Distributed System Upgrade Scenarios

Sameer Ajmani
ajmani@lcs.mit.edu

MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139, USA
First Draft: October 21, 2002
Last Modified: November 7, 2004

1 Thor

Thor [**?**] is an object-oriented database composed of two types of nodes, object repositories (ORs) and front-ends (FEs). ORs provide persistent storage for objects and support transactions on them. FEs fetch and cache objects on behalf of client applications and interact with ORs to commit transactions to persistent storage.

1.1 Add invalidations

This upgrade adds invalidation messages to Thor. Servers send invalidations to FEs when an object they have read is modified by a committing transaction. FEs acknowledge invalidations once they have removed all invalid objects from their cache and aborted all transactions that read the invalid objects. When a server receives an ack, it removes the acked object from the set of outstanding invalidations for that FE.

Model We can model this upgrade as a new mutators to FEs, FE.invalidate(objID). By returning this method normally, an FE acknowledges the invalidation.

Future SOs By calling FE.invalidate(objID), an OR tells the FE which objects it should invalidate. Since the FE does not provide methods for observing the transaction state or aborting transactions, its future-SO cannot invalidate objects and so must cause the method call to fail.

Transform Function This upgrade adds state to ORs and FEs to track invalidations, but it does not change old state. The OR invalidation state is changed only when transactions commit, so the TF can initialize it to empty values. The FE TF initializes the FE invalidation state to empty values.

Past SOs This upgrade is backward-compatible, so the Past-SO is trivial.

Scheduling Function This upgrade affects all FEs and ORs, so every node must be restarted.

Since the FE future-SO does not support invalidations, we might like to upgrade all the FEs before upgrading any ORs. Unfortunately, this schedule is impractical in a large-scale system.

The OR SF must avoid causing any OR replica group to fail and must allow each OR enough time to recover before the next replica upgrades. Thus, the OR SF should upgrade one replica at a time, waiting for each replica to recover before upgrading the next. In pseudocode:

```
determine replica group by examining state
periodically:
  check other replicas' version numbers and status
  if (this replica has the lowest IP address
        among the non-upgraded replicas)
      and (all the replicas are up)
  then upgrade
```

This schedule isn't perfect: if a replica fails while another is upgrading, then the replica group may cease providing service. This follows from the fact that by upgrading a node, we are inducing an additional failure. If a group tolerates more than one failure, we can use this either to tolerate failures while nodes upgrade or to upgrade multiple nodes simultaneously.

We want to schedule FE upgrades to avoid interrupting client applications. This is similar to the problem of upgrading desktop machine operating systems. Simulation mode allows this upgrade to function normally (although with somewhat degraded service), so it is reasonable to delay an FE upgrade until its client applications have terminated.

1.2 Optimize invalidation protocol

This upgrade replaces inefficient invalidation protocol above with one that batches invalidations (similar to Thor's actual protocol). Rather than invalidating one object ID at a time, the OR invalidates sets of object IDs (one set for each transaction).

Model We can model the old protocol as FE.invalidate(objID) and the new protocol as FE.invalidate(setOfIDs). This upgrade is not backward compatible, since the new version does not support the old method.

Future SOs The FE future-SO must implement FE.invalidate(setOfIDs) by calling FE.invalidate(objID), since these mutators affect state that is common to the two versions. For each objID in the invalidation set, the SO calls invalidate(objID). If all these calls succeed, the SO causes the invalidate(setOfIDs) call to succeed.

Transform Function The pre-upgrade OR state is a per-FE list of invalidated objects, and the post-upgrade OR state is a per-FE list of sets (one set for each transaction). The OR's TF converts the per-FE list into a set.

The pre-upgrade FE state is a list of outstanding invalidations. The post-upgrade FE state is a list of sets of outstanding invalidations. Since the FE's TF does not know how assign outstanding invalidations to sets, it simply discards this state. We rely on the OR to resend the invalidations and thus initialize the FE state. This is possible because ORs store the "actual" invalidation state, whereas FEs just keep copies.

Past SOs The FE past-SO must implement FE.invalidate(objID) by calling FE.invalidate(setOfIDs). The SO can do this by putting the objID in its own singleton set.

Scheduling Function This upgrade affects all ORs and FEs, so every node must be restarted. The OR and FE SFs for the previous upgrade can be used again here.

1.3 Add the MOB to Thor ORs

Thor [?] is an OODB that provides persistent storage of objects and supports transactions on those objects. Thor consists of object repositories (ORs) on servers and front-ends (FEs) on clients.

This upgrade adds the Modified Object Buffer (MOB) to ORs. The MOB lets ORs defer writing objects to disk when transactions commit. Instead, modifications are stored in memory and copied to the log. The modifications are lazily written to disk, which allows better batching of writes. If a client reads a page that includes objects in the MOB, the OR updates the page sent to the client to include the modifications. Finally, if an object is modified and subsequently deleted, the MOB lets the OR avoid ever writing that object to disk.

This upgrade contains one class upgrade that affects only ORs. It does not change the OR specification.

Future SOs Since this upgrade does not change any specifications, the simulation is trivial.

Transform Function This upgrade does not affect FEs, so they upgrade trivially. The OR TF, however, must allocate a substantial portion of memory for the MOB. Therefore, the OR TF needs some algorithm to determine how large the MOB should be given an OR's physical memory capacity.

When an OR restarts it recovers from its log and from the other OR replicas. This recovery code initializes the MOB from logged modifications. Subsequent page reads and modifications are processed by the MOB as usual. One unusual case is possible: a page read my include an object in the MOB, but it's possible that that modification was already written to the page before the upgrade. In this case, the OR can discard the object from the MOB.

Past SOs Since this upgrade does not change any specifications, the simulation is trivial.

Scheduling Function FEs upgrade trivially as soon as they hear about the upgrade. The OR SF for previous upgrades can be used again here.

TODO Scenarios below this point need to be updated to use our new model TODO

1.4 Add Multi-OR Transactions

This upgrade extends Thor to support multi-OR transactions. FEs are upgraded to support inter-site references via surrogates. ORs are upgraded to support surrogates, to execute two-phase commit (2PC) when necessary, and to maintain data structures for multi-site GC (i.e., inrefs and outrefs).

Model This upgrade changes the messages exchanged by ORs and FEs to include surrogates. OR.fetch(pageID) may return a page that contains surrogates; so upgraded ORs are not in-call compatible with non-upgraded FEs. Upgraded ORs are out-call compatible with non-upgraded FEs, since upgraded ORs do not initiate any new calls on FEs.

An upgraded FE can create inter-site references, so OR.commit(readObjs, modObjs) may have arguments that refer to objects at other ORs. Thus, upgraded FEs are not out-call compatible with non-upgraded ORs. Upgraded FEs are in-call compatible with non-upgraded ORs, since they support all calls initiated by non-upgraded ORs.

Upgraded ORs support 2PC; we can model this as adding two new mutators, OR.prepare(trans) and OR.commit(trans). Upgraded ORs also support multi-site GC; we can model this as adding two more mutators, OR.insert(ref) and OR.update(ref) (another scenario adds support for cyclic garbage collection). Thus, upgraded ORs are not out-call compatible with non-upgraded ORs. Upgraded ORs are trivially in-call compatible with non-upgraded ORs, since non-upgraded ORs make no calls on other ORs.

Transform Function Since the pre-upgrade states of ORs and FEs do not contain inter-site references, the TF does not need to transform those states. The OR TF changes the code to support surrogates, 2PC, and multi-site GC. Each of these changes will require new runtime data structures but does not change any old state. The FE TF also affects code only.

Scheduling Function By upgrading all ORs before upgrading any FEs, we avoid OR-OR incompatibilities. Since upgraded ORs are out-call compatible with non-upgraded FEs, no OR-FE out-call simulation is needed, either. However, once FEs start upgrading, non-upgraded FEs might fetch pages that contain surrogates from ORs. The FEs will not understand the surrogates and so will fail. Thus, OR in-call simulation is still needed.

Failure/Simulation The OR's in-call SO must prevent non-upgraded FEs from fetching pages that contain surrogates. Failure mode would simply cause such fetches to fail. This might be acceptable if the non-upgraded FEs are unlikely to fetch pages contain surrogates added by an upgraded FE. This may be the case due to "social" scheduling, i.e., all the members of a project team upgrade their FEs simultaneously, and they are the only ones that access a particular set of objects.

Alternately, we might imagine a simulation that attempts to convert inter-site references to intra-site ones. When the FE fetches a page containing surrogates, the OR's in-call SO could resolve those surrogates by fetching those objects, create a new page containing the fetched objects, and rewrite the surrogates to point to the newly-created page. The SO must somehow reverse this process when the FE commits a transaction that uses objects from other sites. Unfortunately, this scheme seems like a nightmare to get right (if it's possible at all).

1.5 Collect distributed garbage cycles

This upgrade adds distributed garbage cycle collection to Thor. The pre-upgrade system does not collect distributed garbage cycles at all; rather it just runs local tracing to collect local garbage and treats references from remote objects as roots. The post-upgrade system collects distributed cycles by back tracing. The upgrade adds messages to find garbage suspects, adds local data structures to enable back tracing, and adds messages to get trace information from remote sites.

Model The pre-upgrade system uses *insert* messages to add references to remote objects and uses *update* messages to remove references. The post-upgrade system adds *distance* information to these messages that lets Thor suspect objects likely to be cyclic garbage. We can model this upgrade as replacing old mutators OR.insert(ref) and OR.update(ref) with new mutators OR.insert(ref,dist) and OR.update(ref,dist).

The back tracing algorithm adds three new message types: a *trace* request for an inter-site reference, a *response* to that call, and a *report* sent to all participants in a trace. Each of these messages affects state the the receiver, so we model them as three new mutators, OR.trace(ref), OR.traceResponse(ref,state), and OR.report(ref,state).

Transform Function This upgrade adds state to the ORs to track the set of inrefs (insets) for each outref. Since a local safety invariant requires that this state be up-to-date, the TF must initialize this information. This is should be done automatically by the OR's recovery code, so the upgrader should not need to implement this.

Each inref / source site pair is assigned a distance for suspect tracking. When a new source site is added to an inref, Thor sets its distance to one. Therefore, the TF can safely initialize all distances to one.

Scheduling Function This upgrade affects all ORs but no FEs, so FEs move to the new version automatically.

Failure/Simulation This upgrade is incompatible both for in-calls and out-calls. Non-upgraded ORs might call the old insert(ref) or update(ref) mutators on upgraded ORs, so in-call simulation is needed. Upgraded ORs might call one of the new mutators on non-upgraded ORs, so out-call simulation is needed.

The in-call SO must simulate insert(ref) and update(ref) using insert(ref,dist) and update(ref,dist). There is no way to know the actual distance associated with the ref being inserted or updated. However, it is safe to assume that the ref has the distance one. Therefore, the SO simulates insert(ref) as insert(ref,1), and update(ref) as update(ref,1).

The out-call SO must simulate insert(ref,dist) and update(ref,dist) using insert(ref) and update(ref). Since the non-upgraded OR doesn't track distances, the SO can simply drop the distances from the method calls.

The out-call SO must also simulate calls to trace(ref) and report(ref,state). It does not need to simulate traceResponse(ref,state) since an upgraded OR will never make a traceResponse() out-call to a non-upgraded OR.

A call to trace(ref) requests that the receiving OR run a back trace on ref. The receiving OR calls back with traceResponse(ref,Live) if the ref is live, traceResponse(ref,Garbage) otherwise. Since a non-upgraded OR does not implement back-tracing, it cannot respond with the correct answer. However, it is safe to assume the remote reference is Live. Therefore, the out-call SO can simulate trace(ref) by discarding it and then calling traceResponse(ref,Live) on the calling OR. Alternately, the SO can simply cause trace(ref) to fail. The calling OR must assume that the remote reference is Live, which is the same behavior as above.

Finally, the calling OR might call report(ref,state) on a non-upgraded OR (i.e., to report the result of a previously-initiated trace). Since the non-upgraded OR does not support this method, the out-call SO must simulate it. We know that this report must be Live: the report would only be sent if the calling OR requested a trace on the non-upgraded OR; since the calling OR's SO always responds Live to such trace requests, this report must also be Live. A Live report just causes an old ref to be maintained, so it has no effect on a non-upgraded OR. Therefore, the out-call SO can simply discard the report() call.

2 Chord

2.1 Add proximity finger selection

Chord nodes select finger table entries at power-of-two intervals around the ring. In fact, Chord can still guarantee logarithmic routing if its fingers are *approximately* at power-of-two intervals. This flexibility in finger selection allows Chord nodes to optimize its fingers according to network proximity. This upgrade adds proximity finger selection to Chord.

Model This upgrade does not change any Chord node method signatures. However, it does subtly change the semantics of Chord's getFingers() method. Before the upgrade, finger table entries were guaranteed to be at particular intervals; the upgrade weakens this spec by allowing approximate intervals. By examining the Chord protocol and code, we can determine that non-upgraded Chord nodes will work with the weaker (upgraded) spec. Therefore, this upgrade is backward compatible.

Transform Function The pre-upgrade routing table does not contain proximity information, so the TF must initialize the post-upgrade routing table entries with default proximity information. This default should be some large value (those entries will be replaced by ones with lower proximity values) or should indicate "unknown proximity."

When a node restarts, it needs to know a bootstrap node. This is originally provided as a command-line parameter and is not available to the state transformer (not to mention the original bootstrap node may no longer exist). Instead the TF can bootstrap the node using its successor and finger list entries.

Scheduling Function This upgrade affects all Chord nodes, so it must restart every node. The SF must stagger node upgrades to avoid causing Chord or its applications to fail. Chord networks are dynamic and must restructure in response to failures. Restructuring requires communication, particularly at the layers above Chord that may transfer state between Chord nodes. Thus, an SF that causes large numbers of simultaneous failures will likely overwhelm the system with restructuring communication.

Instead, the SF should restart Chord nodes gradually. One might consider a schedule that orders the upgrades of nodes in the same replica group. Since replica groups are dynamic, this ordering is a moving target; i.e., the "minimal" node in the ordering will change as soon as the current minimal node restarts.

A simpler solution is to randomize the schedule. Periodically, each node flips a coin; if heads, the node restarts. By appropriately setting the probability of heads and the period between flips, one can bound the number of simultaneous failures. This requires that the upgrader know approximately how long a node upgrade takes (including state transfer).

Virtual nodes complicate scheduling further since an upgrade of a physical node requires that all its virtual nodes restart simultaneously. We could schedule this by shutting down individual virtual nodes according to the per-node schedule. Unfortunately, this causes a physical node to wait until all its virtual nodes shut down before it restarts any of them, so the Chord ring might get very small (or even disconnected) during the upgrade. A simpler and safer solution is to schedule physical nodes individually: when a physical node is scheduled to upgrade, it shuts down all of its virtual nodes (gradually, to allow for state transfer).

Failure/Simulation This upgrade is backward compatible, so no simulation is needed.

2.2 Add recursive lookups

Chord implements lookup(ID) iteratively: the node running the lookup contacts successive nodes to fetch their routing tables. A more efficient scheme is recursive lookup: each intermediate node continues the lookup on behalf of the calling node. This halves the number of messages per lookup.

Model We can model this upgrade as changing the semantics of n.lookup(ID). The old n.lookup(ID) returns the closest node to ID that node n knows about. The new n.lookup(ID) returns the closest node to ID that *any* node knows about (modulo network dynamism). Thus, callers of the old lookup(ID) method must iterate, while callers of the new method need not do so.

Transform Function This upgrade does not add or change any node state.

Scheduling Function As above, this upgrade affects all Chord nodes.

Failure/Simulation This upgrade is in-call compatible: old nodes can call the new lookup(ID) method without a problem. However, since Chord is peer-to-peer, new nodes may also call on old ones. The old lookup(ID) method does not satisfy the new method's spec, so this upgrade is not out-call compatible and simulation mode is necessary.

When a new node calls lookup(ID) on an old one, the new node's SO must simulate the recursive lookup behavior. This is straightforward: the SO just implements the original iterative lookup. If the lookup happens to encounter an upgraded node, that node will continue the lookup recursively.

Note that this SO keeps state about outstanding lookups, but this is soft state. Therefore, this SO's state can be discarded at any time.

2.3 Add DHash block type

This upgrade adds a new block type to DHash. Currently, DHash supports content-hash blocks and public-key signed blocks. Possible new block types include: public-key blocks that use a different signature algorithm, local-name blocks that map a key-name pair to value, and DNS blocks that map a DNS name to records that resolve that name. The common attribute of all DHash block types is that the ID-value mapping is verifiable, typically using cryptographic checks and application-specific routines (like checking that some DNS records resolve a given DNS name).

Model We can model this upgrade as adding a new getter/setter pair for block type "NewType" to DHash nodes. n.setNewType(ID,value) asks node n to store value under the key ID. Node n knows how to verify this block, since the method name indicates the type of the block (alternately, we could model the block type as a third method parameter). Similarly, n.getNewType(ID) returns the value previously assigned to that block. DHash nodes are may call these methods on one another, e.g., to do state transfer due to Chord ring membership changes.

Since upgraded nodes support all old DHash block types, they are in-call compatible. However, they are not out-call compatible with non-upgraded nodes, so out-call simulation will be required.

Transform Function Since a DHash node stores its blocks in a local database, the TF may need to define a new database schema to store the new block type (e.g., if the new blocks are a different size from old ones). Once the node has restarted, in must fetch all blocks for which it is responsible. This is part of standard DHash recovery, and so does not need to be implemented by the TF.

Scheduling Function This upgrade affects all DHash nodes, so all nodes must be restarted. Restarting DHash nodes is much like restarting Chord nodes, except that nodes may require a long time to do state transfer before they have actually restarted. If the SF simply restarts nodes gradually, the upgrader must make the period between restarts long enough to allow for state transfer.

A better solution might be for nodes to wait until any upgrading neighbors have completed state transfer before they upgrade. The SF must specify a predicate that lets a node determine when it can upgrade (e.g., I can upgrade once all my odd-numbered neighbors have upgraded and completed state transfer). The problem with this is that, in a dynamic system, state transfer is always happening, so this predicate may never be satisfied. The simpler randomized algorithm avoids these complications but risks causing too many nodes to upgrade at once.

Failure/Simulation Since non-upgraded nodes do not support the new methods, upgraded nodes need out-call SOs to simulate that behavior. We might imagine simulations that store blocks of the new type only on upgraded nodes, but this may overload those nodes and cause lost of unnecessary state transfer.

Instead, consider failure mode: if a call to n.getNewType(ID) fails, the calling node simply retries the call on n's replicas. If any of n's replicas have upgraded, they can satisfy the call. If a call to n.setNewType(ID,value) fails, then there are two possibilities: either n is dead (so this failure is transient and the caller should retry) or n is full (so the call might instead try to store on n's replicas). The caller's SO can simulate either of these failure modes, depending on which yields the more appropriate behavior for the application.

3 Traffic Sensors

Imagine an automobile traffic control system. The system consists of two types of nodes, lights and pads. A light node controls the traffic lights at a particular intersection. A pad monitors road pressure in a particular lane in front of a traffic light. The pad nodes surrounding an intersection report pressure status to that intersection's light node. The light node, in turn, controls the traffic light to maximize traffic flow.

Model We can model a pad's pressure state report as a mutator on a light sensor, light.setLanePressure(lane, hasPressure).

3.1 Add cameras

This upgrade adds camera nodes to the system. Each camera node is attached to a video camera that monitors a lane of traffic (i.e., there is one camera per pressure pad). The camera nodes use the video data to approximate the number of cars in their lane and report this data to their intersection's light node. The light node uses this information in conjunction with the pad data to improve traffic flow (e.g., by lengthening the light cycle for lanes with long lines).

Model We can model a camera's lane length report as a mutator on a light sensor, light.setLaneLength(lane, length).

Transform Function Camera nodes must be initialized with the ID of their intersection's light sensor. This can be done via a centralized database, a distributed discovery system, or a naming service.

The light node's TF expands its state to include lengths for each lane. These lengths should be initialized to the value "unknown," since no default numeric value is correct in all cases.

Scheduling Function Since this upgrade adds a method to light nodes, they are in-call compatible with non-upgraded nodes. We assume there is some way to upgrade light nodes without interrupting service (e.g., because light nodes are replicated).

A camera node can be added at any time. However, a camera node is out-call compatible only with upgraded light nodes. We can avoid this incompatibility by adding camera nodes only to upgraded lights. Otherwise, failure or simulation mode is needed.

Failure/Simulation If a camera node is added to an intersection with a non-upgraded light node, the camera node's calls to setLaneLength() must fail or be simulated. Failure mode is acceptable, since this mutator has no effect on a non-upgraded light node's state. We might also consider simulating setLaneLength(lane, N) as setLanePressure(lane, true) when N is greater than zero. This provides some additional redundancy in case a pressure sensor fails, but is probably unnecessary.

3.2 Trigger cameras from pads

This upgrade decreases power consumption by turning off cameras unless the corresponding pad detects pressure in the camera's lane.

Model We can model this upgrade as adding two new mutators to camera nodes, turnOn() and turnOff(). Pad nodes must be upgraded to call these methods on their corresponding camera nodes.

Transform Function This upgrade adds an on/off flag to the camera node state. The TF must initialize this state to either "on" or "off;" since the camera's pad node may not yet have upgraded, the camera should start in the "on" state. This upgrade does not change pad node or light node state.

Scheduling Function This upgrade does not affect light nodes. Upgraded camera nodes are in-call compatible with non-upgraded pad nodes, but upgraded pad nodes are not out-call compatible with non-upgraded camera nodes. Therefore, camera nodes should be upgraded before pad nodes. Since each pad node calls on only one camera node, a pad node only waits until the its camera node has upgraded.

The SF for a camera node upgrades immediately. While the camera node upgrades, the pad provides data to the light node. The SF for a pad node waits until its camera has upgraded. The camera node restarts in the "on" state, so it can provide data to the light while the pad node upgrades. When the pad node restarts, it can turn off the camera node.

Failure/Simulation Since we upgrade camera nodes before pad nodes, no simulation is needed.

3.3 Coordinate nearby lights

This upgrade lets light nodes fetch the state of neighboring light nodes so that they can coordinate traffic flow over multiple intersections.

Model We model this upgrade as adding a new observer to light nodes, getState(). Upgraded light nodes are in-call compatible with old light nodes, but are not out-call compatible with them.

Transform Function The TF adds state to light nodes to track neighboring lights' positions, distances, and states. This upgrade changes the traffic control algorithm to take advantage of this additional information.

The TF uses an oracle (e.g., a discovery service) to identify neighboring lights. If those lights have already upgraded, the TF can call getState() to learn those lights' states and positions. By comparing their position to this light node's position, the TF can calculate the distances to the other lights.

If, however, the neighboring lights have not upgraded, the TF must initialize the state to a default value. Since no single default light state is always correct, the TF should initialize the state to a value that means "unknown." The upgraded light's control algorithm must work correctly if these values are unknown, e.g., by simulating the old control algorithm.

Scheduling Function This upgrade affects all light nodes, and there's no way to schedule the upgrades to avoid out-calls to non-upgraded lights. Instead, we will need to rely on simulation or failure mode to mask the incompatibility. The upgrade schedule must simply avoid causing individual lights to fail; we assume some light node redundancy (e.g. replication) makes this possible.

Failure/Simulation If an upgraded light node calls getState() on a non-upgraded light node, the upgraded node's SO must simulate a response or make this call fail. The SO might be able to simulate the getState() call by returning default state information, but it cannot return a default value for a light's position (since the caller might cache this data). If we assume the light node algorithm can deal with unknown state due to communication failures, then failure mode is a better solution.

4 Web Search Cluster

A large-scale web search engine, like Google, might be laid out as follows: Each document on the web is assigned an identifier. Each machine in a thousand-node cluster is responsible for one one-thousandth of the identifier space. A machine indexes its documents and ranks them. When a user makes a query, a front-end machine broadcasts the query terms to all nodes in the cluster. In parallel, the nodes use their local index to satisfy the query and return the top ten ranked matches. The front-end merges the results and returns the top matches to the client. The entire cluster is replicated a few times on separate networks for robustness and added parallelism.

4.1 Remove common words from index

Indexes on common words, like "and" and "the," are very large (since they match most documents) and mostly useless (since they match most documents). Many search engines simply don't index such words (called "stop words") and short-circuit queries that include them. This upgrade removes indexes on stop words from a search cluster.

Model We can model this system as a set of front-end nodes that call an observer, getDocsFor(queryTerms), on cluster nodes. This upgrade changes the response to this method when some query terms are stop words: before the upgrade, the cluster nodes return the best matches for those words; after the upgrade, the cluster nodes return empty results.

Transform Function The TF for cluster nodes removes indexes for stop words and changes the indexing code to ignore stop words. The TF for front-end nodes changes the code to short-circuit queries for stop words.

Scheduling Function If a non-upgraded front-end makes a query for a stop word on an upgraded cluster node, the cluster node will return an empty response. The front-end will simply return this response to the user without explanation. To avoid this situation, the SF should upgrade all the front-ends in a cluster before upgrading any of the cluster nodes. This is fast since there are few front-ends relative to the number of cluster nodes.

Failure/Simulation Since we upgrade front-ends first and upgraded front-ends are out-call compatible with non-upgraded cluster nodes, no simulation is needed.

References