System Architecture Description: Sip Session Replication

Author(s): erik.van.der.velden@ericsson.com

Version:2.21

1 Introduction

1.1 Purpose

The purpose of this document is to document some of the design decisions that were made during the course of the sailfin project.
This document is only **informative** and in no way normative.
It is likely to be outdated on some parts and also describes potential solutions that are not, and maybe never will be, implemented.

1.2 Scope

Although the title talks about Sip Session replication, the artifacts that are replicated do cover more then just the sessions. The replicated artifacts include:

- The SIP application Session (SAS)
- The SIP (protocol) sessions (SS)
- The dialog information; DialogFragments (DFes).
- The converged HTTP (protocol) sessions (CHS) (not covered directly in this document, but coordinated with the converged HTTP session FSD)

There are relations between these objects, in fact, they form a graph of related objects. This does have some implications for Sip Session Replication as is show later in the document.

The name (Sip) Session Replication, abbreviated as SSR, will be used in this document to cover replication all of the above artifacts Also, we will often be talking about sessions, when the proper nomenclature would be (replication) artifact.

1.3 Terminology

- **Session Availability**
  Also called **session retention.**
  The ability to serve traffic in ongoing sessions (even in the case of a failure).
- **SipSessionReplication (SSR)**
  The name or (Sip) Session replication is used in this document for all the activities surrounding the establishment of session availability.
- **SAS**
  Sip Application Session [jsr289]
- **SAS timer**
  The timer that controls the lifetime of the SAS. When the timer expires the application is informed and within this notification the application can request and

extension of the lifetime of the SAS. The container is allowed to reject such an extension.

- **Application Timer**
  The application can create multiple timers which can be periodic or one-shot. These timers are associated with the SAS.
- **SS**
  Sip Session, approximately represents an ongoing SIP dialog [jsr289]
- **DF**
  Dialog fragment. One dialog can be associated with multiple applications via SIP chaining. Sip chaining can be done inside of the container or request can leave the container and re-enter it in the context of the same dialog For each (re-)entry of a dialog a new DF is created. The DF refers to all the Sip Sessions of all the applications in an internal chain. A DF is uniquely identified by the callid, from-tag, to-tag and fragmentid.
- **DS**
  dialog. A collection of dialog fragments in one SIP session that are allocated to the same JVM, due to spiralling in SIP.
- **Converged HTTP session**
  A HTTP session that is related to a SAS.
- **Non-compromised state**
  The state in which session availability can be guaranteed in case of a single failure. In this state there are two copies of every SSR artifact and these artifacts are on different server instances.
- **Compromised state**
  After some events (e.g., a failure) some copies of artifacts are lost. This leaves the system vulnerable. When a failure occurs in the compromised state this might lead to loss of sessions (provided it happens on the wrong server instance).
- **Repair period**
  The period during which the system is in a compromised state.
- **Repair**
  The act of going from a compromised state to an non-compromised state.
- **Eager repair**
  A way of repairing where the system actively repairs, independent of external events. This will shorten the repair period at the cost of capacity.
- **Lazy repair**
  A way of repairing where the repair is driven by events on the session. These events can either be external events (e.g., a request is received) or internal events (e.g., a time-out). Such a mechanism will lead to a longer repair period, the length of which depends on the traffic and timer activity.
- **Replication**
  The act of making a replica copy, on the replication partner, of an artifact in the active cache.
- **Replication partner (replication destination)**
  All servers are ordered in a ring topology. The replication partner is the server instance that is a neighbor in a clock-wise direction. Replications flow from the

active cache on the replication source to the replica cache on the replication destination (the replication partner).

- **Active partner (replication source)**
  All servers are ordered in a ring topology. The active partner is the server instance that is a neighbor in a counter clock-wise direction. Replications flow from the active partner to the replication partner.
- **Eager replication**
  Also called **repair-under-load or forward repair**
  The act of replication of all the artifacts in the active cache.
- **Lazy replication**
  Replication of an active session when the partner did not contain the previous version of the session.
  This is accomplished by normal replication, where the partner did not have the replica copy, e.g., due to a previous failure.
- **Reactivation**
  The act of re-creating an active copy of an artifact on an instance based on a replica copy.
- **Eager reactivation**
  Also called **reverse-repair.**
  The act of copying the all artifacts that are owned by the instance, from replica cache of the replication partner to the active cache of the owning instance.
- **Lazy reactivation**
  Reactivation based on activity in the session (e.g., a request or a time-out).
- **Home instance**
  The instance where the load balancer will route traffic that is targeted to this artifact This is based on the current shape of the cluster.
- **Owner**
  The instance that did the last replication of the artifact is said to be the artifact owner.
  Not to be confused with the home instance. The home and the owner can be different, due to cluster reshapes.
- **Migration**
  The act of re-creating an active copy of an artifact based on another active copy on another instance.
- **Load request**
  The request to load a session for which a sufficient version is not available in the active cache. This can entail a broadcast in the cluster.
- **AS Upgrade**
  The act of replacing the AS with a newer version of the AS. In case of compatible versions of the AS, this should be done without the loss of sessions.
- **Application Upgrade**
  The act of replacing an application with a newer version of the application. In case of compatible versions of the application, this should be done without the loss of sessions.

- **Rolling Upgrade**
  An upgrade where the instances in the cluster are upgraded one-by-one.
- **Failure**
  A failure of a server instance. This will result in the loss of both the active and the replica cache on the server instance.
- **Recovery**
  The server instance recovers after a failure.
- **Shutdown (downscale)**
  A planned shutdown of a server instance.
- **Restart**
  A planned recovery of a shutdown instance.
- **Start (Upscale)**
  An addition of one or more new server instances to the cluster.
- **Load balancer (LB)**
  An entity that routes the traffic to the instances based on some load balancing policy.
- **(Sticky) Round-Robin (RR) policy**
  This is only applicable to HTTP.
  Initial requests are routed independently of the content of the request.
  Subsequent requests in the session are routed to the same instance as the initial request.
  If a request can not be routed to the instance (e.g., during a failure), the request is re-routed to another instance and the affinity is changed, i.e., from then on it remains sticky on the newly selected instance.
- **Data Centric (DC) policy**
  For historical reasons sometimes called User Centric (UC)
  This is applicable for both SIP and converged HTTP.
  Some parameters from the initial request are used to calculate a hash code. The request is routed according to a consistent hashing algorithm that allocates hash codes to the currently available server instances. The hashcode is included in any subsequent requests, all of which are routed based on the same consistent hashing algorithm. However, responses are routed to the same instance as the original request.
- **SIP transaction**
  As defined in [RFC3261]**Properly distributed**
  A situation where every artifact is active on its home instance.
  The advantage of such a situation (if it can be detected) is that every request arriving on an instance can be handled locally, without any load request in the cluster.
- Group Membership Service
  A service that keeps track of the active instances in the cluster.
- Temporary Failure
  A failure situation in which the instance has not yet been marked as failed/stopped by the GMS.

- Permanent Failure
  A failure situation where an instance has been marked as failed/stopped by the GMS.

A short summary in a table:

	Eager	Lazy
Reactivation	Reverse repair From replica (partner) to active (restored) based on restore event (only used for restart in case of upgrade)	From replica (any) to active (home) based on traffic or timer.
Replication	Forward repair / repair-under-load From active (owner) to replica (partner) based on cluster reshape. (only used in restart in case of upgrade)	From active (owner) to replica (partner) based on traffic or timer.
Migration	Not used	From active (owner) to active (home). Based on traffic.

2 Design Overview

2.1 General

The requirements are described in [FSD]. This document described the main design use cases.

2.2 Design principles

2.2.1 Lazy repair

After certain events (e.g., a failure or a shutdown) there will exist only one copy of some artifacts in the cluster. This puts the system in a compromised state, where it is vulnerable for events that lead to the loss of data (e.g., another failure, another shutdown). At least, this can happen if these events occur on the ‘wrong’ instance and remove the only remaining copy of the data.
After the repair period, the system again will have two copies of all the artifacts, and should be in a non-compromised state.

There are different strategies to handle such vulnerabilities. The opposite sides of the spectrum are:

- the **eager** strategy will try to ensure that the repair period (the period of vulnerability) is short by doing bulk copying of data to restore the state where there are two copies of every artifact

  The cost of this strategy is in the extra load on the system to repair the ‘damage’. In a failure situation, these costs might be too high, this might lead to rejected traffic and to instability, something that should be avoided in such a situation, where the system is already vulnerable.
- The **lazy** strategy relies on activity in the session and only then ensures there are two copies of the data created.

The disadvantage of this strategy is that the length of the repair period depends on the activities in the session (external traffic and/or time-outs). The period depends on the traffic mix can be quite long. Also, it is difficult to detect when the repair period is finished and we have restored a non-compromised state.
However, the heuristics for mean time between such events (e.g., mean time between failures) and the average time between activity on the sessions are expected to ensure that the chance of session loss is acceptably low.

Also, since the repair is done based on external traffic (e.g., requests), in the lazy scenario more requests will have a longer latency then in the eager scenario.

The decision is to normally do a lazy repair, based on the expected heuristics. However, in cases where this is not possible (specifically in the upgrade use case) the eager repair is used.

2.2.2 Healthy system

We will try to enforce some state, based on a best-effort basis.

- There will be zero or one active copy of a session (avoid duplicates).
  This is violated only in case of network segmentation or long term network failures.

There are also some nice-to-haves which can probably not be effectively achieved when using asynchronous and non-transactional replication.

- If the active copy is removed, the replica is also removed (no zombies).
- If a replica exists, it has the latest version (freshness).

For Further Study (FFS):
Because of some optimizations we might want to restore a properly distributed situation and be able to detect that this properly distributed situation is established.
A properly distributed system is characterized by the following:

- every active copy is on the home instance
- there is no replica without an active copy (all sessions are activated)

Effectively, this means that the load balancer will direct all requests to where the active copy is located, if there is any active copy. This in turn means that no data has to be fetched elsewhere in the cluster.
Note that a properly distributed situation is different from a non-compromised situation; the first allows active copies without replicas, the latter allows for active instances located at another instance than the home instance. Ideally, the cluster is both non-compromised (I.e., fully repaired) and the data is properly distributed.

An solution that avoids unnecessary load requests without first establishing a properly distributed situation is described in the section about expat-list handling.

2.3 Task Summary and division

This is a rough sketch of the work that has been identified to enable the SSR functionality with the performance as is required. It is a quite cryptic list and only gives a short overview on slogan level.

Ericsson

- Data modeling
  Decide which objects are replicated separately (e.g., will timers be separate objects, will DF be part of DS).Which fields do we want to replicate independently (e.g., Cseq, last accessed time).
  - Proxy refactoring
  - B2BUaHelper impact
  - JSR 289 uncertainty; extra impact from monitoring
- Overall design
- Baseline refactoring
  - Serialization logic & deser (SS/SAS/DS/DF/Timers/etc.).
  - Dirty checking (SAS/SS/DF/DS etc.)
  - save (propagated save)
  - DF refactoring (handle spiraling using unique DFids, reconstruct DS from DFes or remove DS from code)
  - proxy refactoring
  - B2BUaHelper impact
- Local session locking (protect SS/SAS from concurrent access internally, i.e., blocking and from background access, e.g., reaper)
- DF manager
  - CRUD invocations (refactoring of codebase to invoke the D~~S~~F manager)
  - replication trigger based on replication configuration of contributing applications.
- replication triggers
  - replication trigger layer.
    Implemented as a layer in the SIP stack. Invokes replication at the proper times in the SIP transaction
  - out-of-band triggers
    When invoked out of band (e.g., via the SipSessionUtil) changes made to the session should also trigger replication. In worst case this means that every setAttribute is replicated, but some optimizations can be imagined (e.g., definition of a unit-of-work, which allows multiple actions to be replicated together, or something similar to the WLSSAction from BEA).
- Performance test phase 1
- Expired timer handling & missed timer handling

- Remote session locking
- Didactivate & Willpassivate handling
- cleanup activities;
  - propagated loading of protocol children and timers after extensions of lifetime
  - removal of orphan objects.
  - Extra timer for refresh of SAS.
  - Reaper thread creation & handling
- Upgrade;
  - diff handling (both for active and replica, including keeping track of which copies have been repaired and special handling of not-yet-repaired copies, I.e, always doing a load for not yet repaired copies in active cache)
  - generation of list of ids & versions based on owner ?? Or SUN?
  - Save on disk and reload
- find vs create
  - changes to find method (after expat list is available; check for expat after find, call version aware load in case expat is requested).
  - Expat handling (check other instances for session belonging to home)
  - Special handling until expat list is received (always do a non-version aware load if SAS can not be found in active cache)
  - expat list creation (iterator over cache, doing a consistent hash on all items).
- Network segmenation detection
  - Base disconnect / network segmentation detection on heuristic approach and clear the caches or restart the instances if detected.

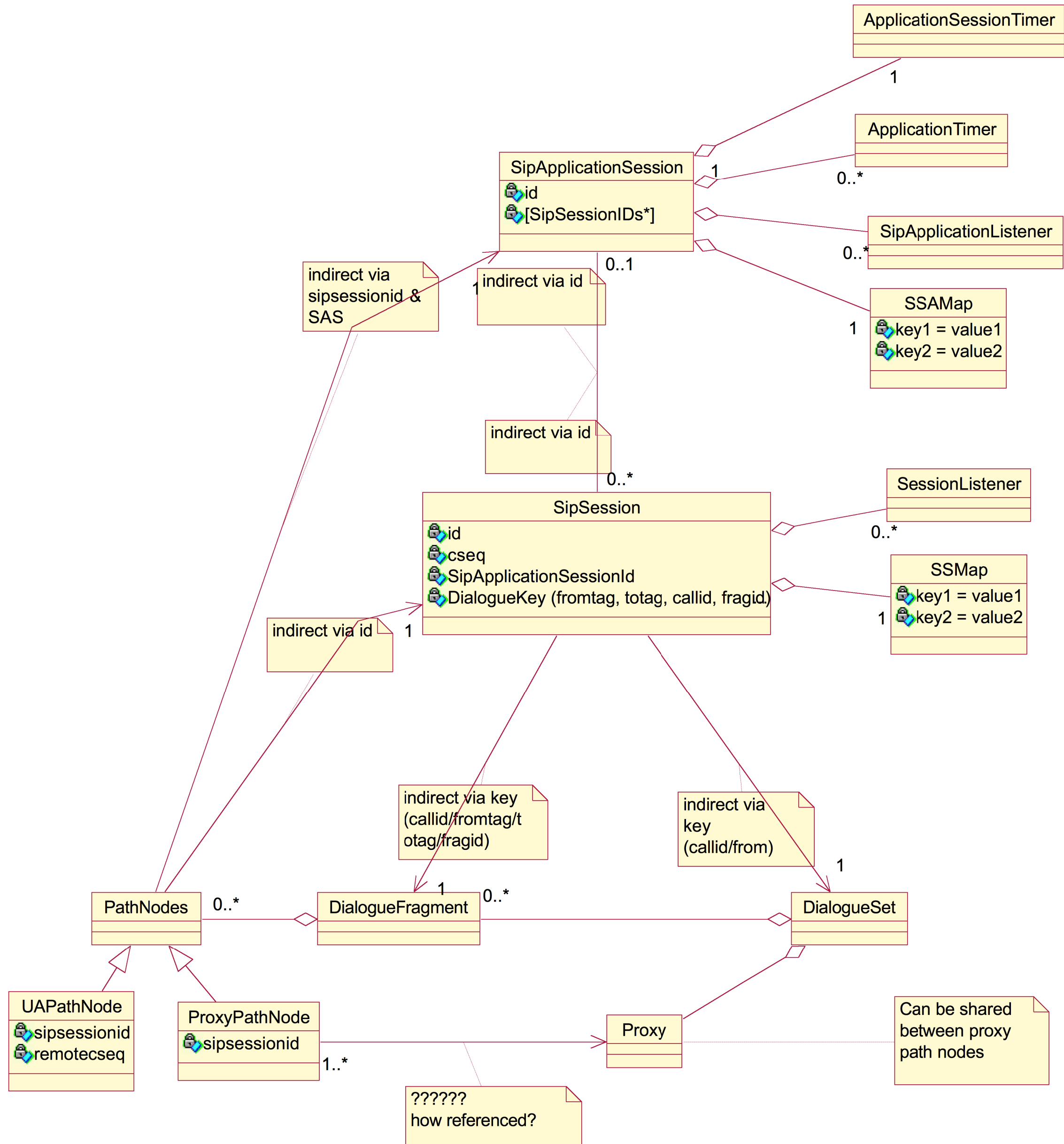
SUN

- Timer expiry on replica
  - peek functionality for staleness check
  - targetted load request
- load request changes
  - non-version aware load
  - 3 step delete of old copies (includes active migration)
  - NACK handling (avoids 4 seconds delay in non-version aware load request)
- configurable reverse repair (not actually required any more..)
- configurable repair-under-load
- configurable offload (already done?)
- upgrade
  - sending of list of ids & version
  - targeted replication request (or fetch of artifact)
- SS Manager
  - Timer refactoring
  - HTTP consistency handling
  - Template code
  - reference handling
- DF Manager
  - template code

- HTTP session manager
  - upgrade support (expat list handling is not needed here)
- JSR289 section 13
  - SAR referencing (ability to address a specific SAR based on the prefix of the SASid)
- find vs create
  - expat list sending and receiving

2.4 Data model

Globally, the Sip Container's data model looks something like this.



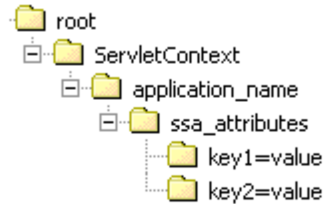
2.4.1 EAS 4.1 data model

In EAS 4.1 session replication [EasSsr] some work was already done on the data modeling for replication. This section is a summary of that work.

The data is split into 4 distinct, but related entities. Each entity has a separate, but somewhat dependant, lifecycle and replication is triggered by different events. The model was based on a JBoss cache, but most of the modelling is still applicable.

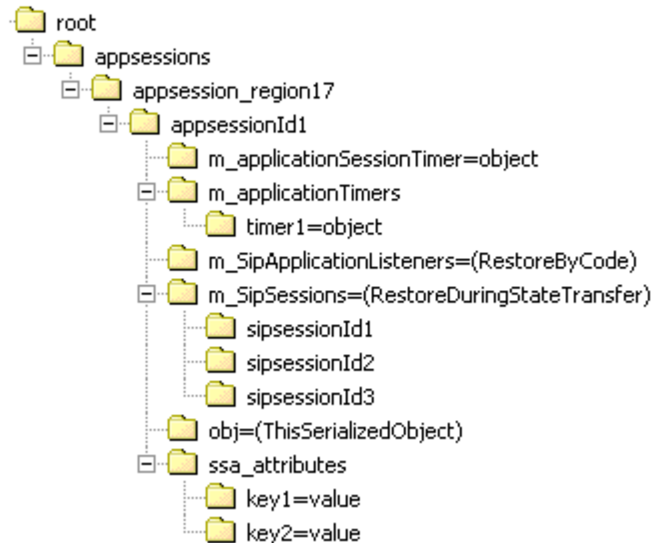
The replicated types defined in [EasSsr] are:

2.4.1.1 servletContext



The node for the application is created if the application has stored at least one attribute.

2.4.1.2 SipApplicationSession

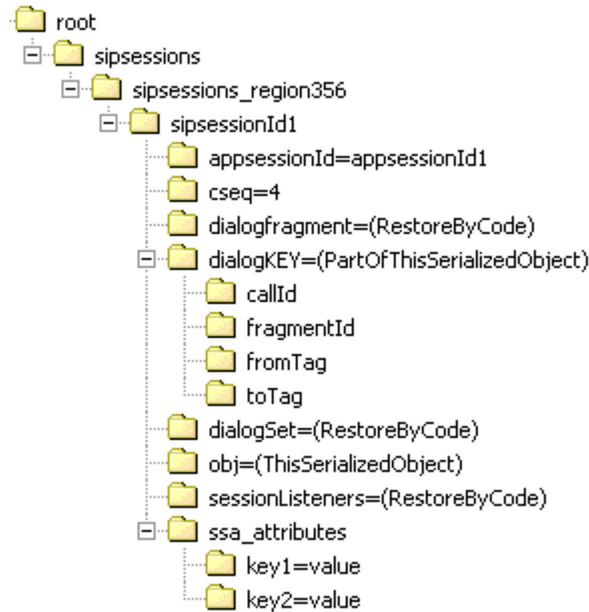


The node m_applicationTimers is created if at least one persistent timer has been created.

The node ssa_attributes is created if the application has stored at least one attribute.

The reference to the SipSessions can be restored “on demand” when accessed (e.g. by SipApplicationSession.getSipSession(id)).

2.4.1.3 SipSession



The application session id has only relevance when the SipSession is restored from cache, therefore it should be stored as a distinct attribute. (i.e. in general, id attributes to be used at restore need not to be resident in memory, only in cache possibly in serialized format as byte[])

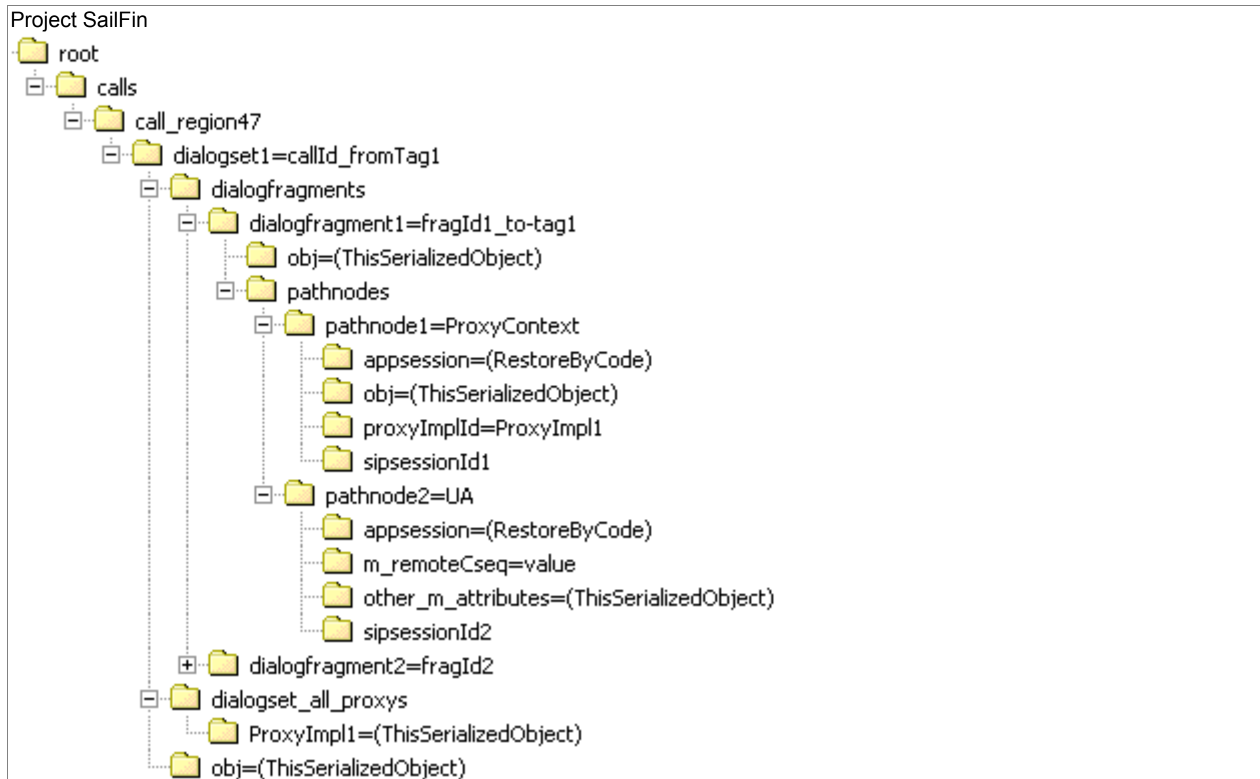
The CSeq value is updated frequently and therefore should be stored as a distinct attribute.

The DialogSet and DialogFragment references can be restored using the “dialogKEY” attributes.

The node ssa_attributes is created if the application has stored at least one attribute.

Dialog related internal structures

The figure shows the internal dialog structure needed by SessionManager to able to handle Sip Dialogs. This example shows an application with a proxy that proxy internally to an UAS instance.



The appsession reference of each pathnode does not need to be stored in the cache; the reason is that it could be restored from via the SipSession reference.
The m_remoteCseq value is updated frequently and therefore should be stored as a distinct attribute.

The ProxyImpl object might be shared between several ProxyContext objects and therefore must be stored separate from the ProxyContext PathNode. It is stored and serialized as one entity including references to ProxyBranches and original Request. (The “node dialogset_all_proxys” should be created only if at least one ProxyContext PathNode is referred)

2.4.1.4 Deletion and Ownership

When a SipSession is invalidated its DialogFragment reference count should be decreased. If no referring SipSession remains the fragment should be removed. When no DialogFragments remains in a DialogSet its structure should be removed.

2.4.2 Sailfin data model

Here we describe the SailFin specific modelling and how it deviates from the model above.

2.4.2.1 servletContext

The servlet context is not replicated. See [FSD].

2.4.2.2 Granularity of replication

In EAS 4.1 the replication is done based on a tree structure described above. The main reason behind this concept is that some data is changed more often than other data. E.g., the Cseq counter is updated at **every** transaction, while the attributes only change when an applications modifies them. The tree structure allows for an optimization of the replication process since it allows you to only update those parts of the tree that actually changed, instead of having to do a full serialization of the complete object at every replication.

Mapping this replication strategy to the SUN replication framework would mean that we replicate using the CompositeMetaData paradigm. In this paradigm, each of the objects to be replicated is represented as a tree of data as described above.

However, we have taken the design decision to not use the CompositeMetaData. The following reasons influenced this decision:

- The actual performance gain is smaller then you would expect
- The correctness is harder to guarantee (especially in the case of out-of-order delivery, which is possible in JXTA).
- The code is more complicated
- There are more test cases
- After a loss of a partial replication message the replica would remain out of sync with the active copy for its complete lifetime, while a full replication would guarantee that the replica would only be out of sync until the first replication event for the session.
- The CompositeMetaData is (currently) only applicable to application attributes and not to arbitrary data structures.

The replication framework does support some slots in that are 'externalized'. These are not stored in serialized form in the replica cache. Typically, this is for the purposes of accessing this data during processing of the replica cache (e.g., during the cleanup activities or responding to broadcast requests). These slots can be updated separately, without also sending the serialized content of the object. This can be used for efficiently updating some, frequently changing, fields like the CSeq number.

2.4.2.3 Timers

Application Timers will be modeled as separate entities with relations to the SAS and vice-versa. The reason for this is that at timeout we do not want to update the complete SAS object, but only the timer object.

The SAS timer is modeled as an externalized field in the SAS replica. This will mean we have less objects to replicate. Also, this makes it more similar to the HTTP case, where the lastAccessedTime is used for reaping purposes.

2.4.2.4 Reference resolving

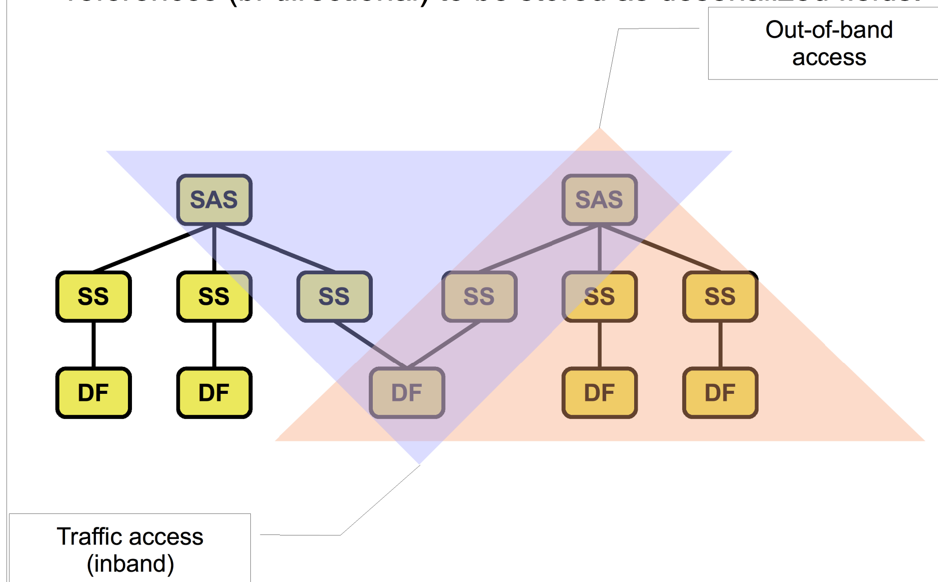
Like mentioned in the EAS 4.1 datamodel, all the references between objects will be stored and serialized as ids.

The references are resolved to real objects either during deserialisation or 'on demand'. Then they ~~are~~can be cached. It should never happen that the referenced object can 'migrate' (i.e., move to another instance) during the lifetime of referencing object, so cache invalidation should not be needed.

2.4.2.5 Tree activation

Currently each object is treated separately from a replication point of view. By resolving the references embedded in the deserialized version of the object, the entire tree of objects can be loaded, one at the time.

FFS:
From a performance and consistency point of view it might be useful to re-activate and migrate the complete tree (SAS-SS+HTTP-DF) together. The problem is that two types of trees can be identified. During traffic (in-band access) the DF, with its related SSes and their SASes should be loaded. When the SAS is accessed, all its descendants (SSs and DFs) should be loaded. This requires quite extensive knowledge about the relationships in the replica cache and would require all the references (bi-directional) to be stored as deserialized fields.



2.4.3 Replication triggers

The following replication triggers have been identified:

in-band-access

- When the dialog moves to the confirmed state, the corresponding DFs, SSs and SASes will be replicated (after the 200OK or speedy notify is handled)
- When a dialog is cloned, the ~~DS/~~DF and cloned SS and all modified SAS and SS in the same tree are replicated (after the 200OK is handled).

out-of-band access

Out-of-band access is only when the objects are not accessed in the context of a SIP transaction that exists in the same tree. E.g., an SS requesting a SAS that is not its own parent via the SIPSessionsUtil constitutes out-of-band access.

- When the SAS is created out-of-band, using the SessionFactory.createSipApplicationSession(applicationKey) or SessionFactory.createSipApplicationSession() the SAS is replicated immediately
- When the SAS or SS is accessed out-of-band, after each setAttribute() invoked on it, the SAS or SS is replicated.
  ~~*Note that this behavior is equivalent to the modified-session replication. There is no equivalent to full session replication for this scenario.*~~
- After a timeout on the SAS or ST in an already confirmed dialog (the SAS and SSes and updated ~~Timers~~ST are replicated when the handleTimeout returns).

~~**Out-of-tree-access**~~

- ~~When during incoming traffic (based on request processing) a SAS or SS is accessed that is not a direct relation (I.e., the no parent, no child or no sibling), then the replication is performed after the request processing is finished.~~
  ~~This kind of access is dubbed out-of-tree, since you modify one tree while processing requests in another tree.~~

The current proposal is to wrap the object returned from the factory (I.e., out-of-band) in a special wrapper. Any container objects obtained via this object would be wrapped as well (recursively). Any modifications done on the wrapped objects are stored immediately.

FFS:

Another possible implementation of the trigger handling can be done by intelligent objects and a threadlocal variable storing the context in which the invocation is done and stores all the objects dirtied by the request processing.

- When an object is modified (made dirty) and the context does not indicate request processing, replication is triggered immediately (out-of-band). Otherwise the object is just marked as dirty and added to the list of modified objects in this thread.
- At the end of the request processing, if the transaction is confirmed, all dirty objects in the tree and all object in the modification list of the thread are replicated (both in-band and out-of-tree).
- At the end of request processing, if the transaction is not confirmed, only those objects that are in the modification list of the thread and that do not belong to any of the trees involved in the request processing are replicated (out-of-tree)

FFS:
We might introduce artifical replication triggers that allow the application to control when replication is done. This is specifically useful in the out-of-band and out-of-tree case. Think of grouping actions in one method (like the WLSSAction in BEA) or grouping actions in a unit-of-work construct.
The BEA implementation not only ensures aggregated replication, but also does obtains a remote lock to guard against current access.

The definition of a unit-of-work concept where all the changes can be marked as one unit and 'committed' together is still under discussion.

A relatively simple solution for aggregation could be to introduce a flag on the above mentioned wrapper object. Setting a special boolean (autoReplicationDisabled?) attribute will trigger the flag. Setting another special attribute will trigger replication (triggerReplicationNow). The latter 'magic attribute' has already been implemented for HTTP.

2.4.4 Lazy creation

When a SS or SAS is not explicitly requested by the application, we must ensure that these objects are garbage collected when the early transaction times out and we should not rely on the SAS timer for cleanup.

It is unclear whether lazy creation will really be implemented.

2.4.5 DF refactoring

The current handling of the DS and DF is based on a single JVM.
I.e., the DS is used as a collection of DFs. The first DF added to the DS does not actually use the DFid, it is implicit. Spiraling is detected by the finding that there is already a DF in the DS and not based on fragment id. Only at the second fragment do we start adding DFids to messages to distinguish them from each other (in the same JVM).

The problem with a cluster setup as we have it in sailfin is that spiraling is not limited to a single JVM. We should detect spiraling in the cluster, since we can never be assured that DFs will not be co-located on the same JVM later, making both parts of the spiral indistinguishable.

Therefore, we will always add a unique fragment id to outgoing request.
Then we can trust on the presence of a fragmentID in the incoming request to determine whether we are in a spiraling situation and do the DFId handling appropriately.

Furthermore, the code can be refactored in such a way that the DS can be removed. Its main purposes were, saving memory by sharing part of the DF state between DFes and detecting the spiraling without the need for an explicit dialogFragmentId on the first fragment. The first purpose can just as well be solved by internalizing the relevant portions of the state (sharing on JVM level). The second purpose becomes obsolete based on the above observations.

2.4.6 Local locking

The replication triggers will replicate the entire tree (from the parent SAS all the children and grandchildren are updated). In case of modified-session, only dirty objects are actually replicated.

However, due to the concurrency of SIP it can happen that multiple SSes in the same SAS are concurrently modified. Replication on one of them will also replicate the chances make by the second one. To remedy this we need some sort of locking.

To lock the tree (the SAS and all its children) for the duration of the SIP transaction is not an option, since SIP transactions can be very long and this would ruin the latency.

The next best thing is to lock the tree for the duration of the message handling. This would mean that any changes made by a application in handling one message would at least be replicated consistently, even if the modifications made over different messages in the same SIP transaction would not.

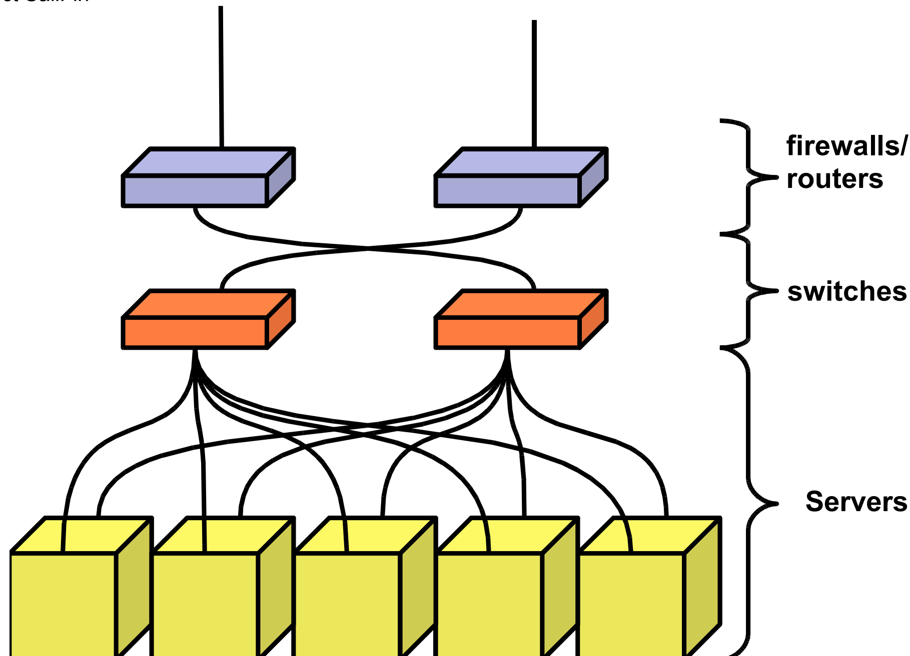
The easiest solution to locking the tree is to always try to obtain a lock on the root of the tree (the SAS in our case). This can be implemented as a local lock. A similar lock is already provided by the replication framework to arbiter between foreground and background activity. But the functionality of this has to be extended.

It should also be considered to extend this locking to remote locking (see below).

TODO:
Do we also need a lock on the DF? There can be concurrent access on the different sessions in the same DF.

2.5 Cluster connectivity

The clusters network configuration is expected to look something like this:



All the servers have dual ethernet connectivity. When one of the ethernet connections fails, the other takes over. This is handled transparently for the users of the replication framework, using IP multipathing (ip multipathing in solaris) or ethernet bonding (ethernet bonding in linux).

Both internal and external traffic are routed over the same ethernet connection. This avoids network segmentation where one server (or multiple servers) can still receive external traffic, while the GMS (based on the internal traffic) assumes that the server is down.

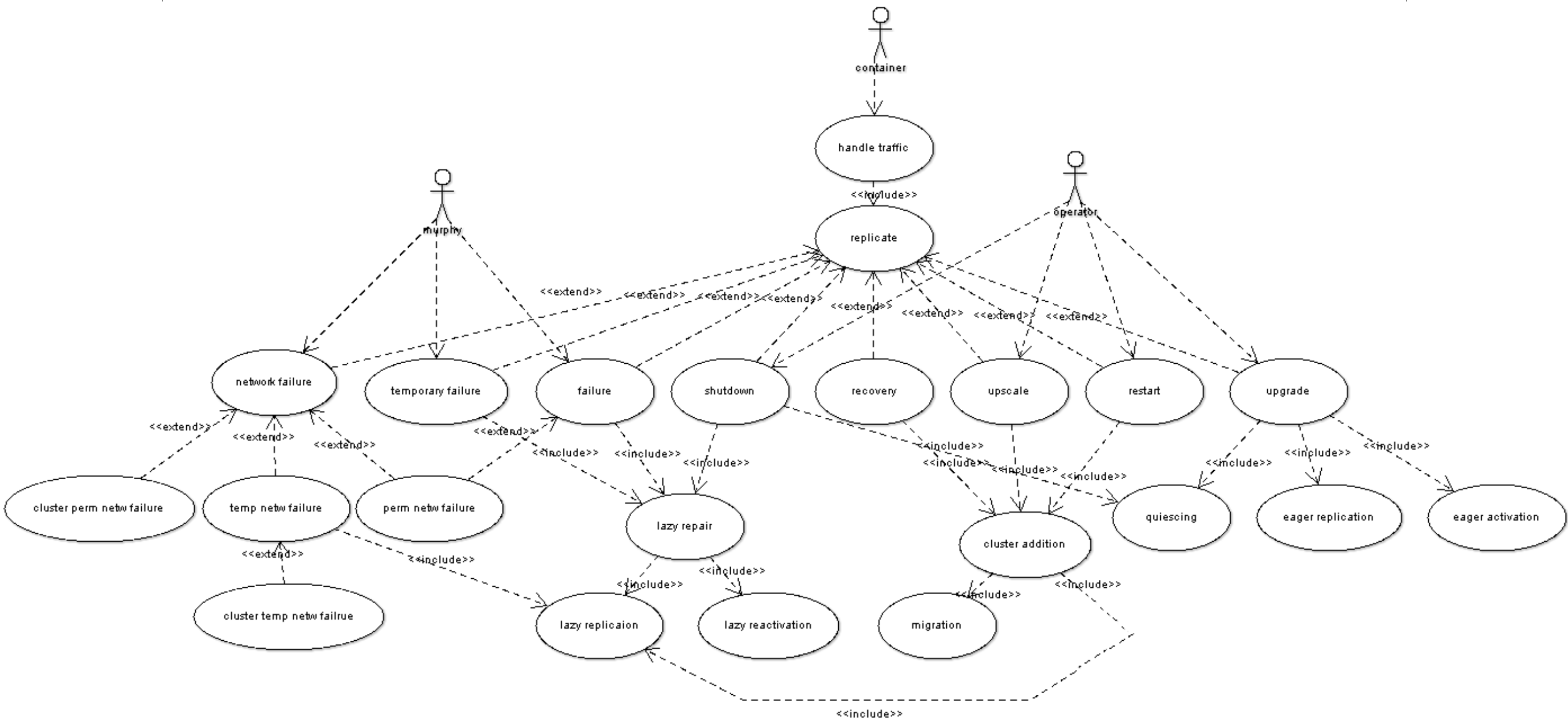
The setup with the switches for routing internal traffic makes it very unlikely that there is a network segmentation of multiple servers. If both switches fail, no communication is possible anymore. If both NICs on a server fail or if both ethernet connections from one server fail, then this can be considered a network failure from that specific server.

It is clear that such a configuration can not be mandated in general. So the SSR can not rely on the qualities such a setup will bring. Effectively, for SSR in general, this means that network segmentation can occur.

2.6 Use cases and design decisions

2.6.1 Overview

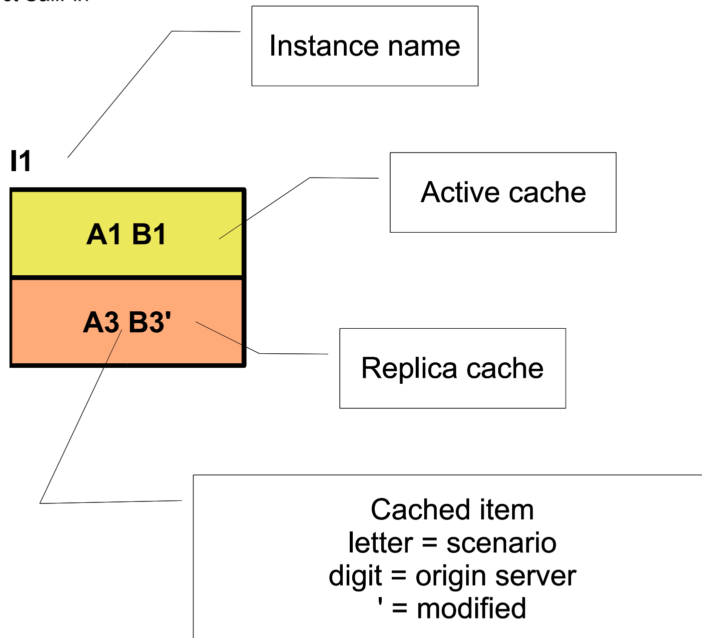
The following use cases are identified. This is probably not a proper use case model. However, they serve as a overview for the rest of the section and as a way of structuring the common parts. The use case 'handle traffic' is not described in this section.



This figure only shows one of the upgrade alternatives. There also is another disk based alternative that is actually preferred.

2.6.2 Notation

The figures used in this section use the following notation.



The cached item naming is as follows:
In general the following letters are used:

A items that are removed during the use case
B Items that are modified during the use case
C Items that are added during the use case

The modifications are indicated as follows:

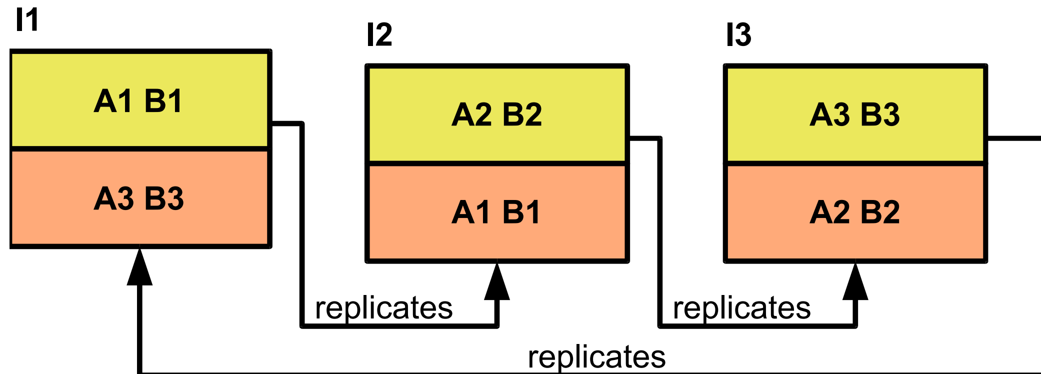
B1 an item created by server instance 1.
B1' this same item after the value has changed change.
B1" the modification of B1'.

So in a sense the number of modifications indicate the version number of the item.
After migration or reactivation the ownership of a session changes. This is indicated as follows:
B1'[2] session B1 is modified on instance I2.

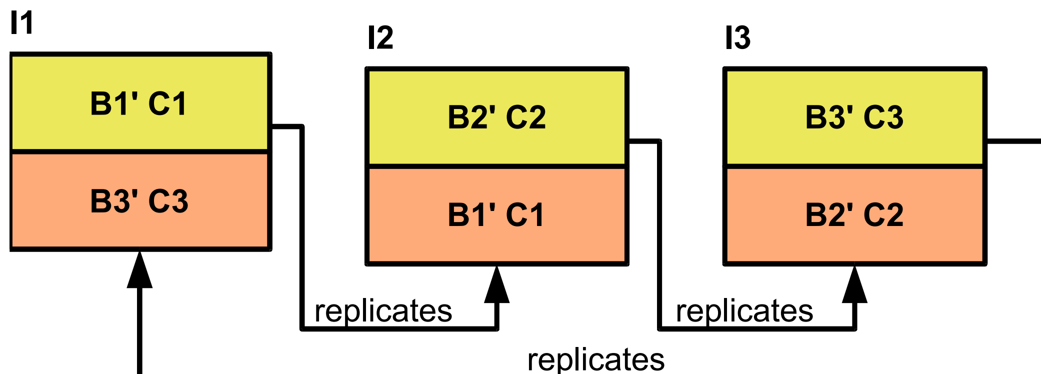
2.7 Replication

The starting situation is based on a ring topology like this:

Each server instance replicates to its neighbor in the ring in a clockwise direction.



When A1, A2 and A3 are removed, B1, B2 and B2 are modified and C1, C2 and C3 are added the cluster looks like this:



Timers will expire only on the active copies of the SAS. Every replication artifact has two copies in the cluster. ~~which is~~ The active copy is located on the home instance. All is well. This is called a healthy, non-compromised state.

2.7.1 Asynchronous replication

Replication is done asynchronously using JXTA. JXTA does offer a reliable transport (it is build on top to TCP).
The replication will be done in an asynchronous way using JXTA.

Asynchronous replication can mean that some replication messages are lost, while the ~~client~~replication source is not aware of that fact.

Unsuccessful replication can lead to:

- non-existing replicas – if the create message is lost
- stale replicas – if the modify message is lost
- zombie replicas – if the delete message is lost

SSR has to deal with all of these. The first two are temporary and will only exists until the next replication (if that ever comes). The last is more tricky. SSR does not offer a way to cleanup these replicas or to avoid timer expiry on the replica copy.

2.7.2 Transactional replication

In SSR the replication consists of several different items with their own lifecycle and replication triggers. Although it has to be stated that in most cases, the replication triggers actually do coincide. Also, the life-cycles of timers, SSes and SAS are nested; a SS can not outlive its SAS, nor can a timer.

It can happen that a part of the total replication fails. E.g., this can be due to either a lost replication message (since replication is asynchronous) or due to a crash of the server before the replication is complete (but this could still be after the message has been handled, again since the replication is asynchronous).

In such a case there can be an inconsistency in the data model. E.g., there can be references to objects that do not exist, or objects may reference objects that have become stale.

These inconsistencies should be rare. They are difficult to avoid without introducing transaction support. Transaction support would again decrease the performance and increase the complexity, therefore the design decision is to not implement it.

Instead we will focus on detecting the inconsistencies. However, inconsistencies between objects that are referencing each other is difficult (or impossible) to detect.

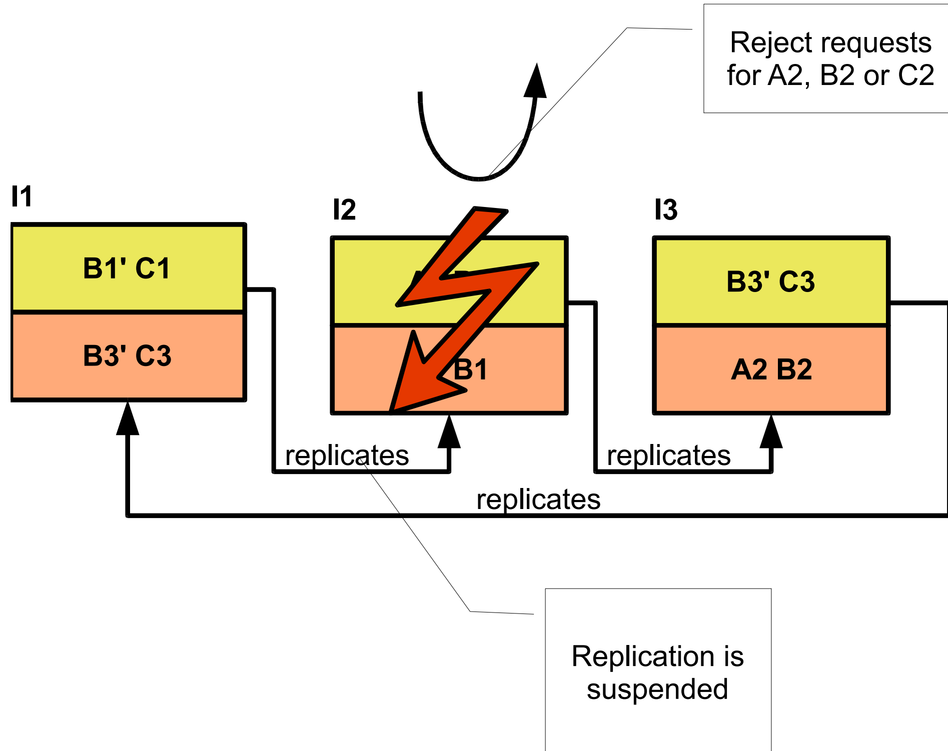
Incomplete replicas trees should be removed once the objects become orphans. This is handled by the cleanup activity.

The decision is that there will be no transactional replication. Each object is replicated separately.

2.8 Temporary Failure

Now let us assume that server instance 2 fails. It takes a while before the Group Membership Service (GMS) concludes that a server instance is down and will perform a cluster reshape.

During the temporary failure of I2, replication to I2 will fail. And, of course, since I2 failed it will also not replicate to I3 during its failure.



As you see in the figure, A1 is deleted, B1 is updated and C1 is added during the failure and these changes will not have been replicated to I2. Therefore, we are vulnerable for multiple failures at the moment. Also, there is only one (replica) copy of A2 and B2.

During the temporary service failure of I2, but before a reshape event is generated the LB (FE) will still direct traffic to this server instance.

When using the UC policy, any requests (whether initial or subsequent) received by the load balancer during the temporary failure will not be handled, instead a 503 error response is returned. The 503 response should include a retry-after header where the time corresponds to the estimated time before a temporary failure. Any responses received during the temporary failure will be dropped.

Using the RR policy, there is no difference between a temporary failure and a cluster reshape. The LB reacts on its local knowledge of the availability of the server instances instead of waiting on the cluster reshape event. It will already route or re-route requests to a backup instance.

Note that using the RR policy, it can happen that the message was already sent to I2 and I2 failed during the handling of the message. Then there is an option to retransmit the message to the backup. However, this should only happen if the message is marked as idempotent (i.e., the handling of the message is without side-

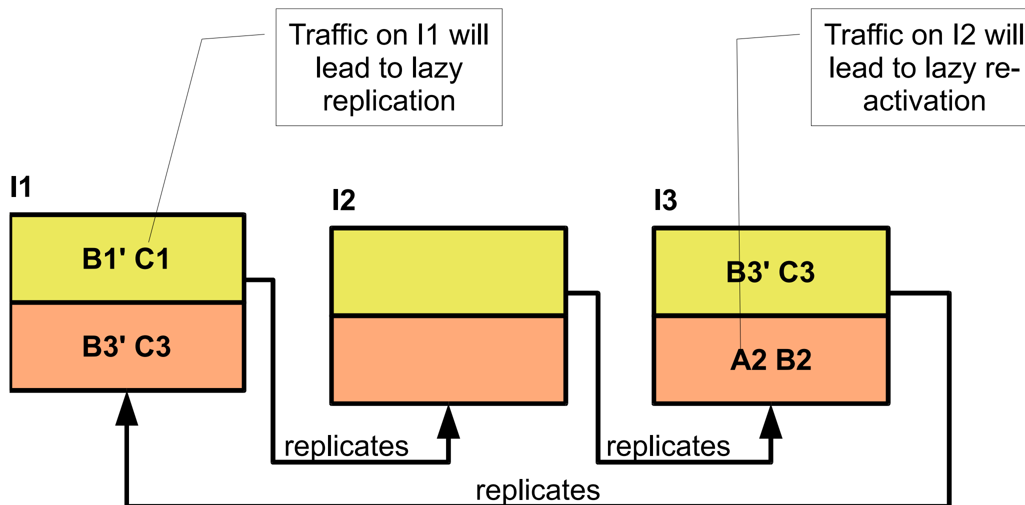
effects). If the message is not marked as idempotent, the LB sends an error message and the client must handle the consequences (i.e., the uncertainty whether the message was partly or completely processed).

No special handling is needed for timeout on the SAS in this case.
I.e., if A2 or B2 are SIP application sessions or timers, these timers will not expire. Since the temporary failure situation is, eh, temporary, this is acceptable.

We do assume however, that the timers on the replica copies must not yet fire during the temporary failure situation. This means that the grace period before firing a timer on the replica, is at least as large as the time it takes GMS to notify the LB of an cluster reshape after an instance failure.

It is assumed that when I2 is restored, its caches are empty. If they are not empty, then we are actually talking about a network failure usecase (see below).

When the instance I2 is recovered after a temporary failure situation (with cleared caches), the **lazy replication** actions will be performed between I1 and I2 (to restore the lost replica cache of I2) and **lazy reactivation** actions will be performed between I3 and I2 (to restore the lost active cache of I2).



2.9 Network Failure

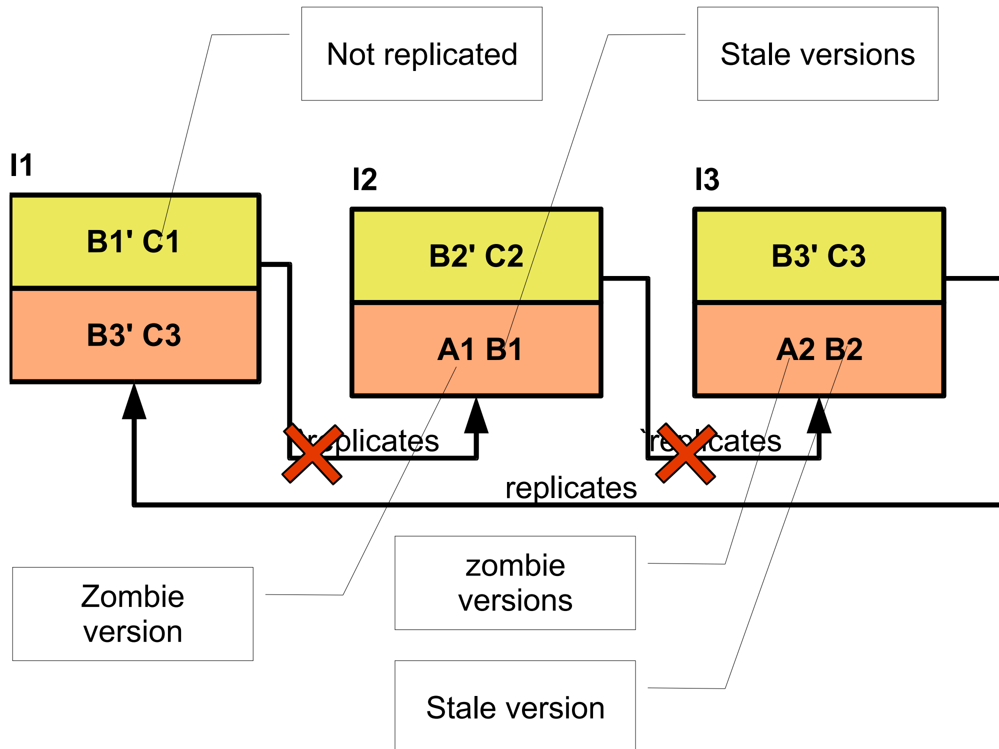
This section assumes the cluster is setup in such a way that network failures effect either one instance or the whole cluster.

2.9.1 Temporary network failure of one server

Let us consider the case where only one server is affected. From the point of view of the rest of the cluster this looks like a temporary failure.

The main problem with this is that changes are not replicated to the disconnected server. Therefore, this can lead to

- non-existent replicas (create message lost)
- stale replicas (updated message lost) or
- zombie replicas (remove message lost)



In our example, when I2 is disconnected, this can lead to several stale replicas and zombie sessions. (we assumed that A2 was removed, B2 was modified and C2 was added, all based on timer expiries. (So even when I2 is disconnected, this does not mean that is content is static).

However, it will **never** lead to multiple copies of replicas on other instances than the on the replica partner.

In the HTTP case, the version numbering and the lack of timers offered protection against stale and zombie sessions. However, as we will see later, versioning is fundamentally flawed in SIP, so the same solution can not be used for SIP.

However, since stale and zombie replicas can never be excluded (they can also occur when replication messages are lost), we must prepare for this anyway.

Stale replicas only exist until the next successful replication message (if any). zombie replicas are harder to handle gracefully, they can not be distinguished from valid replicas where the active copy is lost.

FFS:
The zombie replicas in this specific case (temporary failure) might be handled similar to the upgrade case; I.e., by doing a diff between the replica and the active cache when the temporary failed instance is restored with an intact active cache. Alternatively, we could buffer failed replication messages (specifically the remove messages) to be retried after the replication partner is restored (with an intact cache). However, these solutions seem to be overkill for the problem.

2.9.2 Permanent network failure of one server

During a permanent network failure, the disconnected server will trigger a cluster reshape. The result of this is that replication will continue in the rest of the cluster. This can lead to (re-)activation of sessions that are in the active cache of the disconnected server. This can happen because of traffic that is redistributed by the LB after the cluster reshape or due to timer expiry on replica objects owned by the disconnected server. After activation, these objects can also be deleted. This means that there may be multiple active versions or even active versions of removed sessions in the cluster. This is a particular problem after the network failure is repaired and the disconnected instance re-joins the cluster.

The current solution is to not worry about this scenario. It is expected to be unlikely, since it can only result from multiple failures.

FFS:
The suggestion is to purge both the active and the replica cache on the disconnected instance or better yet, restart the disconnected instance, i.e., treat this as a normal failure scenario. Restarting the instance (via a panic on OS level) is the way that the 'SUN cluster' solution handles these kind of cases!

This leaves the problem of detecting the disconnection.
To detect a disconnection, the instance has to monitor the group membership service. If ,and only if, it looses connection to all the other instances in the cluster will it conclude that this is a disconnection scenario.

There is one problem with this; In a two server cluster, where there is only one server instance per server, both server instances will conclude a disconnection scenario and purge their caches.

For this we introduce an additional check for network connectivity, e.g., an extra check to see if the server can reach other servers (like the DAS) or add a check on incoming traffic, and assume that if no traffic is received in a certain time that the instance is disconnected.

SUN offers an API for clearing the caches. Thus, the sailfin container could implement the disconnect detection itself (based on GMS) and invoke the appropriate APIs.

2.9.3 Temporary network failure of the whole cluster

If the whole cluster suffers a temporary failure, e.g., because both ethernet switches failed, then to each individually this would look like a disconnection.

There will be no external traffic. The only source of potential inconsistencies (zombie, or stale sessions) can be timers that expire on the active sessions (since the replica timer handling will only kick in after the cluster reshape). The stale replicas will be repaired using **lazy replication**. Zombie replicas are a bigger problem.

2.9.4 Permanent network failure of the whole cluster

If the whole cluster suffers a permanent failure (e.g., because both switches suffer a permanent failure) it will be a mess. Every timer will expire twice, once on the active and once on the replica. Fortunately, this scenario will not lead to zombie replicas (provided both applications instances react the same on the timeout), but every session will get at least two active copies eventually. This situation continues when the cluster is restored.

FFS:
The proposed solution (expat lists) for the create vs find dilemma can also be applied in this case. If the cluster is repaired in such a way, the duplicate replicas will be removed lazily.

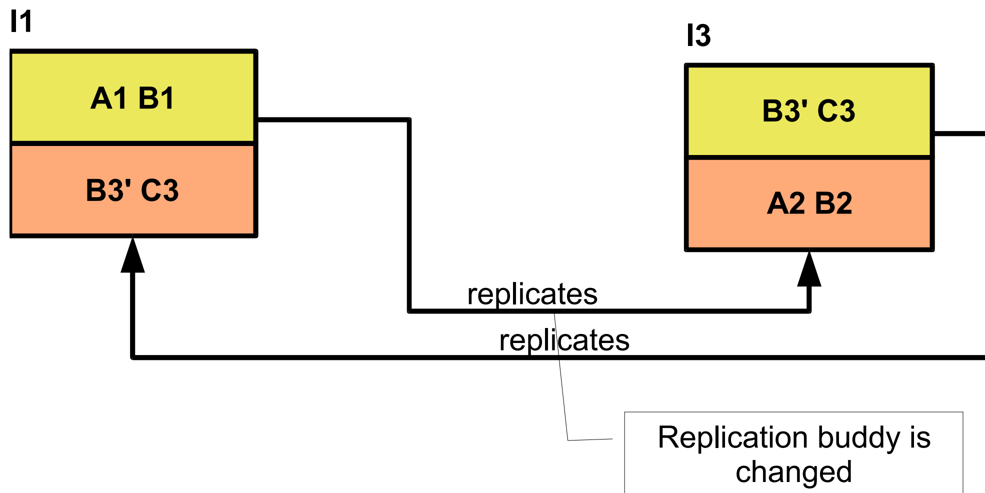
2.10 Network segmentation

If the cluster is set up in such a way that internal and external traffic is routed differently, it can happen that traffic is received on an instance that is considered disconnected by the rest of the cluster. This is similar to the network failure situations, except that the activations can now be based on traffic, and not only on timeout events.

2.11 (Permanent) Failure

In the temporary failure scenario that was described above, the failure was relatively short. If the failure persists the Group Membership Service (GMS) will detect this and a GMS event is generated. This will cause a cluster reshape.

For the Load Balancer (Front End) in UC mode, this means that the node is removed from the consistent hashing. For the replication mechanism it means that the node is removed from the replication circle.



The replicas of I1 (which were on I2) are lost. They will be repaired using a **lazy repair.** I.e., the replication will take place when the sessions are next accessed on I1.

Of the active sessions on I2 only replicas are remaining on I3. The load balancer will direct requests directed to these sessions to other instances in the cluster. These replicas will be reactivated using **lazy reactivation**.

2.12 Lazy replication

Lazy replication occurs when a session for which there is no replica, is accessed on an server instance. The replica can have been lost due to several reasons, of which the most obvious is a failure of the original buddy.

The access can either be due an external event (e.g., a request) or due to a timeout. After the session is updated a new replica is created on the current buddy using the normal replication procedures.

After all the sessions on I1 have been accessed all the sessions in the active cache of I1 will contain multiple versions and will hence not be vulnerable against failures.

Lazy replication is already implemented as part of the normal replication. Nothing special is needed.

2.13 Lazy reactivation based on traffic

Some events lead to the loss of an active cache, e.g., a failure or a shutdown, but also a temporary failure. When the active cache has been lost some sessions will only exists as replica copies.

The lazy reactivation activity creates an active copy of the session based on the available replica copy. All replica copies are removed. After the reactivation the normal replication will ensure that a new (up-to-date) replica is created.

E.g., when A2 is accessed on I1, lazy reactivation is initiated.

The lazy reactivation is based on the ‘load request’. If the object (or the correct version of the object) is not available in the active cache, the object can be retrieved from another instance in the cluster. There are two versions of the 'load request' one which is version aware and one that is not.

Version aware load request
When a session is accessed and the requested version is known the following steps are taken until one of them yields a result.

- the session is retrieved from the active cache on the serving instance, provided it has a sufficient version number (find).
- the session is returned from the replica cache of the serving instance, provided it has a sufficient version number.
- A message is broadcast in the cluster. Every instance in the cluster will return the version it has to the requestor. The requestor will wait until the first sufficient version is returned. If no sufficient version is returned the next step is executed.
- The request is rejected with a 481 response.

The load request behavior relies on version numbering. In HTTP each (external) request does contain (typically in its cookie) the version number of the session object that it requires. This version number is updated in every response. The version number is used in the checks above to check if the found version is sufficient. In certain cases (e.g., when a response is lost) it might happen that the version in the cache is higher then the version in the request. In such cases the version is not considered 'sufficient'!

SIP does not support the concept of cookie, and introduces the additional complexity of multiple responses to one request etc. Therefore, the versioning strategy is somewhat limited in a SIP environment.

Non-version aware load request
For this reason, the SIP procedure needs an additional non-version aware load request. If based on a request the following procedure is followed:

- the session is retrieved from the active cache on the serving instance.
- A message is broadcast in the cluster. Every instance in the cluster will return the version it has to the requestor. The requestor will wait until all a response is received from every instance in the cluster. It will then take the highest version number from all the responses.
  If a response is received that there is an active session with a lock a 503

response is returned (this is not relevant for lazy reactivation, but is relevant for migration described later).
- The request is rejected with a 481 response.

For requests that are not based on a request (e.g., a timer expiry) the procedure is as the same, except for the last bullet. For responses, the load request will never be performed. The reason for this is that we do not have transaction replication. Since responses are always related to an ongoing transaction, it is not useful to retrieve the session from another instance, if the transaction can not be retrieved as well. In those cases the response will be silently dropped.

The reasons for the non-version aware load request are related to the lack of version numbers in SIP.

Changes to current load request

Waiting for all the responses in the load request means that the latency of the load request is the determined by the latency of the slowest response of a cluster member. Furthermore, it has the problem that in failure situations it might happen that our view of the cluster is not exactly correct and less responses are received then there are cluster members in our view. In that case we have to wait for 4 seconds for a response that never comes, since one of the instances is down, even though our GMS says it is up. However, this is not a bad as it seems; the load requests will result from the fact that the LB is rerouting traffic. The LB will only start rerouting the traffic based on a GMS notification. The GMS notifications are received by all instances in the cluster at approximately the same time. Therefore, the GMS an instance has will most likely be correct in single failure situations (barring some small delays in GMS notifications).

Currently, during the load request any reported copies are not removed. The idea is that the version system will ensure that the (stale) replicas are not used anymore and eventually will be cleaned up by the garbage collector, if not accessed in a while. The limited version capabilities of SIP, combined with the more intrusive timer handling, means that we should ensure proper house cleaning, and all the copies reported in the load must be removed as well. This is also needed for the active migration described later.

Furthermore, there is no negative acknowledgment implemented. This means that every load requests for a non-existent artifact will give a 4 seconds delay as described above.

Find vs create

The SIP @SipApplicationKey targetting mechanism and the createSipApplicationSession method, should both be interpreted as a find if the object already exists, and a create if it does not. However, when the object is not in the local cache this can have two reasons:

- the object does not exist (it was never created or it expired. It does not exist anywhere in the cluster)
- The object does exist, but due to a previous failures or restores it is not located on the instance where the load balancer directed the request.

If we can not distinguish between these two situations, then we have to do a non-version aware load request for every request of this type which can not be located in the active cache, effectively, this means that every create will be preceded by a (superfluous) find.

FFS: The possible solution to this problem is to be able to distinguish the case where there is a copy of the object in the cluster or not. For these purposes the following procedure is followed:

- after any cluster reshape each instance requests the others to creates an 'expat list' for it. This is a list of all the objects that are located in its active or replica cache of that instance but do have their 'home' at requesting instance. This can be determined by the consistent hash located in each active or replica copy.
- The expat lists contains the id and the version of the objects and is used on every home. Instance during the find as follows:
  - If the object is found in the active cache this is used
  - if the object is found in the expat list, a version specific load request is performed. If the version specific load does not give a result this can be due to the fact that it is expired after putting it in the expat list, but before it is being requests.
  - If the object is not found in the active cache, nor in the expat list, or if no result is returned from the version aware load request, null is returned from the find. The invoker of the find creates a new object (in case of the SAS being accessed out-of-band) or returns an error (in case the object should exist, e.g., when the SS is requested during traffic).
- After the cluster reshape, but before the expat lists are received, the instance must do a non-version aware load request is the object does not exist in its active cache.

The reason this scheme works is because the LB guarantees that migration and reactivation will always be done on the home, so the expat list never changes without the home being aware of this (contrary, in Round Robin, there would be no concept of home, so this scheme would not work for RR!)

In pseudo code this looks something like this:

find functionality

```
SAS findSAS(long id) {
      sas = activeCache.get(id);
      if (sas == null) {
            // not in active cache
            if (expatlist = null) {
                  // we do not have an expat list yet
                  // fall back to the non version aware load
                  return nonVersionAwareLoad(id);
            } else {
```

```
                        ExpatItem expat = expatlists.get(id);
                        if (expat == null) {
                              // it does not exist in the cluster
                              return createNewSas(id);
                        } else {
                              // it exists elsewhere
                              expatList.remove(id); // we move it to here
                              return versionAwareLoadRequest(id, expat.version);
                        }
                  }
            } else {
                  return sas;
            }
      }
```

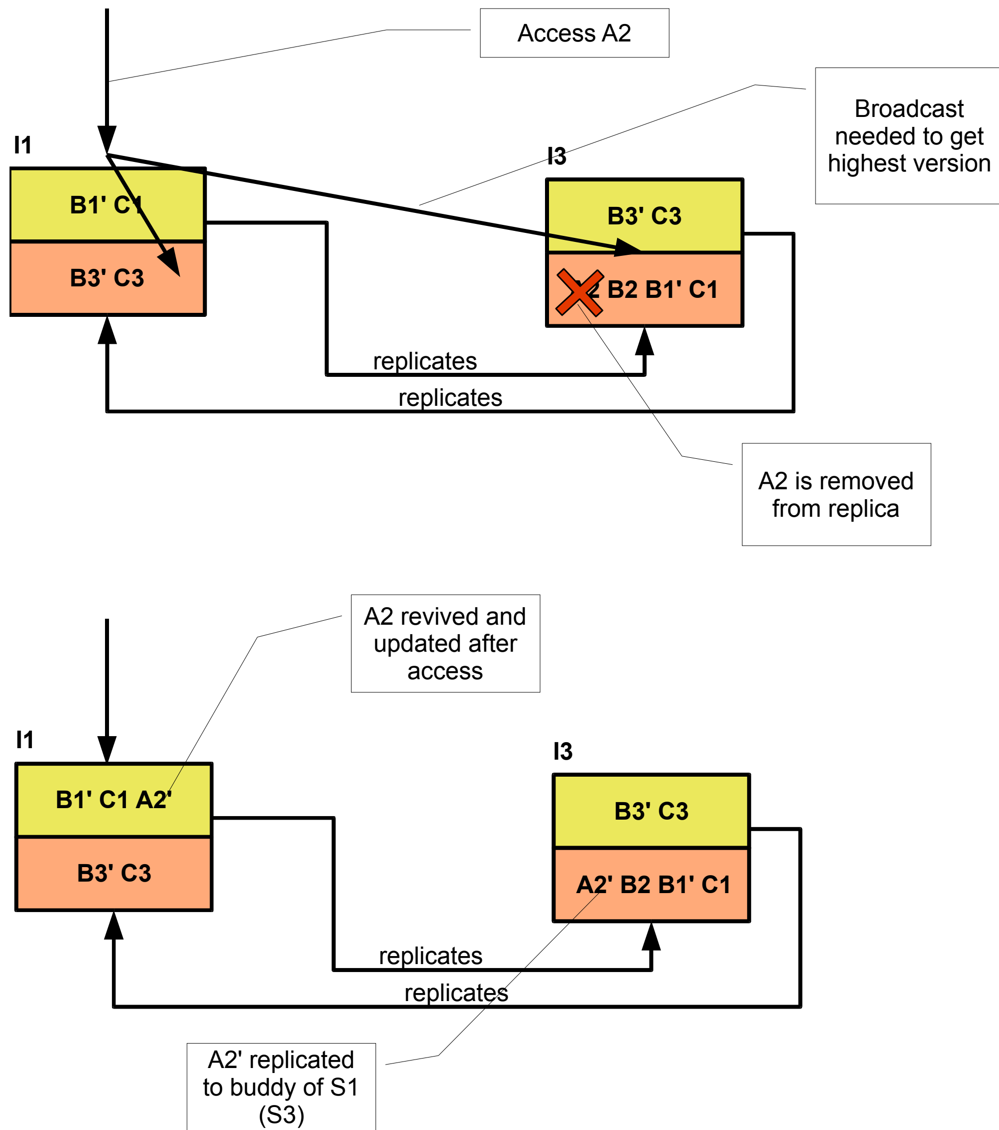
expat list generation

```
for(ActiveItem item: ActiveCache) {
      int instanceID = applyConsistentHash(item.hashcode);
      expatlists[instanceID].add(new ExpatItem(item.id, item.version));
}
for(ReplicaItem item: ReplicaCache) {
      int instanceID = applyConsistentHash(item.hashcode);
      expatlists[instanceID].add(new ExpatItem(item.id, item.version));
}
```

There can be some optimizations on where and when the expat lists are requested. In case of an node joining the cluster, only the new instance needs to request the expat lists, since the active cache of the others should still contain appropriate expat lists (because the new instance is just added it does not contain any data). The expat lists will be too big (e.g., home location has changed, so there will be items in the expat list that no longer belong there), but since only that subset of the expat list that is used are those items that have their homelocation on the instance. UC routing ensures that a node addition, will only shrink that and not otherwise influence this. Another optimization is that after a failure, the expat lists are not requested immediately, but after a delay. The reason for this is that the instance might return after a failure, and the added effort of the expat list handling might be wasted.

The latter optimization depends on the relative overhead of the non version aware load requests compared to the expat list handling for that duration.

Scenario continued



In the above figure, when A2 is accessed on I1, it will see that A2 is not in the active cache of I1.

Without the expat list handling I1 will issue a non-version aware load request, when using the expat list handling, I1 will find a version of A2 in the expat list and issue a version aware load request, which results in a broadcast for the specific replica version in the cluster.

For simplicity, the broadcast is also shown to itself, but no network communication is needed to access the instance's own replica cache.

The load request will result in A2 becoming active on I1 and after the update it will be replicated to I3 (as A2'[1]).

So the replicas are 're-activated' lazily on the server instance where the next external event is directed by the load balancer. As shown in the figure, I3 does contain rather more data then I1 since after the failure it contains both the replica of I1 and I2. However, assuming that the LB will redistribute the session from I2 evenly over the remaining server instances, the data distribution will eventually even out.

In summary, the following updates have to be made to the replication framework.
- After the load request any returned artifacts must be removed (including active versions)
- A non-version aware load request must be provided as an alternative to the version aware load request.
- The load request must implement the negative ACK to avoid waiting the entire 4 seconds in case of lost message or cluster shape disagreement.

The next sections describe some of the problems with version numbers in SIP.

2.13.1.1Versioning

In HTTP versioning is introduced in the client. Each request will contain the version requested by the client.
This can be used for:
- staleness check of the active copy
- version aware loading (faster then non-version aware loading)

If an instance can not be found in the active cache, a load-request is always performed in the HTTP case.

In the SSR case,we are not able to use a version provided by the client, for the reasons described in the next sections. This means that the staleness check can never be done in a guaranteed way. However, versions are still used and usefull:
- using the principles mentioned above, we can still use versions for the version aware load request based on the expat list information.
- Versions are used to select the highest version during the non-version aware load request.
- Versions are used to determine staleness during replica timer expiry.

So versions are needed and useful. It is just that their use is different from the HTTP case.

2.13.1.2Access with versioning

Besides protocol events directly accessing the session there are also other events that need access to the session data and that are not directly related to external events.

Access via the SipSessionsUtils

The specification allows applications to access a SipApplicationSession by its SASid. This id can be obtained via an out-of-band mechanism. In this case, obviously, there is no version information available.

There is no way that we can avoid this. By its very nature the access is based on the session id only. Requests can be done from any context, e.g., from an EJB, from an unrelated SIP session or HTTP session, from a webservice etc.

Instead the ‘best-effort’ principle is applied in this case. In practice this means that we will use whatever latest version of the active object is available on the server instance where the request occurs. It might even be that there is no active version of the object available on the server instance. This can have two reasons:

- The instance is not the home instance
- The instance is the home instance, but due to earlier failures or cluster reshapes the active copy is not available there.

We will not support the first case, only out-of-band access on the home instance is supported, since it can lead to trashing of data, see [FSD].
If the SAS is accessed on an instance that is not the home instance a null reference will be returned.

Indirect access

Messages are always related to protocol sessions (HTTP or SIP). But, based on the reception of such a message, the application may access other sessions indirectly. I.e., a SIP session can be a child of a SIP application session. When obtaining the parent SIP application session no version information is available. Via the SIP application session other protocol sessions can be accessed. Similarly, no version information is available when indirectly accessing these protocol sessions in this manner.

One way to ensure integrity of the complete data would be to include version information in the internal references. However, this would in effect mean that any update of any of the children of a SAS would result in updates of the complete graph (i.e., if a child changes its version has to be updated, then the version of the reference to the child in the parent has to be updated, since this changes the parent itself, the version of the parent has to be updated, and since the version of the parent is updated, all the children have to be updated). Effectively this would mean that always the complete graph would be updated (in a transaction :-). This is not inline with the principle of regarding the objects in the graph as independent entities with their own lifecycle and own replication triggers. And, like stated before, transactions are not supported.

Internal references will be resolved using the load request as described above, i.e., returning the latest version in the cluster if no active copy is available on the instance where the request is initiated.

Access via @SipApplicationKey or encoded URI/URL
The SIP application key annotation can be used by the application to control the allocation of incoming sip sessions to a sip application session. Such access will not contain a version number.

Similar to the association of incoming HTTP or SIP messages that are associated with the SAS via the encode URL/URI functionality.

For the @SAK it is unknown whether it is a new SAS or an ongoing SAS.

As discussed above, we can avoid the non-version aware load-requests based on the awareness of the other versions in the platform (the expat-lists).

Access of DF
When a subsequent request is received and there is no dialog fragment available in the active cache, a load request should be performed to get the highest version of the DF in the cluster.

For initial requests, this is not needed.

2.13.1.3 Use of cookies in SIP

This section investigates if and how version information can be included in the SIP protocol.

The cookie concept in HTTP is not present in SIP. However, there are some possibilities to sent opaque data and have this returned later by the UE.:

VIA (TS is UAC, TS is proxy)
Opaque information can be included in the via header of an outgoing request. Any responses to this request contain this opaque information.

Can be updated with every new outgoing request.

RR/Route in req (TS is proxy)
Opaque information can be included in a record-route entry in the outgoing request. Any subsequent requests sent in the dialog by the receiver of the original request contain this opaque information in their route header.

This can never be updated! (see below)

RR/Route in response (TS is proxy)

Opaque information can be included in a record-route entry in the response. Any subsequent requests sent in this dialog, by the receiver of the response will contain this opaque information in their route header.

This can never be updated! (see below).

Contact (TS is UAS)
Opaque information can be added to the contact field in a 200OK. Any subsequent requests sent by the receiver of the 200OK will include the opaque information in the request URI.

This can be updated with an (expensive) re-INVITE/200-OK/ACK sequence (e.g., initiated by the TS as UAC in the context of the dialog).

Contact (TS is UAC)
Opaque information can be added to the contact field in an INVITE or re-INVITE. Any subsequent requests sent by the receiver of the INVITE will include the opaque information in the request URI.
This can be updated with a re-INVITE, as described above.

Contact (TS is Proxy)
Opaque information can be added to the contact field in an INVITE or re-INVITE. Any subsequent requests sent by the receiver of the INVITE will include the opaque information in the request URI. In case of a proxy, the opaque information added to the contact has to be removed before forwarding the subsequent requests. This is a 'small' violation of the SIP spec, but since it is not noticeable to the clients, this is acceptable.

This information can be updated with a re-INVITE, as described above.

The reason the RR information can not be updated can be found in the RFC3261:

```
12.2 Requests within a Dialog

   …
   Requests within a dialog MAY contain Record-Route and Contact header
   fields.  However, these requests do not cause the dialog's route set
   to be modified, although they may modify the remote target URI.
   Specifically, requests that are not target refresh requests do not
   modify the dialog's remote target URI, and requests that are target
   refresh requests do.  For dialogs that have been established with an



Rosenberg, et. al.          Standards Track                    [Page 72]

RFC 3261            SIP: Session Initiation Protocol           June 2002


   INVITE, the only target refresh request defined is re-INVITE (see
   Section 14).  Other extensions may define different target refresh
   requests for dialogs established in other ways.

      Note that an ACK is NOT a target refresh request.

   Target refresh requests only update the dialog's remote target URI,
   and not the route set formed from the Record-Route.  Updating the
   latter would introduce severe backwards compatibility problems with
   RFC 2543-compliant systems.
```

The conclusion is that the cookie mechanisms available in SIP are not suitable for transporting frequently changing cookie information, such as versioning information.

2.13.1.4 Use of Cseq

The suggestion here is to use the CSeq number as already defined in the SIP protocol for similar purposes as the version information.

Unfortunately, there are some differences between a version number and a CSeq number.

- In a SIP dialog there are two CSeq numbers; one for transactions flowing from caller to callee and one for transactions initiated in the other direction.
- A message will only contain the CSeq number of the transaction it pertains to.
- There can be multiple messages in a transaction, all of these will contain the same sequence number (e.g, the INVITE, all the responses to it –e.g., 100 trying, 180 ringing, 200 OK—and the ACK will all contain the same sequence number).
- The UA has to increase the version number by exactly one for each new request it sends (except for the ACK and the CANCEL)
- The container is responsible for the CSeq numbers of the application that is acting as an UA.

- There are strict rules on how a UA must react on inconsistent CSeq numbers. These are implemented by the container sailfin, however, as stated before gaps in the sequence numbers, while not normal, are allowed.
- Proxies do not normally check on CSeq, it is the responsibility of the UA to implement said rules.
- Due to the use of authenticating proxies, gaps can occur in the Cseq number range. The UA may not consider this a fault situation.

The consequence of these differences is that

- The UA only indicates a part of the overall ‘version’ in each request.
- There is no ‘total’ version number, so we can not really use the framework support for versioning directly.
- We can do a best effort match
- We might suspect that we have a stale version based on the Cseq, but due to fact that there may be gaps we can never reject access to a stale session. Instead we have to continue the session with the highest version number found.

A figure to illuminate the concept.
>x is the Cseq for requests originating from UA1
<x is the Cseq for requests originating from UA2
(x,y) is the ‘combined state’ of the session as indicated by the two sequence numbers.

```
UA1            PROXY(TS)       UA2
  >INVITE(>1)> 0,0
               1,0 >INVITE(>1)>
               1,0 <200OK(>1)<
  <200OK(>1)<  1,0
  >ACK(>1)  >  1,0
               1,0 >ACK(>1)>
  >INFO(>2) >  1,0
               2,0 >INFO(>2)>
               2,0 <INFO(<1)<
  <INFO(<1)<   2,1
               2,1 <200OK(>2)< *
  <200OK(>2)<  2,1
  >200OK(<1)>  2,1
               2,1 >200OK(<1)>
```

For example at the point indicated in the sequence by * the only information the system gets is that the UA2 is responding to a request (sent by UA1) with sequence number 2. Based on this information it can not determine conclusively if the state it has in the active cache corresponds to the latest. Either the (2,0) or the (2,1) ‘version’ would satisfy the requirements.

The conclusion is that the Cseq numbers can also not be used for versioning. They could be used for a (limited) staleness check, but given that there may be gaps in the Cseq, this is also not useful. So instead of relying on the Cseq numbers we will

perform the load request as proposed earlier. This means we will always use the active version if available and if not available we will use the highest version in the cluster.

Versioning handling for converged HTTP session. It might happen that a request initiated by an EJB updates the CHS session (via the SAS). Such an out-of-band request will not update the version number. This limits the use of version numbers in the staleness check for HTTP, so theoratically it can happen that those changes get lost without the client being aware of this.

Decision: The Cseq mechanism could be used to get a hint about staleness, but can not truly detect staleness. Therefore, the Cseq mechanism will not be used! Instead we will rely on the fact that an active version is never stale. ~~And~~If implemented, we can also rely on the expat-list handling to do version aware load requests to get the highest version in the cluster.

2.14 Lazy reactivation based on timers

In several usecases there are situations, i.e., after a failure or a shutdown, where some sessions do not have any active copy. There will be only a replica copy until there is external activity on the session (lazy reactivation based on traffic). The advantage of this is that the re-activation is then controlled by the Load Balancer, and hence the re-activation is done on the correct instance.

However, in the SIP domain, not all activity is driven by protocol messages. There are two timer mechanisms. One is the SIP application Session timer, which controls the lifetime of the SIP application. If this timer expires and no extension is requested or granted, then the SAS become eligible for invalidation. Then there are the application timers. Applications can create many of these and they will notify the application when they expire (in that sense these are similar to the EJB timer service timers).

During the repair period after a shutdown or failure these timers ~~should~~must still fire, although a delay in firing is acceptable. Since the repair period can be quite long as it is based on lazy activation, and since it might even happen that we have ‘hanging’ sessions, with no external activity, it is not an option to wait for external reactivation.

To handle this the proposal is to introduce a deserialised field (let's call it ~~nextFiring~~ExpirationTime) in the replica of the timer for the application timers. This field will indicate the next firing time for application timers. A similar external field is also introduced for the SAS, indicating the next expiration time of the SAS. Whenever the active copy of the timer is updated, the ~~nextFiring~~ExpirationTime of the replica is also updated.

In addition to the ~~nextFiring~~ExpirationTime the replicas also contain an 'source' field. This indicates the server instance that last replicated the session. ~~However, this field is not used in the context of the expired timer firing on the replica at the moment.~~

Whenever the owner (source) instance is gone (detected by GMS), all the instances that have a replica from this source will become responsible for ~~the replica partner will~~ handling~~e~~ the timeouts for the replica application timer and the SASes. The majority of these will be on replication partner, in a normal situation.

The timers on the replica will fire some time after it expires. The added grace period is to avoid replica timers from firing in the case of temporary failures. This grace period should be greater then the GMS delay, for this reason.

There are two implementation options:

- the replica partner uses a thread to periodically scan the replica cache for any timer instances that were present in the cache when its source (owner) went down. ~~where the next firing time has expired.~~
- The buddy starts timers (using the servlet timer implementation which already is optimised for a large number of timers and uses a pool of threads). When the replica is removed from the replica cache (e.g., after it is reactivated) the timer is cancelled. ~~When the nextFireTime value is updated the timer is rescheduled.~~ When scheduling the timer, the grace period must be taken into account.

When the timer expires on the replica this can have several reasons:

1. The active copy is not present anymore, e.g., due to a crash
2. The active copy is invalidated, but the replica was not removed (the replica is a zombie)
3. The active copy does exist, but for some reason failed to update its replica with the updated time (e.g., ~~a replication message was lost~~because it changed replication partner and we did not purge the replica)
   ~~The active copy does exist, and the replica is fresh, but the active copy failed to fire (e.g., due to high CPU load)~~

Only in the first case we should fire replica timer. However, the second case can never be distinguished from the first case! So, also in the second case the timer will fire on the replica (potentially leading to unexpected results, such as messages being sent on expired sessions).

When the timer expires on a monitored instance in the replica, we will sent a message to the current home instance. This message is called a load-advisory. It will indicate that the home instance should issue a non-version aware load request to obtain an active copy of the object. The advantage of re-activating the item on its home instance is that the burden of the timer expiry handling is spread over all the instances in the cluster and that (barring futher cluster shape changes) the item does not have to be migrated if there is external traffic received on it.

If the home instance already contains an active copy then (barring network segmentation) that version must be the highest version. The response to the load

advisory will then be an indication that the sender should remove its replica. And no load request is done (a load request would have resulted in the same outcome but is more expensive).

To handle the latter cases and also the case where there are multiple replicas of the same object, a check is introduced to check for the staleness of the replica when the timer expires. This is accomplished by a 'peek'. This is similar to the non-version aware load request, except that each instance only returns the version of the requested object and not the actual contents. Instances that have a version lower then the requested version in the peek should remove this version from their cache, but any higher versions will remain. It should never happen that there are two replicas with identical versions!

If the peek shows that there are other higher versions in the cluster, the replica timer will not fire. Instead the object (SAS or Timer) will remove itself.

If there are no higher versions in the cluster, then the timer or SAS will issue a targeted load request. It will check which instance is the home instance, based on the consistent hash in the replica. Then it will send a message to the home instance. The home instance will issue a version-aware load which will load the replica.

There are rules on how to handle any missed deadlines when we activate an instance due to a timeout (see [FSD]. This is an implementation detail and not described here.

So in summary; We will include the nextFireTimeexpirationTime as a deserialised field in the replica. It is based on the next time the timer should fire including a grace period. We will actively monitor timeouts of SAS and Timer replicas for instances where the owner died and activate these on the home instance if the replica is not stale. The timeout is handled on the home instance.

2.14.1 Cleanup activities

The general idea for the cleanup handling is that no child should outlive its parent. For this reason any updates to the expiry timeexpirationTime of the parent are propagated to the children. By including a grace period that increases the farther down the tree we go, we give the parent the time to handle any expiry before the child is expired. So if the expiration time on a replica child has expired before the parent updated it, then we can assume that the parent does not exist and remove the child as well. It is clear that the grace period must be sufficiently large.

The SAS is the parent of the tree and is reactivated by the mechanism described in the previous section. The reactivation can be the trigger to update the children's expiration time as well.

However, the expiry time of an ST/SAS can be very large or even infinite (which is possible with SAS timers according to the SSA specification). This means that relying only on the timer mechanism described above for timers some objects might never be re-activated or reactivated very late.

For this reason, the reactivation mechanism described in the previous section is enhanced with the following properties:

- The re-activations of all items is spread over a (configurable?) time window after the GMS notification.
- The upper bound of this time window is configurable; all items will be re-activated before this time window.
- An SAS/ST can be re-activated before its expiration time, but never after its re-activation time (taking some timer slack into account).
- There is a lower bound to the time window. Before the lower bound only items whose timer expires are re-activated. After this lower bound also items whose expiration time has not yet passed my be re-activated.
- Children are never re-activated before their parents.

The latter 'requirement' is introduced to avoid unnecessary migration. After a failure, there is a large chance that the instance is started within a certain time. We should avoid re-activating the items on the current home instance only to have to migrate them when the failed instance is restarted.

The above holds for re-activations caused by a failure (GMS notification). These will be handled more efficiently, since the exact set of items that needs to be monitored is well-defined.

As a backup solution there is also the cleanup task. This task will have similar handling, except that in this case this is not triggered by failures.
What exactly this means and how the two mechanisms interwork is still unclear.

We will look at each object in turn.

SAS

The SAS has a timer. This timer is monitored by a process as described in the section on timer expiry on the replica.
The SAS will never be cleaned up automatically, instead it is reactivated on the home instance.
The following pseudocode applies on the replica

```
void handleTimeoutOnReplica(SASReplica replica) {
      // timeout is triggered on externalized timer + grace period
      long highestVersionInCluster = peek(replica.id);
      if (!highestVersionInCluster > replica.version) {
            remove(replica);
            // since the children are orphaned they will be removed as well, eventually
      } else {
```

```
                ServerInstance home =
UserCentric.applyConsistentHash(replica.hash);
                loadOnTarget(home, replica.id, replica.version);
                // this instance will receive a load request for the specific
instance.
                // this is handled normally (return replica and remove)
        }
}
```

The following pseudocode applies on the home instance

```
voidboolean processTargettedSASLoadAdvisory(long id, long version) {
        if (activeCache.getSas(id) != null) {
                return true; // indicate delete to requestor
        } else {
                SAS sas = versionAwareLoad(id, version);
                // during deserialisation the timers are started.
                Sas.activate(); // starts timer & timer handling in new thread
                return false;
        // since the timer expires immediately
        // do not yet load the children, since this is wasted effort if sas
is invalidated
}
```

The following pseudocode applies on the home instance

```
void activate() {
        if (sas.getExpiryTime > currentTime) {
                int t0 = sas.getExpires();
                SipApplicationSessionListener sl = sas.getSessionListener();
                sl.sessionExpired(createEvent(sas));
                int t1 = sas.getExpires();
                if (t1 > t0) {
                        // extended the session
                        tree = loadAllChildrenAndGrandChildren(sas);
                        updateExpiryTimeInSSes(tree);
                        updateExpiryTimeOnDFes(tree); //only if later time
                } else {
                        remove(sas);
                        // children will be orphaned and eventually removed
                }
        } else {
                // not yet expired - load the children
                tree = loadAllChildrenAndGrandChildren(sas);
                scheduleTimer(sas.getExpiry())
}
}
void handleTimeout(SASImpl sas) {
        int t0 = sas.getExpires();
        SipApplicationSessionListener sl = sas.getSessionListener();
        sl.sessionExpired(createEvent(sas));
        int t1 = sas.getExpires();
        if (t1 > t0) {
                // extended the session
                tree = loadAllChildrenAndGrandChildren(sas);
                updateExpiryTimeInSSes(tree);
                updateExpiryTimeOnDFes(tree); //only if later time
```

```
        } else {
                remove(sas);
                // children will be orphaned and eventually removed
        }
}
```

SS

The SS has the same expiry time as its parent SAS plus an additional grace period.
The following pseudocode executes on the replica instance:

```
void handleTimeoutOnReplica(SSReplica replica) {
        // timeout is triggered on externalized timer + 2 times grace period
        remove(replica);
        // no check needed on replica.parent since it must have expired
before
        // the SS (since we added the extra grace period)
        // expiry of the SAS should have removed it or migrated it (and the
SS)
        }
}
```

Timer

The TImer has a expiry time for itself, unrelated to the SAS (it can be longer or shorter).
The following pseudocode executes on the replica instance:

```
void handleTimeoutOnReplica(ReplicaServletTimer replica) {
        // timeout is triggered on externalized timer + 2 times grace period
        SASReplica sas = replica.getParent();
        if (sas == null) {
                // no parent
                remove(replica); // remove the timer
        } else {
                // SAS parent exists, apparently it did not expire yet
                // so the timer expired before the SAS
                long highestVersion = peek(replica.id);
                if (!highestVersionInCluster > replica.version) {
                        remove(replica);
                        // XXX what about the SAS parent? Should we also remove
that?
                        // or wait until it expires itself?
                } else {
                        ServerInstance home =
UserCentric.applyConsistentHash(replica.hash);
                        loadOnTarget(home, replica.id, replica.version);
                        // this instance will receive a load request for the
specific instance.
                        // this is handled normally (return replica and remove)
        }
        ServerInstance home = UserCentric.applyConsistentHash(replica.hash);
        loadOnTarget(home, replica.id, replica.version);
}
```

The reactivation of the Timer will trigger the reactivation of the SAS as well. This could trigger reactivating the rest of the children as well, but it does not need to.

DF

The expiry time of the DF is the maximum of the expiry time of its SSes.
When the SS is migrated in the scenario above it takes the DF with it and update its expiration time if the new SS expiration time is bigger.

```
void handleTimeoutOnReplica(DFReplica replica) {
      // timeout is triggered on externalized timer + 3 times grace period
      remove(replica);
      // no check needed on replica.parent since it must have expired
before
      // and should have migrated (taking us with it)
      // or the parent should have been removed.
      }
}
```

~~**Extra timer**~~

~~The expiry time of the SAS now serves two purposes;~~

- ~~potential expiring the SAS~~
- ~~cleanup delay for the children~~
- ~~activation of the SAS, resulting in activation of the children.~~
  ~~This will help in the repair actions (after time out two replicas will be available again).~~

~~The latter two could also be achieved with a separate timer, e.g., a refresh timer, which can be separate from the expiry time of the SAS. At refresh, if the SAS exists it is reactivated together with all its children, thereby avoiding cleanup of the children. If the SAS does not exist at the refresh, the children will not be refreshed before their grace period expires and they will be removed.~~

~~The refresh timer is specifically interesting if the SAS timer is set to a very long value Or to 0, which means an never ending SAS according to the current spec, although Ericsson wants to change this in the spec.~~

2.15 Shutdown

Shutdown is handled similar to a failure. The main difference is that it will be preceded by a sort of **quiescence** period.

After the shutdown there is a repair period during which **lazy reactivation** and **lazy replication** repair the lost cache information.

2.16 Quiescence

The purpose of quiescence is to limit the the loss of transactions and sessions when the shutdown is done.

There are several levels of Quiescence. Not all of these deserve the name quiescence, though the effects are similar:

- SAS starvation

  Request that belong to an ongoing Sip Application session are still handled by the instance, but any requests that start a new session are routed to a backup.

  advantages

  - at the time of shutdown, there are not so many sessions that have to be re-activated on other instances. This keep the amount of replication data down.

  disadvantages

  - it can still happen that the actual shutdown happens in the middle of a message or a transaction, in which case this message or transaction is lost.
  - It is impossible to detect whether a request belongs to an ongoing SAS!

- SS session starvation

  Request that belong to an ongoing Sip session are still handled by the instance, but any requests that start a new session are routed to a backup.

  advantages

  - at the time of shutdown, there are not so many sessions that have to be re-activated on other instances. This keep the amount of replication data down.

  disadvantages

  - it can still happen that the actual shutdown happens in the middle of a message or a transaction, in which case this message or transaction is lost.
  - Since the starvation happens on SS level, new SSes related to ongoing SASes will already be routed to the new instance. This means that the SAS is being accessed from two instances.

- Message starvation.

  The instance is given the time to finish handling the ongoing messages. Any new messages are routed to the backup.

  advantages

  - The shutdown does not impact ongoing messages.

  disadvantages

  - The amount of sessions to reactivate will not dramatically decrease during the quiescence period (only those which were in the process of terminating anyway).
  - If the message was not the end of a transaction, the SIP transaction is lost (since transactions are not replicated so the backup is unaware of the ongoing transaction whenever a message for that transaction is received on the backup).

- Transaction starvation.

  The instance is given the time to finish handling the ongoing transactions. Since in SIP the transaction data is not replicated, the backup instance can not

continue the ongoing transaction. Any new transactions are routed to the backup.

advantages

- at the time of the shutdown not many transactions are lost.

disadvantages

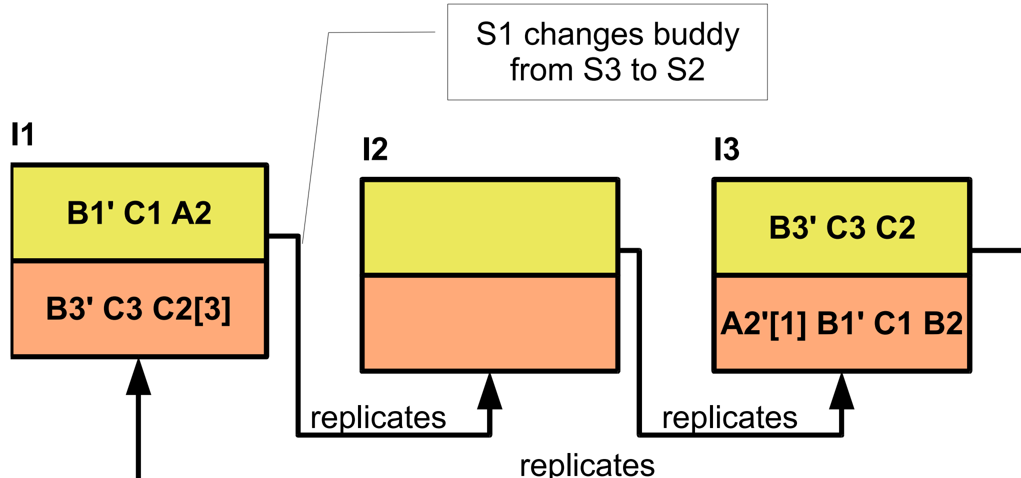
- multiple transactions can occur simultaneously in the same SIP dialog or session, which can result in the same session being accessed on different instances
- In order for the LB to implement the quiescence it would have to be quite protocol aware (at least the start and stop of the transactions).
  Currently responses are already routed via the VIA header and will end up on the instance where there the request originated from. However, CANCEL and ACKs are more difficult to route without keeping both state and awareness of the protocol.

Since there are very strong disadvantages for the SAS and SS level starvation, the only real candidates are message and transaction starvation.

We can give a hint to the administrator to shutdown instance as follows: Disable the instance from the LB by issueing a asadmin command. Wait a short while to give the replication framework a chance to flush its data (message quiesceing) and potentially a bit longer to give some responses the time to be handled. Only then shutdown the instance.To ensure that the flushing completes is difficult, responses might lead to new replications and might also lead to new requests being sent (and new responses etc.).
Then in the future, the LB could implement a nicer form of quiescence where also subsequent requests related to transactions are routed using to the quiescing instance (i.e., CANCEL and ACKs).

2.17 Recovery after failure

When an instance is recovered after a failure the cluster is reshaped. I.e., if I2 is recovered, I1 will start replicating to I2 again, where it was first replicating to I3.



After the cluster reshape we have different data to repair

- replicas lost due to the failure
  These will be repaired using **lazy replication.**
- Active copies lost due to the failure which have not yet been reactivated.
  These will be repaired using **lazy reactivation.**
- Active copies that were reactivated during the failure period (either due to time out or due to traffic).
  The UC load balancer will route traffic meant for such sessions back to the recovered instance. When this happens, these will be (lazily) **migrated**.

The migration is described in its own section.

Note that any replicas that I1 created on I3 during the failure of I2 will, due to lazy replication, be created now on I2. The old (stale) copy remains on I3. This has to be handled very carefully, since the stale copy on I3 can potentially become a zombie.

There are several possibilities to avoid zombies in this case

- purge all the replicas that I1 created on I3 when I1 gets a new replication partner. This leaves the system in a more vulnerable state.
- Purge the replica on I3 when the first replica is created on I2. I.e., creation of a replica with a version higher than 0 (meaning it is a repair and not a normal replication). This can be detected by the replication target and then it can broadcast a purge for this version.
- Purge the replica on I3 when the first replica is created on I2, but now triggered by I3. When it receives a new replication partner I3 marks **all** its active versions as a possible source for duplicate replicas. When the first replica is created after the cluster reshape, I3 broadcasts a purge for that version.
- Purge the replica on I3 when the replica is created, but now based on the expat list. When the expat list indicates that there is a copy of the data on another instance then the replication partner, this information can be used to purge the

copy when the first replication is done after the cluster reshape. This is more efficient solution since it only removes duplicates that are known to exist.

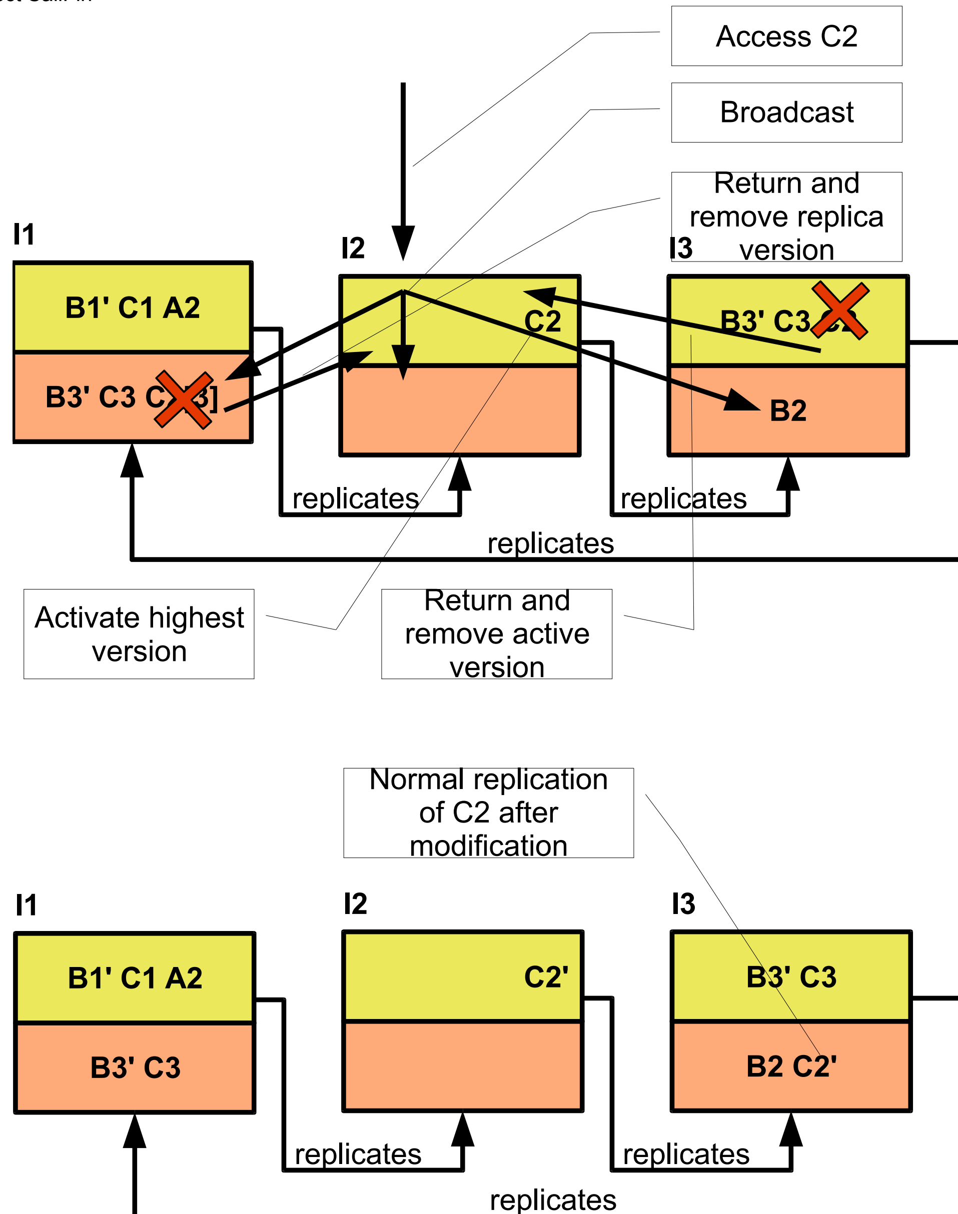
Zombie prevention is not yet addressed in the solution!

2.18 Migration of active sessions

The User Centric Load balancing strategy will ensure that after a recovery of an instance traffic originally directed to the failed instance is routed back to the recovered instance.

Then sessions that were reactivated on another server instance during the failure situation have to be migrated back to the recovered instance.

As the figure shows, a normal load request is performed (either version aware or not depending on the expat list handling). However, in the broadcast also active versions are returned and removed. Effectively, this means that the session (C2) is migrated from I3 back to I2.



The migration is only applicable in the case of user centric routing. In case of a RR load balancer policy, the sessions would never have to be migrated, since the request remain sticky to the failed-over instance (I3 in the above case), even after the restore of I2.

In the current replication framework the load request would lead to multiple active versions. And the framework would rely on the version awareness to avoid using stale versions. Multiple active versions should be avoided in SIP, because of the

possibility of multiple timer expiries of the same object and the whole problem with versioning.

Therefore, we extend the functionality of the load request as described earlier. I.e., remove all the replicas *and any active version* during the load request. This should be done in a transaction, i.e., they should not be removed until the reception of the new version is acknowledges by the requestor.

2.18.1 Remote Locking

During the migration it might be possible that the active copy that must be migrated is currently being accessed. This constitutes a form of concurrent access and can lead to inconsistencies in the data (e.g., lost modifications).

To combat these cases, we propose to lock the objects when they are being accessed. When a load-request broadcast is received for a locked active object, this is indicated in the response message (e.g., by including the lock field in the serialised representation). This response will cause the broadcast to fail. When initiated from a SIP request, this should result in a 503 error response.

This mechanism ensures (in a rather brute force way) that there is no concurrent access on an object.

Locking should be done whenever an object is loaded and released when the object is finished with it (e.g., when it calls the save). Extend the load-request broadcast with an error response. Handle this error response. We do not distinguish between read and write locks.

2.19 Upscale

From a user centric load balancer perspective there is no difference between a recovery and an upscale.

Also in this case the instance that gets a new buddy due to the upscale will issue a **purge** request, the damage of which will be repaired with **lazy replication.**

During an upscale 1/(n+1) of the traffic will be directed to the new instance. Since all of this traffic already has active copies on other instances, each first request to such a session will case a **migration.**

2.20 Restart

Restart is the same as recovery, for the purposes of this document.

2.21 Upgrade

Upgrade is treated as a shutdown and a restart for each instance.

The main difference between the upgrade and a normal shutdown/restart cycle is that in the upgrade case we do know that the server will be restarted within a limited time period. Furthermore, we have to make sure that restarts do not remove the only copies of sessions. This means we can not rely on lazy repair actions, since the repair period is too long for the upgrade. Instead we have to implement some form of eager repair.

During the upgrade of an instance we will suspend replication to the upgrading instance (effectively suppressing the cluster reshape during the upgrade). Any data that would be replicated to the buddy of the upgrading instance would have to be purged when the upgraded instance is restarted and is acting again as a replication partner.

When the upgraded instance is restarted it must be actively repaired to ensure that the original situation is recovered (excluding any sessions that were re-activated on another instance during the upgrade). The repair will need assistance from the neighbors of the upgraded instance.

This should ensure that there is a minimum loss of sessions, provided that servers are not being upgraded while they are involved in the repairing. There are two options:

- wait until the eager repair is finished before rolling to the next instance.
- skip a server during every roll. Since the repair is being done from the neighbors, those three instances (the recovered/restarted instance and both its neighbors) are vulnerable during the repair period. Skipping these will result in a faster upgrade (e.g., in a 10 instance cluster upgrade as 1, 3, 5, 7, 9, 2, 4, 6, 8, 10) Mmm, it might even be better to skip two servers for the load distribution (i.e., after upgrading 1, but instances 10 and 2 will be busy. After upgrading 3, 2 will be double busy...)

2.22 Eager repair after upgrade

There are two ways to restore the situation of the upgraded server as it was before the upgrade (minus any sessions that migrated during the upgrade); disk based repair and partner based repair.

The first is optimized for the case that the upgrade time per instance is short enough that the majority of the sessions have not been changed. It has the advantage that most of the repair handling is done by the upgraded instance. This should limit the throughput loss and the traffic loss (errors) due to overload instance during the repair.
E.g., PGM with a lot of sessions (300K per instance) of which relatively few are touched during the upgrade is the prime example of this.

The second option involves the partners more in the repair process. This can lead to more throughput loss and more traffic loss due to high CPU usage during the repair. However, if we assume short lived sessions, most of which are updated during the upgrade of an instance, the differences between the two blur.

2.22.1 Disk based repair

The repair will be done by storing the caches on disk and reloading these caches when the instance is restarted. Since the caches have become outdated while the server was upgraded the content in it is suspect. They are repaired using information from the replication partners. This is described in more detail in the following sections.

The procedure for upgrade now globally as follows:

Start

- set a 'global' flag that we are doing upgrade (once for the complete cluster). This ensures that instances are not syncing their configuration and apps with the DAS anymore.
- Backup the cluster configuration
- deploy the new version of the app on the DAS.

Roll

1. tell the load balancer to disable the instance (asadmin command). This means it is removed from the consistent hash.
2. wait a short while to give the inflight data from that instance the chance to make it to the replication partner.
3. shutdown the instance. If the 'upgrade flag' is set the instance will store its state to disk.
4. restart the instance. The instance will retrieve its configuration state and any new application version from the DAS.
   If the global 'upgrade flag' is set the instance will restore its state from disk.
5. after the restart/reload is finished (and the state is read from disk) the instance has to be enabled for the LB.
6. THen there are some repair actions to the stored state that was retrieved from disk, since it is outdated.
   The neighbors (which are involved in this repair) must not be rolled before the repair is finished.

Stop

reset the global flag.

Save and Reload

Save and reload are fairly simple. During the safe all the active caches are serialized, by serializing the cache object in which they reside (normally an hashmap or similar). The replica cache is already serialized and can be written quite simply.

During the reload the reverse happens.

Both the replica cache as well as the active cache needs to be updated.
This is most easily described in pseudo code. Note that this in no way suggests an implementation, e.g., the suspect marking can also be done by keeping two caches and moving entries from one cache to the other.

2.22.1.1 Repairing the active cache

The idea behind repairing the active cache is very simple. Any items that were in the active cache of upgraded instance should (under normal circumstances) be in the replica cache of the replication partner. During the upgrade, traffic meant for the upgrading instance is redirect to other instances. This will result in reactivaction of those sessions, which means that the owner changes. So during the upgrade the items in the replica cache that are still owned by the upgrading instance will only decrease (never increase). After the upgrade, we just have to remove all the instances from the reloaded active cache that have been reactivated elsewhere in the cluster. Until that task is completed, any items that have not yet been repaired in the active cache are marked as suspect and access on those will trigger a load request, even if it is found in the active cache (since in the mean time it may have been reactived elsewhere)

After reload, but before handling traffic, on the upgraded instance

```
for (CacheItem activeCopy : ActiveCache) {
      // for all the active caches.
      activeCopy.suspect = true;
      // marks the entry as suspect. It is not yet repaired, which might
      // mean it is removed, or update & reactived in the mean time
      // note that update&reactive always go together
}
```

When traffic is started; on upgraded instance

```
List<VersionAndId> versionlist =
getVersionListFromReplicaTargetsReplicaCache();
// get the list of versions and ids from the replication partners replica
cache.

for (CacheItem activeCopy : ActiveCache) {
      lock(activeCopy);
      if (activeCopy.suspect) {
            // not yet updated due to traffic
            replicaCopy = findInVersionList(versionList, activeCopy.id);
            if (replicaCopy == null) {
                  // no longer in replica cache of buddy
                  // so it is reactived elsewhere or removed
                  ActiveCache.remove(ActiveCopy);
            } else {
                  // it still exists in the replica cache.
                  // only possbile if it was not accessed in the mean time
                  activeCopy.suspect = false;
            }
```

```
        }
        unlock(activeCopy);
}
```

Traffic handling on upgraded instance

```
void find(long id) {
      activeCopy = ActiveCache.find(id);
      if (activeCopy.suspect)
            // always do a load request when suspect
            return nonVersionAwareLoadRequest(id);
      } else {
            return activeCopy;
      }
}
```

TODO:

If we have expat lists we need to update these as well.
There are several solutions:

- During the upgrade we could work without expat lists. This means doing the non-version aware load request. This might be best, especially if we are upgrading and/or repairing multiple instances in parallel. After the upgrade is complete we revert to the expat list handling (but how to know the rolling upgrade is complete?)
- We do the expat list handling during the upgrade. The expat list might replace the version list mentioned above. The upgraded instance can consider itself the owner of all its expat replicas located on its replication target. This assumes that the expat lists does distinguish between replica and active copies.

2.22.1.2 Repairing the replica cache

The replica cache repairing is a bit more difficult then the active cache repairing. Where the active cache can only shrink during the upgrade, the replica cache must be able to handle additions, modifications and removals that happened on the replication source of the upgraded instance during the upgrade.
Since the most common case is that nothing changed during the upgrade (we hope), the solution is optimized for that.

Again in pseudocode:

After reload, but before handling traffic, on the upgraded instance

```
for (ReplicaItem replica  : ReplicaCache) {
      // repeat for all the replica caches.
      replica.suspect = true;
      // marks the entry as suspect. It is not yet repaired, which might
      // mean it is removed, or updated in the mean time
}
```

When traffic is started; on upgraded instance

```
List<VersionAndId> versionlist =
getVersionListFromReplicaSourcesActiveCache();
```

```
// get the list of versions and ids from the replication partners replica
cache.

for (ReplicaItem replica : ReplicaCache) {
      lock(replica);
      if (replica.suspect) {
            // not yet updated due to traffic
            activeCopy = findInVersionList(versionList, replica.id);
            if (activeCopy == null) {
                  // no longer in active cache of buddy
                  ReplicaCache.remove(replica);
            } else {
                  // it still exists in the active cache.
                  If (activeCopy.version > replica.version) {
                        // updated in the active cache
                        // fetch it
                        triggerReplication(replica.id);
                  } else {
                        // still equal; no action needed
                        // we hope this is the most common case
                  }
                  replica.suspect = false;
            }
      }
      versionList.remove(replica.id); // can be more efficient
      unlock(replica)
}
for (VersionedItem activeCopy : VersionList) {
      // everything that is in the version list but not in the active copy
      // so this was created during the upgrade
      // since there is nothing there is also nothing to lock on...
      triggerReplication(activeCopy.id);
}
```

load request handling on upgraded instance

```
// this should not happen if there are no failures
void processLoadRequest(long id) {
      replica = ReplicaCache.find(id);
      if (replica.suspect) {
            return null; // act as if not present
      }
}
```

There is a worry about this approach in that network based IO can be quite expensive (e.g., in an NFS mounted file system).

An alternative to the suggested approach is that the replication source 'buffers' the changes it would make to the replica cache during the upgrade. This can be done on different levels, the simplest would be that the replication framework justs grows its internal queue during the upgrade and flushes it after the upgrade. Or, the act of replicating could be suspended, I.e., the dirty markings (in case of modified-session replication) would just remain. Then after the upgrade finishes, all sessions marked

as dirty in the active cache are replicated. With the latter solution, the disadvantage is that this would imply that also removed sessions would have to be retained until after the upgrade.

2.22.2 Partner based repair

In the partner based repair the upgraded instance does not restore its cache from disk, but purely based on the caches of the partners.

2.22.2.1 Eager replication

This currently is already implemented under the name repair-under-load or forward repair. It means that the replication source replicates the entire contents of its active cache.

Repair-under-load is named thus, since it will only replica those items in the active cache that are not yet touched since the cluster reshape.

The repair-under-load does not have to be updated. It should however, only be invoked in the upgrade scenario for the SSR case (which can be a configuration option).

2.22.2.2 Eager re-activation

We will try to restore the original configuration as it was before the failure (minus any sessions that have been reactivated on other instances during the upgrade).

The repair-under-load is an already existing mechanism, but the reverse repair is new.

The reverse repair could work as follows. Each session will maintain information about the origin of the session (i.e., where the active copy resides or the last instance that replicated this data).
During the reverse repair, the clockwise neighbor of the recovered or restarted node will copy all the replicas that originated from the recovered instance back to the active cache of the recovered instance.

This can be done by sending a list of all the ids in the replica cache, after which the receiver does the normal load-request procedure to actually load the sessions, one by one.

The reverse repair should be done ‘in the background’ without much throughput impact on the instance. However, when data is requested based on traffic, this should get priority over the ‘bulk copy’, to limit the latency impact.

Since eager re-activation is only used during the upgrade and replication is suspended in that case, it should be enough to just copy the complete replica cache

to the active cache of the neighbor. However, since the owner information is needed for other purposes as well (e.g., purge), the suggestion here is to implement it based on the owner information. This makes the reverse repair reusable in other situations as well (e.g., after a recovery).
An alternative to the eager re-activation could be the re-activation based on timers as described in the section on cleanup handling and timer handling on the replica. If the upper limit to the time interval for re-activating the replicas is chosen to be short this can be used during an upgrade to ensure that all the replicas are reactivate in a certain time. This is less efficient from a signalling perspective than the full eager replication described above. Also, the timer mechanism is triggered by the GMS failure notification and not the GMS join notification. The lower limit to the time interval should be set to the time it takes to restart the instance during the upgrade.

2.22.3 Memory-based repair

The future TSP-SAF solution to the application upgrade will be to:

- deploy a new version of the application in inactive state
- deactivate the old version
- 'repair' the new (inactive) version with the data from the old version
- activate the new version
- remove the old version

(step 3 and 4 could be reversed).

Such a scheme would allow an efficient migration of both the active and the replica cache between the old and the new versions of the application. The SSR would have to be aware of the relation between the two applications (from a glassfish perspective they are just two applications, where the version is coded in the name).

To migrate the replica cache between the application versions should be no problem, since this is in deserialised form anyway. To migrate the active cache the items in the cache would have to be serialised and deserialised to also migrate between classloaders. But still, this is within the same JVM and can be done quite efficiently.

Such a solution is FFS.

2.23 Design

The design will be similar to the HTTP. ~~converged load balancer~~.

Managers

- the builder pattern is used to create a manager based on the configuration
- In the converged container there will be one manager for the HTTP replication.
- there will be one manager for SS/SAS and Timer replication.
- The addressing using the appid (for the purposes of routing) is the same for both managers, with the exception that the first is prefixed with http: and the latter with sip:
- There will be one (singleton) manager for dialogFragments

- The appid for that will be based on a unique name (e.g., df:dialogFragmentManager)

Replication objects

The replication objects contain the serialised data and a number of slots. The extra slots are data that needs to be accessed on the replica cache or information that needs to be stored efficiently without updating the complete serialised object.
The following fields have been identified:

- lastAccessed Time – last time the object was accessed
- version – the version
- id – the identification
- source – the owner of the replica
- hashCode – the hashcode used to create the replica
- Cseq – the sequence number (for SS and –possibly-- for Dfs)
- nextFiringTime – the time object needs to be awakened, either for cleanup or because it needs to be activated.
- Parent – the id of the parent (is this really needed?)
- extra slot – for future extensions

For other design tasks see the task list.

3 Quality and Availability

<How do you handle availability concerns? Does this feature introduce any new failure modes? List testing and failure scenarios that quality team needs to worry about.>

4 Performance

<How do you want performance team to measure this sub-system? Any micro benchmarks necessary?Any goals? Anticipated scalability limits or goals?>

5 Management and Monitoring

<Describe how performance, management status, and diagnostic information is exposed. How does this feature handle dynamic configuration changes?>

6 Formal Interfaces

<How is this feature(s) configured by administrator? Does it introduce new commands or modify existing ones? Show syntax of expected administrative commands and response codes. What is the schema/syntax for new configuration in domain.xml? Show the DTD snippets later in this section. What are their default values? What are the validation rules? Think about the stability level for each of the above. Are you expecting that the proposed design will change?]

7 Packaging, Files, and Location

<Does this feature add new jar files or extend existing ones? Where are they located?>

8 Documentation Requirements

<List the required documentation to support this product feature.>

9 References

FSD	*Functional Specification for Sip Session Replication*
Jsr289	*JSR-289 Sip Servlet API 1.1*
Rfc3261	*SIP: Session Initiation Protocol*
easSsr	

10 Open Issues

- Performance
  There still is a big question mark whether we will be able to meet the requirement goals stated in [FSD]. Specifically:
  - Upgrade handling
    There are two ways of handling upgrade. Both have their own peculiarities.

    The Disk based variant might work well for large amounts of relatively static sessions, but less well for very volatile sessions.

    Also there is a worry for the disk based solution on the performance when saving the caches over NFS to a remote disk.

    The repair of the replica cache in the disk based repair, could be a lot more optimal is the changes would be buffered in the replication framework. If we can reach agreement on this, we could gain perfomance.

    For all eager repair actions, it is not clear how much they influence the CPU load and hence (via overload protection) can cause traffic loss. Throttling might have to be implemented.
  - find vs create
    Depending on the traffic case, the performance loss of the @SAK can be quite severe, given the find vs create problem.
- Zombie prevention
  We still have no good zombie prevention mechanism for the cluster reshape case.
- network segmentation
  In case of network segmentation multiple active copies might be created.
- The out-of-band triggering of replication could be optimized.
- Etc.