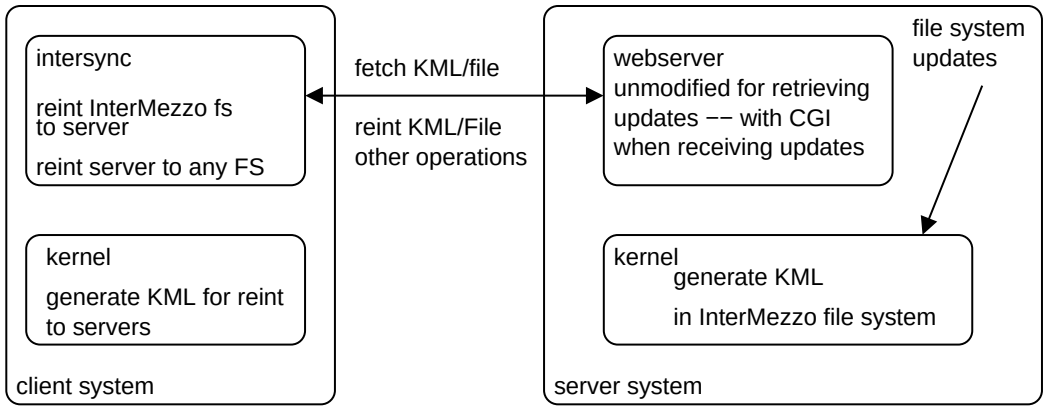
INTERMEZZO: FILE SYNCHRONIZATION WITH INTERSYNC

PETER J. BRAAM <BRAAM@CLUSTERFS.COM>

ABSTRACT. The InterMezzo file system uses is a filtering file system to generate a modification log file which is suitable for replay on other hosts. InterSync is a file synchronization tool that exploits the InterMezzo file system modification log and HTTP servers such as TUX or Apache. InterSync is a scalable client server system to synchronize InterMezzo file systems. It avoids the need to scan file systems and can benefit from proxies and highly performing webservers. Basic versions of InterSync can perform one or two-way synchronization. Fully featured ones will replace InterMezzo's file servers with a different, more scalable system. InterSync has applications in mobile computing, cluster management and high availability server replication.

1. INTRODUCTION

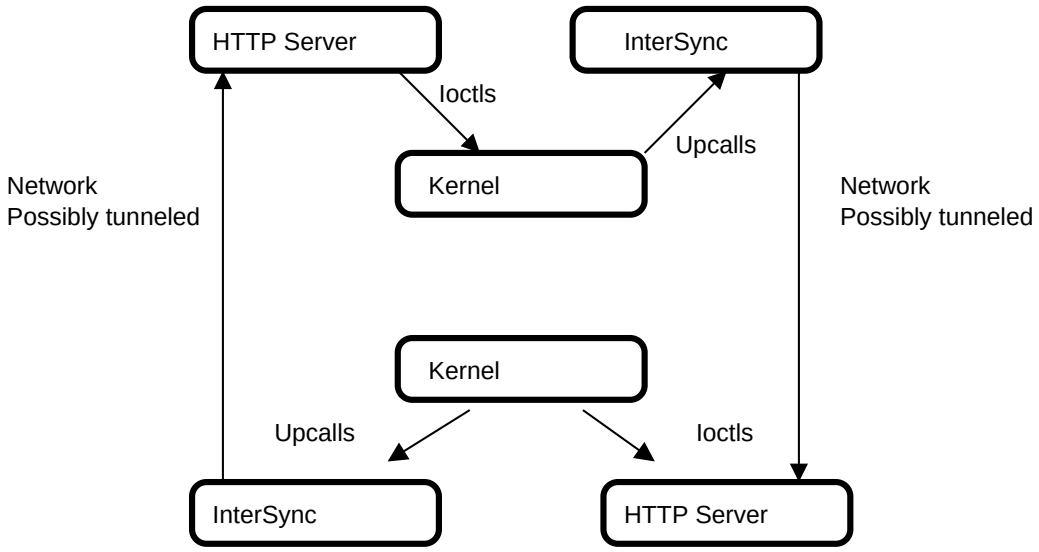
InterSync is a client server system that synchronizes collections of folders (a *fileset*) between a server system and clients. InterSync periodically polls the server for changes and reintegrates those changes into the client file system. The changes are recorded on the server by the InterMezzo file system, which maintains a *kernel modification log (KML)* as it modifies the file system. The modification log makes it possible to collect the changes in the server file system without scanning for differences. InterSync synchronizes the file system by fetching the KML, which is simply a file, using the *http protocol*. InterSync then processes the records in the KML and when it comes across a file modification record, it fetches the file from the server again using the http protocol. As such InterSync is a web based tool, that can benefit from web caches for scalability and from SSL connections for security of the data.



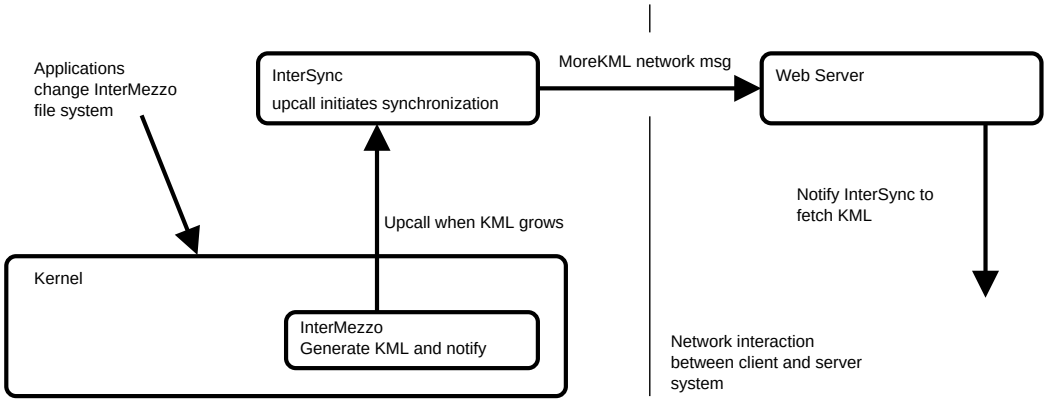
In some cases when synchronizing many clients from a master it is reasonable to assume that the changes are made on the server system and that clients are all the same. Hence it is not necessary to track changes made on any individual replica – all replicas track the master. InterSync can synchronize any file system on the replicas with the master InterMezzo file system. Fully featured InterSync systems run an InterSync and HTTP server on all systems.

Date: Version 0.9.3, Mar 20, 2002.

We would like to thank Phil Schwan <phil@off.net> for extensive discussions as well as the many interested contributors to the InterMezzo project. As with all work on InterMezzo, many of the ideas have been derived from Coda.



The InterSync system can also *actively synchronize* changing InterMezzo file systems with other systems. Active synchronization means that the synchronization is driven by changes made to the file system, and not detected through polling for new KML. Active synchronization requires that an InterSync runs on both systems. InterSync can listen to the kernel through an established *upcall* mechanism. It also requires that the server system can notify the client system through a network connection.

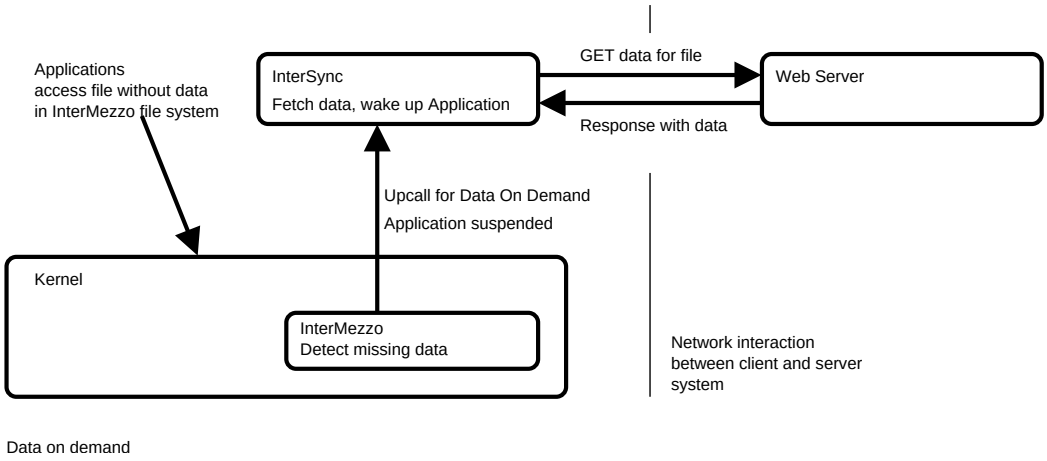


Active Synchronization

If large filesets require synchronization, fetching all the data can introduce latency. For this purpose InterSync can also be told to listen to *data on demand upcalls* issued by InterMezzo. The process trying to access data which is still missing is suspended until InterSync has fetched the data.

If the file system on both sides is an InterMezzo file system, InterSync can be used for *two-way synchronization* – in fact for this there are two distinct mechanisms. One is to stick with a simple client-server system and perform client to server synchronization through pushing the KML and files to the server. This is attractive since it can be performed against any http server, including an SSL enabled server, that has been enhanced with a CGI binary that can handle reintegration of KMLs and upload of file segments. The second approach is to run InterSync on client and server and exploit the symmetry of the situation and allow a server to *switch* into the role of a client.

All this variety begs the question what really is the *difference with the full InterMezzo file server*. As far as the features go, it is possible to have approximately the same features with a fully featured InterSync as with the full system. The differnce is that we use web servers; sometimes this will lead to losing and reestablishing connections. Also, the current InterMezzo server Lento maintains a lot of state for clients to manage synchronization. In the InterSync approach this will be delegated to the kernel code.



Simpler InterSync systems, such as clients using one-way synchronization without concurrency management, do not maintain the full server semantics. InterMezzo servers do not allow the modification of file systems before they have seen the changes from other clients. Secondly, the full server is highly reentrant, while simple InterSync systems are not. While it is processing any kind of request, other requests can be handled. InterSync systems are more serialized, for example they can do data on demand, but for only one file at a time. In case of fully populated replicas, serialization is not a hindrance to performance, since the file system remains fully write back caching enabled.

2. Reintegration and the KML

InterMezzo is a filtering file system layer, which sits in between the virtual file system and a specific file system such as ext3, tmpfs, ReiserFS, JFS, XFS. Below we will pay special attention to the case of JFFS2 which is important for handheld systems. A partition or logical volume which is formatted as an InterMezzo file system is still a valid file system of the type that is being filtered, but intermezzo adds a few files and directories for control purposes. InterMezzo subdivides a partition or logical volume formatted for use with InterMezzo into *filesets*. A fileset is a subtree of the directory tree in the file system, with a designated root directory, which is also called the fileset mountpoint.

The purpose of InterMezzo's kernel infrastructure is to intercept updates of the particular filesets in the file system and in conjunction with these updates:

- generate modification records for updates and append these to the KML
- suspend the calling process it if tries to access data which still needs to be fetched
- maintain record sequence numbers associated with updates

A fileset can also contain mount points for other filesets. InterMezzo merely maintains the contents of a few special files in the file system. These files are for *each fileset* a:

KML (kernel modification log): a log of operations suitable for replay on other systems
RCVD (received records): the file maintaining what records from remote replicas have been reintegrated
SML (synchronization modification log): an optimized KML to sync up an empty replica

Because each fileset has its own KML, clients can synchronize one or more filesets contained in an InterMezzo volume.

2.1. **The KML and reintegration.**

The KML file consists of records, each of which encodes a change to the file system. The records track in detail:

(1) what object(s) was (were) affected by the change
(2) the identity and group membership of the process making the modifications
(3) the version (timestamps and size) of the object that was changed

(4) the new attributes (timestamps, owner, mode etc) of affected objects
(5) a record sequence number

2.1.1. *Changing attributes.* A key aspect in the reintegration algorithm is a solid control over what inode attributes need to be changed with what operation: the reintegration cannot simply set the attributes to the *current time* but must use the KML records to record the times the modifications introduced on the client systems. Most of the KML operations are give an execution pattern on the replica that is quite similar to that found on the system where the records originate through system calls. However, *close* records, generating after a modified file is closed, are an exception.

A UNIX system modifies file time stamps and sizes after each file write, but not upon close. The *close KML record* must provide the latest time stamps and sizes that are present on the file. In this way the omission of the write records from the replication sequence in the KML does not affect the attributes of the file. If data is fetched into a file upon demand, for example through fetching data at file-open time or during read for streaming purposes, then the attributes of the inode should not be changed.

The following table provides insight in what attributes are set changed by local file systems and what attributes are changed by intermezzo KML records:

op	fs	atime	mtime	ctime	pmtime	pctime	patime	uid	gid	mode	size
create	xfs										
	ext2										
	intermezzo										
mkdir	xfs										
	ext2										
	intermezzo										
close	xfs										
	ext2	no	no	no	no	no	no	no	no	no	no
	intermezzo	no	yes	yes	no	no	no	no	no	no	yes
write	xfs										
	ext2	no	yes	yes	no	no	no	no	no	no	yes
	intermezzo	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

2.1.2. *Authorization.* All KML records should reintegrate in a certain user context: before the kernel reintegrates a record we set the fsuid, fsgid and group membership to that of the process that made the changes in the first place. For every record, except close, this context is found in the KML record, and copied into the record from the calling process' task structure at the time of creation of the record, but for close things are different.

Wether a file can be written to is determined at open time, nothing changes that permission afterwards. InterMezzo writes files on the server later when the close record reaches the server, so we need to keep the mode, uid, gid of the file *and* the fsuid, fsgid and group membership of the process at open time around until the close record is written, which incorporates this information.

2.1.3. *Further details.* Typically a KML record is between 100-300 bytes in size, depending on the operation being performed and the length of the pathnames. Once the KML has been transferred from one system to another the process of *reintegration* can begin. Reintegration goes through a few steps:

(1) unpack the records in the KML segment

(2) check if the versions of the objects that are being modified match those given in the record. (If not this is an indication of a possible conflict.)
(3) make the change to the file system.
(4) if this is a store record, fetch the file.
(5) proceed with the next record

InterMezzo, like AFS and Coda, synchronizes files when they are closed after writes. More frequent synchronization is possible through the kernel based I/O-daemon (implemented by Tacit Networks), which can force synchronization records when files remain open for a long time. The daemon can also limit the synchronization frequency when files are frequently opened and closed. This is particularly important for running a file server such as NFSD over InterMezzo as NFSD opens and closes a file for every write.

Often files can be modified several times in a short while and temporary files are removed. InterSync will detect during reintegration that a temporary file was removed and it avoids fetching the same file multiple times. Further *KML optimizations* exist (such as not creating directories which are removed soon afterwards) but these optimizations are of much smaller impact on the performance. There is some subtlety involved with backfetching files, as files may have been removed or renamed. For this InterMezzo has a mechanism of opening files by file handles that are immune to renames. Also permissions and group memberships need to managed carefully. InterMezzo uses the same model as NFS, namely that client's group membership determines the handling on the server.

InterSync can also handle a push based reintegration which sends the KML to a peer first. The peer reintegrates all records but does not proceed to backfetch files. Instead the client follows this will file uploads for all records in the KML that affected file contents. This system is entirely client driven and does not require the client to be capable of responding to RPC's from a peer.

2.2. **Tracking progress.** File systems managed by intermezzo have a UUID associated with them for each client. While intermezzo is reintegrating KML from a client with a particular UUID it atomically:

- updates the file system
- writes KML records when required
- increases the last received record numbers and offset associated with the UUID.

The received numbers are managed per fileset in a RCVD file. The RCVD file contains records labelled by UUID and contains several integers describing the offsets and record numbers of records that were reintegrated.

The atomic aspects of this are discussed in the Recovery section below and form the pivot for recovering the file system after crashes.

When intermezzo systems do never propagate updates they receive to other systems, no KML records need to be generated, but the changes to the file system and the received status is still important. This situation happens on client systems which can trust server data integrity.

In high availability situations, clients typically will keep KML records so that replay to a recovering partner offers maximum chances of recovery and maximum information in case conflicts need to be resolved.

When changes are reintegrated to branches it can happen that last rcvd and KML records are updates, while the file system itself is not updated. The reason for this is that the branches are represented by operation logs and not by cache state.

2.3. **KML truncation and Initial synchronization.** InterMezzo maintains a secondary, optimized, replication log called the *SML - synchronization modification log.* The contents of the SML consists purely of creation of objects and is an acceptable concise replay history for a fileset. The SML grows with the size of file set but does not have possibly unbounded growth as it does not contain create/unlink and rename operations which lead to growth. It serves the purpose of allowing an empty or heavily out of date client to sync up in an efficient manner. For new clients this synchronization is optimal, for heavily out of date clients it may not be as optimal as KML would have been.

A small C program builds the SML by first noting what the current KML offset is. Any KML created during the build of the SML must be replayed on the client as its actions may not have been included in the SML. When the SML creation is complete, the state of the KML is noted again. Only the segment of KML generated by the kernel during the SML creation is subject to possible overlapping information with that contained in the SML. There are two details to consider here. First, renames introduced while the SML is being generated could lead to the SML being incomplete. We turn off renames by asking the file system to return -EXDEV during SML creation. If sufficient information is added to rename records, rename operations on leaf nodes could remain in the KML. This still leaves the possibility for conflicts between the KML tail and SML. The conflicts arise from deletions already implemented in the SML and from creations in the SML which are repeated in the KML tail - these are not hard to handle. When the SML creation is complete, the KML needs a marker and renames can be re-enabled.

A newly-connected client should do a *replicator status* call to find out the status of the KML. If it determines that it is older than the last KML truncation, i.e the clients last rcvd is less than the current KML logical offset, it first fetches the SML. The client reintegrates the SML as usual, but with a much higher tolerance for events that would otherwise be considered conflicts: for example, being told to create a file that already exists. If a client remains connected but the KML is truncated, the client will notice that the records it is expecting are not in the expected place and it can place another replicator status call.

Following the reintegration of the entire SML, the client must remove any files present in its local cache which were not referenced in the SML or updated on the client - they are no longer on the server. The removal of files concerns files deleted after the client was last connected, but before the SML was written. Care must be taken to avoid removing files which were modified locally; because these files no longer exist on the server, they are in conflict. For this the server can tell the client the timestamp of the last client record it reintegrated. Objects newer than this should be sent to the server.

Once the entire SML is reintegrated, the client fetches the next part of the KML. It integrates this with similarly relaxed conflict checking until it encounters the first record following the SML creation. Once this segment of KML is reintegrated, the client is once again up to date and resumes activity as normal.

The client can detect any mismatch between record sets in the KML it receives and it expects. However, during truncation proper - the moment the tail of the KML is renamed to the new KML and the old KML is unlinked - the KML must not be open by intersync initiated GET requests. The kernel will synchronize this access.

2.4. **File Data.** File data can be fetched at three particular instances. We intend that this will be kernel driven.

replication mode: File data is fetched fully during reintegration as part of the execution of the reintegration of close records. The mechanisms for fetching could be HTTP GET or rsync commands

data on open: File data is fetched fully when files are opened and the kernel detects not all data is present. This can be done using GET or rsync methods

streaming: File data is fetched in chunks when files are read and file data is missing

These mechanisms are suitable in different scenarios. Servers will on the whole build full replicas of client's caches.

File data can also be purged on clients by a utility that scans a database for not-recently used files and purges data from these files accordingly.

We foresee that in the future we will also fetch metadata on demand.

2.5. **Conflicts and conflict resolution.** Conflicts can arise when updates are made on two systems. In many cases updates on two systems do not affect the same objects in the file system hence do not pose conflicts.

The first check to determine if conflicts are absent is to check if perhaps only one of the systems made changes since the last reintegration. If this is so, the system can blindly reintegrate all records. InterSync makes this check. Before reintegration starts, InterSync (as does the full InterMezzo server) determines the state of the peer system, by finding out what it last received. The last received records are known to be exact, since they are transactionally updated as the file system moves forward. If the peer indicates that it has no more records

than were already received we know that no changes were made there. So InterMezzo knows for certain when conflicts may have happened.

The second check on conflicts is to check for versions. While Coda maintains exact versions inside the file system (at the expense of significant complexity having its own management for all metadata) InterMezzo can leverage the timestamps and sizes of files. In most cases this is sufficiently accurate, but it is possible that objects retain the same timestamp and size while they are modified. The idea behind checking the version is that if the versions of the affected objects match, one assumes that the update can be applied.

If both of these checks fails, we are dealing with a possibly fake conflict. In dealing with conflicts it appears tempting to take a *value based* approach, i.e. perform a semantic check for conflicts. The second approach is *operation based*. InterMezzo and InterSync at present take the second approach.

Directory and metadata conflicts were classified in the Coda and Ficus projects into a small category. Only four obvious types of conflicts need manual intervention:

Name/Name conflicts: objects with the same name are created on multiple systems.
Update/Remove conflicts: one system removed an object while another modified it
Update/Update conflicts: multiple systems update the same object
Rename/Rename conflicts: an object is renamed by multiple systems to different destinations

File conflicts are something we regard as an update/update conflict: this happens when file data is updated in multiple locations and require corrective action.

Conflict resolution. A major difference between Coda and InterMezzo is that InterMezzo resolves conflicts according to a policy without user intervention. Use intervention follows the synchronization actions taken by the system independently. There are several policies we will use.

Mobile policy: to synchronize between mobile devices and servers. We keep the conflicts on the mobile device and let the mobile object move out of the way when conflicting server objects are introduced. The mobile device is always the client node.

HA policy: for synchronization between high availability *fail-over* servers. When reconnections happen there is a *failed node* re-joining the *active node*. The conflict resolution policy here is to let the active node prevail and we move conflicting objects on the re-joining failed node out of the way. If the failed node is the client this policy coincides with the mobile policy.

Re-synchronization policy: used when re-synchronizing systems when available KML is not sufficient to perform the synchronization or after KML has been truncated. We discuss this in a separate section, but it is substantially similar to the subject matter here.

To describe the algorithm in detail and understand its correctness the following algebraic description of the problem is useful to keep in mind. The client and server were identical at some point at which they had a directory tree D. On the client the tree D was transformed into D1 by applying KML1. On the server the tree was transformed into tree D2, the changes are described by the server log KML2. The client has knowledge of D1 and the two modification logs KML1 and KML2. The KMLs are sequences of operations, e.g.

KML1 = O1[N]...O1[1]

Furthermore, the problem at hand is really one of *commutation* of operations: how does KML2 KML1, which is the logical sequence on system 1, compare with KML1 KML2, the sequence on system2. From elementary group theory we see that computing *conjugates* of the transformations is the relevant operation to perform. The conjugate of a by b is $c = bab^{-1}$ which solves the commutation problem cb = ba. Our operations are transformations of the file tree, and the conjugation of transformations is done by applying the operation on their arguments and values. In our case this means that switching the order of rename operations with other operations involves applying the renames to the pathnames in these other operations.

We will assume that system1 is processing the KML2 it has fetched from system2, but system2 has not yet seen KML1. We will change KML1 and the tree D1 into a local tree D1(L) and KML1(L) and supply a remote

KML1(R). Similarly the records in KML2 are changed to become KML2(R) records. The aim is that reintegration results in:

(1) KML1(R) KML2 on system2 equals KML2(L) KML1(L) on system 1
(2) There are no conflicting operations between KML1(R) and KML2 and between KML2(L) KML1(L).

There are multiple solutions to this problem and the solutions we choose reflect the policies introduced above. We describe this for the mobile policy. The task of obtaining this comes down to:

(1) move conflicts out_of the way: make local (l) changes to D1 = KML1(D) and obtain D1(l) = KML1(l)(D). This involves eliminating records from KML1, changing the following records in KML1 accordingly, and applying a suitable operation to D1 which corresponds to the removal of the conflicting object. The corresponding remote operations for such KML1 records to be applied on system 2 become noops.
(2) apply KML1 renames to KML2: This is relevant if a KML1 rename record affects pathnames in a KML2 record. The paths of the KML2 record need to be renamed accordingly. This leads to KML2(l) records, the records of KML2 modified for local reintegration.
(3) apply KML2 renames to KML1: This is the opposite of step 2 and leads to remote KML1(r) records, suitable for reintegration on system2 in the tree D2 = KML2(D).

The algorithm implementing the *mobile conflict policy* proceeds by taking the last record of KML2 and pulling it across the operations in KML1. This is repeated for each record in KML2 until the first record is reached. Whenever a record O2 of KML2 is pulled across a record O1 of KML1 it results in

(1) in case of no-conflicts new records O2[L] and O1[R] compensating for renames
(2) in case of conflicts O1(R) and O(L) become noops, O2(L) == O2 and
    (a) modifications are made to the local tree D1 to undo O1
    (b) the remainder of KML1 is changed to adjust for the removal of O1

This transformation achieves:

no conflicts: O1(r)O2 == O2(l)O1
conflicts: O2 will apply locally without conflicts to the modified tree.

By repeating this transformation across each record in KML1 for each record in KML2, we finally obtain the corresponding *local and remote* KML1(L), KML1(R), the local tree D1(L) and the local KML2(L) such that:

(1) D1(L) = KML1(L) D
(2) KML2(L) applies cleanly to D1(L)
(3) KML1(R) applies cleanly to D2
(4) KML2(L) D1(L) == KML1(R) D2

The conflict resolution now finishes by locally reintegrating KML2(L), and then reintegrating KML1(R) on system2. Note that KML1(L) is not used for reintegration, but represents the computational state when adjusting records in KML1 for conflicts. KML1(L) is being used to compute the next KML1(R) records in the procedure. Also note that a property of this transformation is that no conflicts are introduced among records of KML2(L).

We now describe the process of transforming the operations O1, O2 and changing the local tree D1 and KML1 in more detail.

(1) Determine if O1 and O2 conflict. This is a somewhat elaborate function checking all pairs of operations for all possible conflicts.
(2) If no conflicts exist O1(l) = O1:
    (a) Determine if O1 and O2 commute. If so, O1(r) = O1, O2(l) = O2. Non conflicting operations commute unless one affects the pathnames of the other, i.e. one or both are a *rename operation* affecting an ancestor in the path(s) of the other.
    (b) O1 and O2 do not commute compute the effect of a rename O2 on the paths of O1 to get O1(R), and that of a rename O1 on the paths of O2 to get O2(L). Now O2(L)O1 == O1(R) O2, as desired.
(3) If conflicts exist, the *mobile* policy says that we will move conflicts out of the way.

(a) If O1 and O2 are a name/name conflict. The basic operation is to eliminate operation O1 from KML1, and instead perform O1 in the conflicts area for the fileset. So O1(R,L) become noops, and O2(L) = O2 is unaffected. Note that:
   (i) this can have a chain effect on other operations in KML1. For example in the case of a mkdir in KML1 all children also need to be undone, or moved out of the tree.
   (ii) requires scanning the remainer of KML1 to know how to undo O1 on D1. For example if mkdir foo conflicts with O2 but is then renamed, it is in fact the renamed object that needs to be moved out of the way.
   (iii) An important refinement is that we could first scan the KML1 to verify if the conflict was undone by a later operation in KML1, for example, if both O1 and O2 are mkdir foo, but KML1 renames foo to foo1, we can change KML1, including O1, by replacing the pathname foo with a temporary different pathname which does not conflict with O2. Similarly, conflicting objects which were created and then removed will not cause real conflicts.

(b) update/update conflicts are handled in a similar fashion. We move the updated object to the conflict area subject to path renames in the remainder of KML1. We apply the update there and eliminate O1. Note that
   (i) close records are considered updates.
   (ii) before O2 can be applied, it may be necessary to refetch the data from system2.

(c) update/delete conflicts are handled as follows.
   (i) If O1 is a delete and O2 an update, a *note* is written in the conflict area to indicate that O1 removed an object modified by O2. The removal of the deletion O1 from KML1 means the O2 object has to be re-created as well as updated. This can introduce subsequent name/name conflicts between the remainder of KML1 and O2. A further complication is that records following O1 in KML1 may have deleted further ancestors of the object in O1. All of these need to be re-created and their last valid attributes (owner, mode, times) need to be retrieved from the unlink/rmdir objects. The fix up of ancestors eliminates further records from KML1.
   (ii) If O2 is a delete and O1 an update, we move the modified object to the conflict area, subject to renames in KML1 affecting the pathname in O1. By induction we know that the object in O2 is a leaf object in the directory tree, so children are not present. This inserts the deletion O2. So O2(L) = O2, O1(R,L) become noops.

(d) rename/rename conflicts are handled by eliminating the rename O1 and replacing it with rename O2. This can affect subsequent paths in KML1.

Further considerations. It is clear that it is undesirable that the KML1 or KML2 grows further during conflict resolution. The client is should revoke permits from the server and from its kernel while completing the resolution. It is also clear that once conflict resolution is initiated it is undesirable for either node to lose records due to system crashes, an event which InterMezzo normally handles gracefully. Hence a sync call is appropriate on both nodes before starting the operation.

If the process is interrupted (system crash, network down) it should be possible to resume it, and for this one needs to carefully consider the changes that are made to the tree D1. These changes occur (a) when a conflicting object is removed, which is associated with a pair of records (O1, O2) and it is probably wise to sync the record numbers of (O1, O2) to disk to know where things were left off. Secondly these changes occur when records from KML2(l) are implemented. This is a standard reintegration operation for which recovery is handled. In connection with this, it is a good idea to track very precisely what remains to be done after recovery from a crash on the client node. The last received offset for KML2 is one factor in this, the other his how far the conflict removals associcated with a record O2 have proceeded.

To simplify the work and gain efficiency it is useful *to optimize the KML* first in the following way (1) move all rename operations to the beginning (the commutations affect pathnames in other records) (2) eliminate creation/removal pairs behind the renames. There is one subtlety associated with the reorderings. Moving records across changes in permissions caused by setattr routines can cause failures if the reintegrations are not performed with root-like permissions.

The code implements handlers for each conflict case, using the following basic operations.

(1) move subtree out of the way
(2) write warning about attributes
(3) generate attributes for ancestors by scanning client KML
(4) instantiate ancestors (and adjust attributes of ancestors)
(5) mark certain records in the client's KML as noops.
(6) rewrite pathnames in KML records to compute commutations with rename operations

Checking ultimately that remaining client KML1(r) does not conflict with the server KML2 segment as received is a useful check. The check is not necessary, but the algorithm is complex enough for this to pay off. In order for conflict handling to be implemented KML records will contain somewhat more information about the old objects than they do in prior versions of InterMezzo - namely the old attributes of unlink and rename operations must be preserved.

We will have a hash table that maps a pathname, and possibly ancestor pathnames, to a linked list of KML record offsets in the client KML in this tail segment. (And something that cleans this up.) We will assume there is enough memory for this. This hash will allow the retrieval of old attributes of ancestors for example.

The original KML records can be obtained simply by memory mapping file segments. Applying unpack routines which merely cast offsets in the file segments as KML records or strings used in the records. We form a doubly linked list of records. When the records are modified a private (copy on write) mapping is useful, so that modifications can be done in-place. It is helpful to give the records extra char * fields that allow us to replace the strings that were memory mapped with transformed strings that are external to the memory mapped region (and need to be freed). While re-packing the doubly linked list into the new local or remote KML, strings are copied out from either location, and inserted records can easily be included.

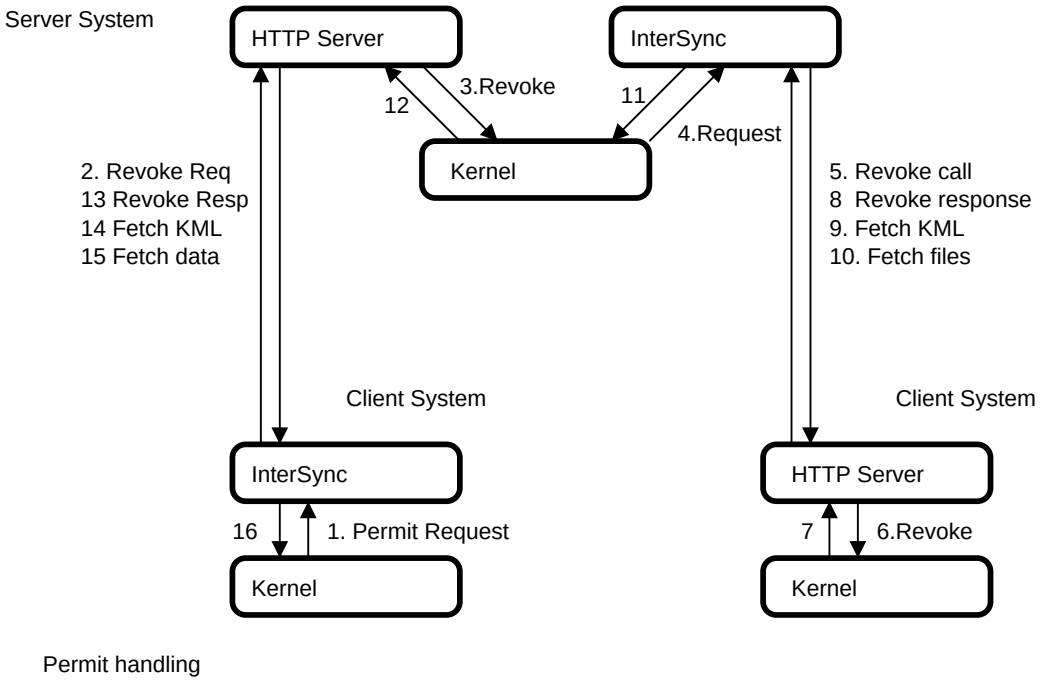
2.6. **Permit handling.** InterMezzo has so far typically allowed single system to update file sets, through a permit. One purpose of InterSync is to have more of a laissez faire approach and allow reintegrations to happen without permits. The expense is that conflicts have to be fixed when they are encountered.

While this limits update concurrency on multiple systems it can lead to the avoidance of conflicts. InterSync implements client-server functionality with permit support. It might be attractive to write permit information to files so that state can be preserved between runs.

InterSync Permit handling is different from the current implementation in Lento. The kernel would detect that a client system wishes to make an update. InterSync notifies the server and asks the server to revoke the permit. When the server has completed the revocation of the permit from another node it grants the permit to the InterSync on the client. Before InterSync grants the permit to its kernel it fetches the KML and reintegrates this. The figure below shows the interaction between systems when permits are requested.

A further complication arises when multiple systems try to get the permit simultaneously. Here InterMezzo's kernel implementation enforces a *progress rule*: systems will not surrender a permit unless all file system operations queued before revocation requests come in have been served.

Finally we speak of *permit theft* when the permit is granted while the current permit holder cannot be reached. In this case it is important that systems retain on persistent storage at what KML offsets this event took place. In this way, a *common ancestor* of the state between the thief and the unreachable system is well determined.



2.7. **High availability clusters and rollback.** In a high availability cluster with failover services it is possible for a system to leave the cluster before its peers manage to synchronize the contents of the nodes. By using a permit system, peers can be aware that they are stealing the permit from a node that is no longer reachable. When this node comes back, it may be that it finds its KML in conflict with KML found on other servers. One way to deal with this is to perform *rollback:* rollback means that a node undoes changes by analyzing KML records and using data from other servers. KML has to be undone in the reverse order from which is was created, and if files were modified, data needs to be fetched from peers.

2.8. **KML for the JFFS2 file system.** The JFFS2 file system is becoming the default flash file system for Linux handheld devices. As this filesystem is a journaling file system, it is possible to build much of the modification log on the fly, through a few minor changes to the file system which we propose here.

(1) The file system, while not garbage collecting freed space can insert a transaction number in to each updated node it writes to flash. Likely the operation performed can be inferred from the node information - if not, a field may have to indicate the operation.
(2) While the filesystem is being built, an ordered list (or tree) of transactions is constructed.
(3) During node garbage collection, garbage collected transactions, their names, inode and parent inode numbers are collected in a KML garbage file. This file can be used to reconstruct the deletion entries for a full modification log from a very compact representation of deleted items.

With a few ioctls a KML could be generated from this information on the fly. This mechanism introduces two new node types into JFFS2: new directory node and new inode types which hold the list and transaction id information.

2.9. **Branches and snapshots.** InterMezzo can be used to maintain branches and snapshot views of file systems. For a general discussion of branches and snapshots see [versionfs]. Here we give an overview of the principal mechanisms used in branching and snapshotting.

The first key event for snapshot and branch file systems is the freezing event which happens when a certain branch forks or when a file system needs to be retained as a snapshot. On the InterMezzo server we separate the KML of the frozen system from that of the branch or live snapshot which continues to evolve. Also file data that is moved to the live branches or live snapshot needs to be stored in a separate area of the file system.

The client sees relatively few changes in its interactions with the server. To sync up a live snapshot the KML's of the frozen snapshots that preceed it are fetched and reintegrated. When file data records are encountere

3. RECOVERY

Linux Ext2 file systems have to be checked upon startup if the system was not shut down cleanly. A simple example that necessitates such file system checks and recovery arises as follows: when a new file is created, several modifications are made to the disk structures. The file has a name and this name needs to be added to the folder containing the new file. But the file also has an inode, which needs to be allocated. If the system were to crash between two of these allocations, say after the name had been written into the directory structure, before the new inode was allocated, then the file system is in an inconsistent state.

Fsck repairs such inconsistencies, by doing a comprehensive scan across all object stored on the disk and performs checks for every possible inconsistency. Ultimately this imposes serious limitations on scalability of the file system through excessive recovery times.

Journaled file systems implement a completely different solution to this problem. A journaled file system will group a collection of related changes into a intransactionls. All the data modified in the transaction is first copied to a journal file. Only after it is on stable storage will the data be written to its final location. When recovering from system crashes, copying the journaled transactions from the journal to their final location will restore the file system integrity. This process does not involve scanning the file system, nor does it involve analyzing all possible forms of inconsistencies that might occur. Instead, the system simply copies the transactions from the journal to their final locations. However, journal systems are far from simple. One factor that contributes to their complexity is that buffers backing file systems may be involved in multiple transactions simultaneously and corresponding versions of buffers must be present in the kernel. The ordering constraints imposed by the journaling mechanisms can lead to situations where it is complicated for the kernel to free memory. These problems have yet to find a satisfactory solution in the kernel.

Early on in the InterMezzo design process we decided to leverage journaled file systems as the cache. Several of these systems are now in an advanced stage of development, notably Ext3, ReiserFS, XFS (contributed by SGI) and JFS (contributed by IBM). InterMezzo can use any of these, provided they offer nested journal transactions and journaled file writes. Nested transactions allow InterMezzo to initiate a transaction that encompasses changes made to the cache by the journal file system and to data managed by InterMezzo. Journaled file writes are needed to update the operation log, which is simply a disk file, transactionally with the cache. When InterMezzo commits the nested transaction, either the cache and the InterMezzo data structures are updated, or neither of them is. The transaction is called nested because the transaction made by the journal file system is isnestedlo in the middle of the compound InterMezzo transaction. Fortunately nested transactions are needed for other purposes such as implementing quota. InterMezzo significantly increases the number of possible forms of inconsistencies upon crashes. We will discuss the two most obvious cases, which involve the journal and the replicator structures.

When the InterMezzo file system cache is modified by an application, the corresponding operations are logged in the InterMezzo operation log. The first case of inconsistencies after a crash is when the cache modifications do not accurately reflect the contents of the InterMezzo operation log. The way this can be avoided is by building a nested transaction: InterMezzo starts a journal transaction, and first asks the file system to perform it"s modification as the inner part of the transaction, then the operation is written by the InterMezzo kernel module to the operation log as part of the same transaction. Now the log and the cache will remain consistent.

A similar problem arises when reintegration records are received from another system. The cache manager needs to apply the records to the cache and increment the last_received record counters in the replicator structure. If the counters are not incremented within a transaction involving the updates to the cache, one can again get complicated inconsistencies. We will be handling this problem again by treating the file system operations performed by the cache manager as a nested transaction. The inner transaction is that of the modification of the cache by the journaled file system and the outer one is InterMezzo initiating the cache updates and updating the replicator database. These are not the only two problems: a more complicated recovery problem involves that of file data.

A substantially more delicate issue concerns modification of file data. InterMezzo 1.0.0 synchronizes file data after a file is closed. Although writes to a file must happen in order to modify the file(footnote: excluding the case of memory mapped files) a close will not be processed by the file system if the system crashes. After recovery from a crash the system needs to have a way to synchronize the data that was modified. In order to accomplish this, InterMezzo maintains a Local Modification Log that has records of files that have been modified. Such records are created with the first write to hit a file. Upon recovery, InterMezzo replaces the LML records by genuine KML close records which will be reintegrated, thereby avoiding lost updates and maintaining scalability.

4. INTERSYNC INTERNALS

In this section we will discuss InterSync. If the purpose of InterSync were merely to synchronize a server data collection to a client, a simple InterSync client mode program would suffice. However, intersync treats the replicas as filesystems and allows for two-way updates. It connects to an http server, fetches KML and during reintegration can fetch file data too. The next level of complexity would is introduced by data on demand is added. Here the InterSync must linger and listen to the kernel for data requests when files are accessed.

The environment gets more complicated when the system does kernel driven synchronization or two way synchronization. Now we need InterSync daemons on all systems, and these can be spawned for a single synchronization run or for long lasting sessions.

The current implementation implements all of these features.

4.1. InterSync execution environment.

InterSync is meant to act as a *client server* system or as a *run-once* system. Depending on parameters the system will perform the following steps:

HTTP server startup: When InterSync is started it always starts the http server on the same node.

Setup tunnels: If InterSync is to run over a tunnel ssh will set these up. The tunnel mechanism simply tells intersync on what port it can reach the http server on a client. We have two types of tunnels: *high availability tunnels* which attempt to re-establish themselves if connections are lost. *Run once* tunnels exit and tear down remotely instantiated servers when they fail. We will not discuss high availability tunnels in this paper. Run once tunnels can very flexibly be constructed with *OpenSSH*.

Instantiate remote: As part of setting up a tunnel ssh can start an remote intersync daemon (which starts its associated http daemon) on the remote side. This is useful for run-once synchronization.

The actions described above can be implemented by simple shell wrappers of the true intersync binaries. When forking off remote web servers and intersync clients, configuration information for these agents has to be supplied but is minimal.

4.2. InterSync and server states.

The InterSync program performs operations in two styles:

connecting: InterSync tries to establish a connection with a server or accept a client. After a connection failure, one attempt to reestablish the connection is made without delay when a request requires a connection. After that the system operates in disconnected mode for a certain time interval, after which a reconnection attempt will be made. Reconnection attempts are always made when permits were stolen or when KML is lingering. If neither are the case, reconnections are made on kernel demand.

client context execution: InterSync will go through execution contexts to complete data on demand, reintegration and permit acquisition requests. No data is pushed to servers: all data, KML, file data and configuration information is fetched.

We believe that the InterSync client is largely stateless (the kernel carries the state) and therefore can be run as a multithreaded program.

The http server services GET requests for KML, file data segments and configuration information and it must also handle several control calls.

GET_mode: the system serves KML and file data to a client through httpd. The client and server make judicious use of *persistent connections* and *http pipelining* of requests.

CGI_mode: the server runs CGI scripts to handle *moreKML, revokePermit* and *reintegrateKML* requests. A CGI method to confirm reintegration is desirable as it allows the server to dispose of KML when all clients have fetched it.

An important part of the request handling is *intersync ping pong notification*, which switches the system transitions from server to client mode or vice versa. Notification can be done through opening a special file (while the server is in GET mode) or by issuing an *ioctl* from a CGI script. The CGI script or open system call can be blocked by the kernel to complete interactions with InterSync through upcalls.

4.3. **Fundamental interfaces.** On each system running intermezzo there are three major components: the http server, intersync and the intermezzo kernel based file system. The roles of these systems are as follows:

intersync: This subsystem has several interfaces which it uses or offers

uses:: httpd GET and CGI reqeuest interface offered by http daemon intersync makes http requests to the http daemons on other systems. The usage model is a simple request response model. No network requests are answered by intersync

implements:: upcall interface used by kernel intersync listens for upcalls from the kernel code. Such upcalls are the driving force for activity in intersync. Some upcalls are processed asynchronously, i.e. they do not require a response to the kernel, others are synchronous, i.e. the hold processes waiting until they complete and are answered. Upcalls are made by sending messages through a channel associated with an open file handle on a character device instantiated by intermezzo in the kernel. The standard poll/select mechanism wakes intersync upon arrival of an upcall. Some upcalls are *ping-pong upcalls:* such upcalls are generated as a result of the http server making an ioctl to intersync.

uses: ioctl interface implemented by the kernel ioctls are made to the kernel to notify the kernel of events. Some events given to the kernel are very complex, for example file system changes retrieved from other systems are reintegrated locally through ioctls. Ioctl's made by intersync do not cause upcalls or contact other systems directly.

http server: This server has two interfaces:

implements:: GET/CGI interface used by intersync: this interface provides the channel for data operations which fetch files and metadata updates from the server to the client systems and vice versa. The associated control commands are implemented by CGI methods.

uses: ioctl interface offered by kernel:: The httpd server uses two styles of ioctls to the kernel: some ioctls merely retrieve state form the kernel which is included in the reply to intersync. Other ioctls *ping-pong* to intersync, i.e. they cause an upcall in intersync, causing it to contact other systems on behalf of the http server. Ping-pong ioctls can be synchronous, i.e. they await the return of a synchronous upcall, or asynchronous, i.e. they deliver a message to intersync that requires no reply to the kernel.

intermezzo kernel_file system: This system offers two api's and uses one:

offers: file system API to the VFS: allowing applications to make updates to and retrieve information from the file system. Sometimes such usage can only complete after upcalls to to intersync complete.

uses: upcall api offered by intersync: This API is the primary API to get network action including data retrieval and coordination between systems.

offers: ioctls to http & intersync:

The remaining parts of this section will outline these api's as well as the state they manipulate.

4.3.1. *Upcall interface used by kernel offered by intersync.*

FS driven upcalls::

upc_kml::

(1) ells intersync more KML is available,
(2) propagated with cgi_go_fetch_kml http network method invocation
(3) invokes ping-pong ioctl on http server to fetch KML

(4) results in remote fetching KML through get

upc_permit::

(1) required when applications want to modify the intermezzo FS
(2) if client has permit upcall returns immediately
(3) otherwise network request job_getpermit does revoke_permit_ioctl on the server (through cg_permit)
(4) - should run job_KML with reintegration (doesn't do this)
(5) - send reply to kernel permit upcall.

upc_getfileid::

(1) new one, goes to server to get the ino and

upc_backfetch::

(1) run job_backfetch gets file data from a UUID system

upc_open::

(1) used to do a backfetch by name from a UUID that wass passed in.
(2) Will be depricated in favor of a combination of getfileid and backfetch.

Ping pong upcalls::

upc_connect::

(1) other node is connecting to us
(2) arguments: ipaddress and port and uuid of peer.
(3) description: causes an intersync to connect to a peer. This is necessary to establish a connection or session between two systems.

upc_go_fetch_kml::

(1) results from go_fetch_kml_ioctl,
(2) arguments: filesetname, kmlsize, uuid of peer
(3) effect: intersync fetches KML from the server in a *job_kml*
(4) description: this upcall arises from a cgi_go_fetch_kml invocation which is a notification to a server that more KML is available on a peer.

upc_revoke permit::

(1) argument: fileset, uuid of permit holder.
(2) effect: on a server does a job_get_permit from other replicas, on client upcall tells intersync who will get the permit (not necessary?)
(3) should be unified with permit upcall as the results are effectively the same.

4.3.2. *http interfaces to the server*.

ping pong initiating::

cgi_go_fetch_kml::

(1) arguments: fileset, uuid of caller, caller KML size
(2) causes: go_fetch_kml ioctl
(3) returns: nothing,
(4) is asynchronous
(5) description: this notifies other systems that more KML has been generated and can be fetched. The arguments describe how much KML has been generated, where and for what fileset. After the pingpong the intersync returns to the sender of this CGI request to fetch KML.

cgi_permit::

(1) arguments: filesetname uuid of caller
(2) effect: causes revoke_permit ioctl on server.
(3) returns 0,1
(4) description: this CGI is used to request the permit to come to a system. It may involve the server revoking the permit itself from other systems.

cgi_uuid::

(1) optional argument: client port
(2) effect: ioc_connect

(3) returns: the server's UUID on success
(4) description: this command initiates a session between a client and server. It causes the server to request intersync to establish a reverse connection.

non-pingpong cgi_requests::

cgi_repstatus::
(1) arguments: filesetname uuid of caller
(2) effect: get_rcvd and get_kmlsize ioctls
(3) returns: repstatus to caller

cgi_set_kmlsize::
(1) argument: fileset, uuid of caller, caller KML size
(2) cause: ioc_set_kmlsize
(3) sets: remote last offset
(4) returns: nothing to caller
(5) description: this call is made by a client when it detects that the server is not aware that it rolled back or truncated KML (e.g. in a intermezzo/tmpfs file system after a remount, or after KML truncation).

data fetch_requests::

file data:: fetches file segments
KML_data:: fetches KML data
profile:: fetches profile file (this will become a CGI).

4.3.3. *Ioctl interfaces to the kernel.*

intersync ioctls::

ioc_get_channel::
(1) made on: a directory which is an intermezzo mount point
(2) returns: on success the channel for upcalls associated with mount point at mount time

ioc_set_channel::
(1) made on: intermezzo mount point directory
(2) argument: an /dev/intermezzo file descriptor
(3) effect: sets the channel number for the /dev/intermezzo file descriptor
(4) returns: success/failure

ioc_set_ioctl_uid::
(1) made on: intermezzo mount point directory
(2) argument: uid of user "intermezzo"
(3) caller: must be root
(4) effect: tells the kernel the uid of the priviliged intermezzo user which has full privileges in intermezzo file systems, no privileges anywhere else.
(5) returns: success/failure

ioc_set_pid::
(1) made on: intermezzo mount point directory
(2) argument: process id of intersync

ioc_clear_fset:: deprecated in favour of kernel only handling
ioc_clear_all_fsets:: deprecated in favor of kernel only handlling

ioc_reint_kml::
(1) made on: file set root directory
(2) argument:
  (a) a KML buffer fetched from a remote
  (b) offsets?
  (c) uuid of originator of KML
(3) effect: the changes containeden the KML will be applied to the local copy of the file system.

ioc_get_rcvd::
(1) argument: UUID

(2) returns: the received status of a fileset associated with the remote host

ioc_set_fset::

(1) made on: new fileset root
(2) arguments: fileset name, flags
(3) effect: tells kernel to treat a directory root as a new fileset

ioc_mark::

(1) manipulates a flag associated with a cache, fileset or dentry

ping pong ioctls: (always made by cgi):

ioc_revoke_permit: (may not pingpong on client systems)

(1) if permit is on the system where the ioctl is made, the ioctl blocks for active kernel processes to complete
(2) otherwise does upc_revoke_permit to intersync to request intersync fetch the permit from a remote node

ioc_connect:

(1) called from: cgi_connect
(2) argument: port number and ip address http daemon for client system initiating the cgi_connect
(3) does reverse connect through upc_connect
(4) returns server's UUID upon success.

ioc_go_fetch_kml:

(1) called from: cgi_go_fetch_kml
(2) arguments:
(3) effect: upc_go_fetch_kml
(4) description: asks intersync to fetch KML from the caller of cgi_go_fetch_kml

ioc_branch_undo:

(1) called from: incontrol
(2) effect: upc_branch_undo
(3) description: causes intersync to roll back a branch to the parent node

ioc_branch_redo:

(1) made by: incontrol
(2) effect: upc_branch_redo
(3) description: asks intersync to promote a filesystem down a branch from a node.

cgi-initiated non-ping-pong ioctls::

ioc_get_fileid::

(1) called by cgi_fileid
(2) argument: filesetname, pathname, full credentials of caller
(3) returns: file identifier (inode number & generation) of filename or error if file no longer exists or has changed generation
(4) description: this call is made in preparation of doing backfetches by inodes.

ioc_set_kmlsize:

ioctls used from intersync & cgi::

ioc_get_rcvd:

(1) made on: intermezzo directory in a given fileset
(2) argument: a uuid of the client for which the rcvd data is requested
(3) returns: last_remote_recno, last_remote_offset, last_local_offset, last_local_recno for this client
(4) description:

ioc_get_kmlsize:

(1) made on: intermezzo directory in a given fileset
(2) argument: a uuid of the client for which the rcvd data is requested
(3) returns: the KML size on this system, including the last record committed in memory.
(4) description: the file size cannot be used to obtain the size of the KML as the tail may be sparse. This ioctl returns the offset behind the last committed record.

4.4. **Security.** InterSync will support system-to-system security by using ssh tunnels and/or SSL connections to the http servers. Moreover, the intersync client can run with any UID/GID and this will allow KML records to be integrated only when the owner and group of the KML match the own and group of the InterSync client. We expect more sophisticated authorization and authentication for reintegration and data fetches to follow in the future.

4.5. **Features and requirements.** The following tables summarize what components are required for the functionality we have discussed so far.

purpose	**server requirements**
sync clients from server	httpd, intermezzo fs
sync clients to server	httpd (with reintegration CGI), intermezzo fs
...with DOD	same
symmetric two way sync	role switching, client module, server listener
...with client DOD	as before
...with permit handling	as before

purpose	**client requirements**
sync clients from server	connector, client fetch reintegration, any file system
sync client to server	as above add client push KML and push file code
symmetric two way sync	role switching, http protocol server module
...with client DOD	add kernel listener
...with permit handling	add kernel listener
...without server	add RSH server spawning

Data structures and state. The fundamental data structures that are needed during a synchronization run are:

The diagram below shows more of the states and transitions.

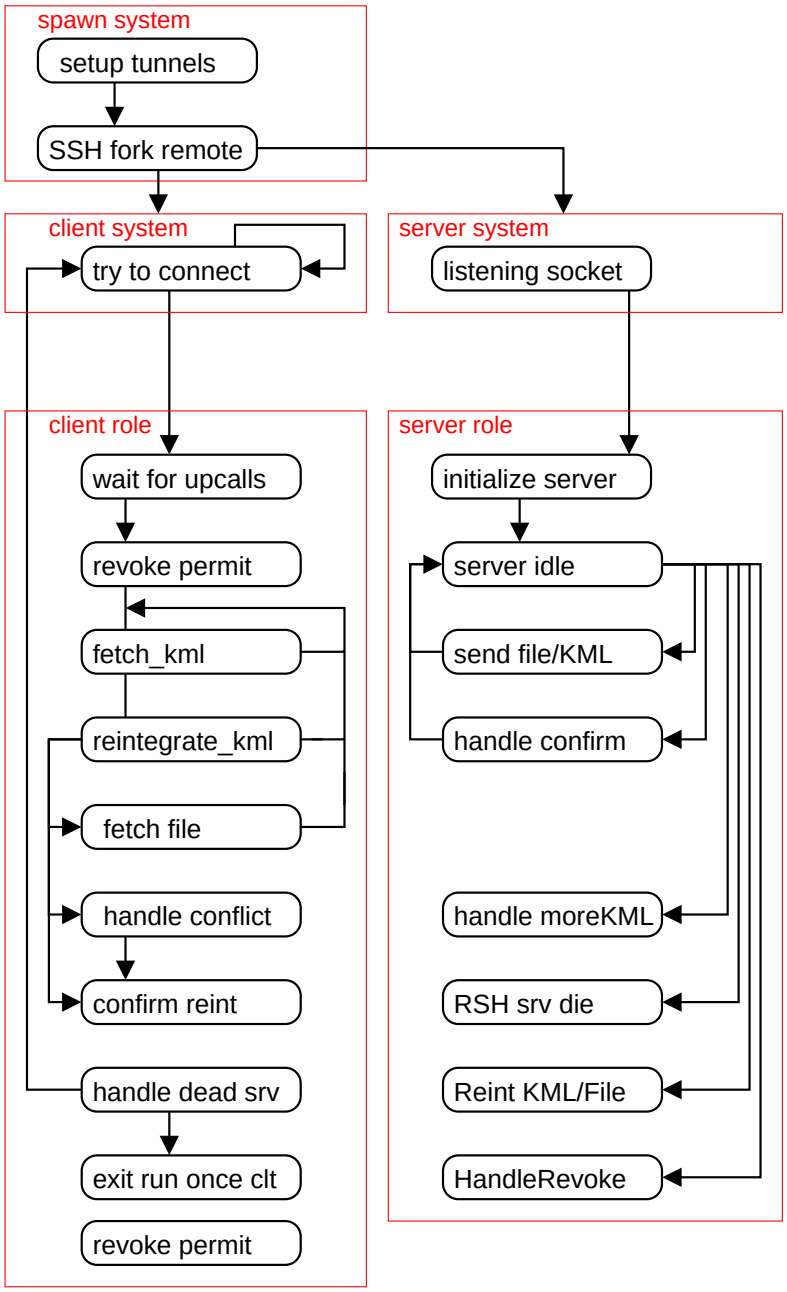
configuration what servers, IP addresses, port numbers, authentication information, filesets, and mount points? This information can be acquired from configuration files and/or from the /proc file system.

servers and connections: information: which connections do we have to an http server. When did we last see a failure of a connection? Has the retry once connection be made. Is this a *run once connection?* Are there pending requests or stolen permits for this server?

client execution context: What segment of the KML do we need to fetch? Which record are we reintegrating? What file are we fetching? Have we revoked the permit?

client reintegration: During reintegration a client may encounter conflicts, or otherwise proceed through the record sequences.

InterSync has been designed in such a way that it can fairly easily be implemented in kernel space. This is particularly attractive for data on demand requests as it will reduce the latency.



5. Applications of InterSync

5.1. **Mobile clients.** InterSync makes for a simple synchronization tool for mobile clients. In this case two way synchronization is desirable and often data on demand is important on resource limited clients. Security can be added to this system in at least two ways: clients can use SSL authenticated and encrypted access to servers using https, or can use ssh to spawn a remote server over a secure ssh connection. The latter requires no system administration or http aliases on the server.

5.2. **High availability clusters.** When a number of servers needs to be replicated InterSync can provide a scalable solution. In failover systems two way synchronization with conflict elimination through rollback is important.

5.3. **Cluster systems management.** When a large cluster of systems is configured almost identically, InterSync using a web server based approach using proxies can provide a scalable solution to synchronize thousands of systems. Of particular importance here are mechanisms to control when systems are allowed to move forward, and as for high availability clusters a rollback feature is desirable. Typically clusters will combine a few two way synchronization systems with a large server used for one way synchronization.

Proxies can be used to increase scalability for fetches, but also to forward notification of changes to replicators. There are minor differences between proxies and servers:

(1) Proxies forward *moreKML* messages to their replicators
(2) When a *permit revocation request* reaches a proxy it handles the request treating the root server as a client.

6. INVOCATIONS

InterSync was designed to make it as simple as possible to synchronize data. Typical invocations are:

- intersync -e ssh user@server:fileset
- intersync http(s)://user@server/fileset
- intersync -e ssh /client/mount/path user@server:/server/mount/path

InterSync can retrieve the information it requires from the /proc/fs/intermezzo directory or from configuration files on both client and server.

Configuration of clients and servers will be simpler if we allow creation of filesets in an InterMezzo mounted file system. Configuring the space for replication can move towards the following simple commands:

- inconfig share filesetname /path/under/intermezzo/mount/point replicator1,replicator2
- inconfig import {-o DOD} server:fileset /path/to/client/dir

7. COMPARISON WITH OTHER SYNCHRONIZATION METHODS

There are many mechanisms to synchronize filesets. The table below gives an overview of the various features which systems offer.

mechanism	InterSync	rsync	NFS/CIFS	AFS/Coda
scan free synchronization (scalability)	yes	no	yes	no
disconnected operation (availability)	yes	yes	no	yes (ro/rw)
persistent cache (performance)	yes	yes	no	yes
track renames (efficiency)	yes	no	yes	yes
change notification (real time)	yes	no	yes	yes
proxies	yes	no	no	no

File systems typically are driven by notifications noted in the kernel. Except for Coda and AFS file systems do not offer a persistent cache, allowing for read performance equal to the local file system. NFS/CIFS do not really synchronize filesets, they merely make them available over the network. AFS and Coda scan the file system for equality of attributes after disconnections.