Lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conference*, 1998.

[29] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[30] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[31] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Trans. on Information Theory*, IT-22(6):644–654, 1976.

[32] R. L. Rivest, A. Shamir, and L. M. Adelman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[33] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[34] M. Tompa and H. Woll. How to share a secret with cheaters. *Journal of Cryptology*, 1(2):133–138, 1988.

[35] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *17th ACM PODC*, pages 101–111, 1998.

[36] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[37] D. Malkhi and M. Reiter. Byzantine quorum systems. In *29th ACM STOC*, pages 569–578, 1997.