IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations

Take a deep dive into extending ClearCase 7.0 to ensure the consistency and reproducibility of your software configurations

Foreword by Lars Bendix, Ph. D., ETP, Lund University, Sweden

Marc Girod **Tatiana Shpichko**

[PACKT] PUBLISHING enterprise professional expertise distilled

Table of Contents

Chapter 11. MultiSite Administration..... 1
Setting up the scenery..... 2
Configuration..... 5
Monitoring..... 12

11
MultiSite Administration

Following up on the previous chapter, understanding MultiSite administration cannot hurt. On the contrary, it helps lifting unreasonable expectations: people with an experience of centralized or disconnected systems may assume synchronous behavior (that the data is *in sync*) without really needing it or without considering the implied cost (in terms of waiting). **Asynchrony** in ClearCase MultiSite actually protects the user from depending upon remote (thus both slow and often out-of-reach) resources and events. A good understanding of the difference between latency and bandwidth is a valuable asset in a world where the latter is both well advertised and increasing regularly, and the former is often overlooked and largely incompressible.

Two Open Source version control tools—Mercurial and git—have recently made distributed systems popular. We shall remember that ClearCase was a precursor in this domain as well, even if it may surprise most of its users—mainly those of UCM or of the various web interfaces and Eclipse plugins, who have been driven to see ClearCase as a centralized system. MultiSite comes with an off-the-shelf configuration that favors fairness between all replicas but which is dramatically not scalable. We shall show how to ensure **scalability** without jeopardizing **fairness**: the very value that distinguishes between centralized and distributed systems.

This chapter will be structured along three parts:

- A brief top-down view, with considerations on replica properties and connectivity
- A presentation of two simple but effective enhancement proposals
- A bottom-up review of some troubleshooting cases

Setting up the scenery

MultiSite is implemented with the **multitool** command and the **shipping_server** program. It is somehow unfortunate that multitool is distinct from **cleartool**, although they share a number of operations, which has been growing over time. There also came from the beginning a couple of Perl scripts—`sync_export_list` and `sync_receive`—using a common library `MScommon.pl`. The two scripts are intended for use from the scheduler (`sync_receive` is also suitable for use as receipt handler: see below).

A minor note:

The scripts come, in fact, each as a pair—the real Perl script being found in a `.bat` file with a clever preamble suitable for interpretation by Perl as a dummy array declaration, and by the Windows command as an instruction to pass the file to the bundled (under Windows) `ccperl`. The other files without extension— `sync_export_list` and `sync_receive`—are UNIX shell scripts passing the former to the bundled (under UNIX) Perl.

t some point in time, there came a new family of scripts, together with a `syncmgr_server` program. This new family possibly had some advantages, but it was backwards incompatible and required a synchronous switch on all the sites. This condition proved unacceptable in all the contexts we have met (because of subcontractors sites for instance), so that we'll simply ignore the sync manager and build our configuration on top of the original scripts, to which we made backwards compatible enhancements.

Every vob may be **replicated** to as many sites as needed. The set of replicas builds up a **vob family**—a *replica* on every site. Both kinds of entities have their typed object in the database. In every vob database, there is one vob object and as many replica objects as there are physical replicas, imported on the various sites. Every vob thus maintains a representation of itself and of its siblings.

Events produced on one site are journaled locally into *oplogs*. These oplogs are collected on a schedule, and exported as sync packets for the other replicas. They are then shipped, possibly routed, and finally imported on their destination host. Every replica keeps an *epoch* number of the last oplog it has exported to every other replica, and of the last one it has imported from them. This builds up an *epoch table*: every replica maintains its own. What is important to understand is that at no point in time is there any guarantee that these tables match, and thus that the replicas are *in sync*. On the contrary, it is a safe assumption that there are always some packets on their way, which explains small discrepancies. Obviously, these discrepancies should not grow: the epoch numbers, on the contrary, should grow on every site.

```
$ pwd
/vob/a
$ mt lsrep -s
s1
s2
$ ct des -fmt "%[replica_name]p\n" vob:.
s1
$ ct des -fmt "%[replica_host]p\n" replica:s1
beyond.lookingglass.uk
$ mt lsepoch s1
oid:d5249fcb.011611db.90b6.00:01:83:10:fe:84=32783 (s1)
oid:6ba49d09.011611db.8aa1.00:01:83:10:fe:84=78    (s2)
```

What the table shows here is, on site `s1`, its own *row*, made of lines with the epoch number for every registered replica. For itself this gives the best authorized information: all the oplogs that were effectively created here (`32783`). For the other replicas, this gives the best knowledge we have locally, that is, the number of the last oplog imported from there (`78`).

```
$ mt lsepoch s2
oid:d5249fcb.011611db.90b6.00:01:83:10:fe:84=32781  (s1)
oid:6ba49d09.011611db.8aa1.00:01:83:10:fe:84=78     (s2)
```

If we now look at the row for another replica (`s2`), the values we see are:

- On the line for our own replica, the oplog number (`32781`) we, as the site hosting the s1 replica, last exported to s2, whether or not it could be successfully imported there (even the shipment is unsure).
- On the lines for any other replica, again the best knowledge we have locally: the last oplog known to have been exported from there—`78`. This information may have been received indirectly, that is, from a third replica.

This mechanism shows that the replication is an all or nothing issue on a per vob basis: one cannot avoid sending certain data, confidential for example. Oplogs must be imported in order (serialized): if an oplog is missing, later oplogs cannot be imported.

Permissions preserving

One apparent exception to the previous rule is protection events, which get filtered based on a property of replicas (a creation option that may be changed afterwards). Originally, the choice was only between preserving and non-preserving (and this encompassed both permissions and identities), but later a separate permissions-preserving option was added. The latter is the obvious correct option in most cases (in which NIS or active directory are not shared between sites): it is a good idea to transfer the events that set the execution right to a program or the write permission for group to a directory! For protections events to get transmitted, both the exporting and importing replicas must be preserving of the related flavor.

However, note that this filtering doesn't affect the epoch numbers.

Connectivity

Some multitool commands have `-actual` options—`lsepoch` and `chepoch`. These commands are exceptionally synchronous. They require direct connectivity to the remote hosts and do not usually work through firewalls.

Another command that attempts to bypass the standard replication model is `reqmaster`.

These commands obviously break the fairness between replicas. They may be handy interactively: don't use them in scripts or build a process upon using them!

Note in any case that mastership changes are normal events (whichever command, `reqmaster` or `chmaster`, was used to create them), and they are therefore shipped and imported only after any previous oplog has been successfully imported. While a mastership is in transit, nobody holds it.

This alone shows that transferring mastership should be avoided for any other purpose than administrative (for example, sending the mastership of the replica object after it was successfully imported so that it would be self-mastered, and before deleting a replica).

The rules of thumb for sound multisite development processes are simple (see *Chapter 5*, *MultiSite*), yet seldom followed:

- No mastership transfer
- No remote commands

Especially, one should avoid merging back branches to integration or other upper branches, which obviously may only be mastered on one site.

Configuration

The main problem of the initial MultiSite setup is one of combinatorial explosion. The simplest is to describe it with an example scenario.

Let's take a vob: `/vob/a`, with replicas `s1` and `s2` (we'll designate the replicas and the sites with the same names).

Suppose we create from site `s1` a new replica `s3`, to be shipped and imported to site `s3`. Now, `s2` will start creating and shipping to their destinations, sync packets for both `s1` and `s3`.

Suppose now we create from site `s3` a replica `s4`. On the same schedule as previously, `s2` will now create and ship packets for `s1`, `s3`, and `s4`. And so forth. And every replica will receive packets from every other. All the three phases: export, shipping, import are thus impacted, and for all the existing replicas!

This is clearly suboptimal: there is a lot of redundancy in the contents of the packets. In fact, if we do produce packets to all replicas, they should contain the same data—the only difference coming from the fact that they are produced at slightly different times.

The same is of course true of received packets; they will mostly contain the same oplogs, which will thus get imported once and skipped while importing the following packets— again some useless load.

Finally, the network will suffer artificial congestion, the packets typically taking the same routes for a large part of their paths.

The naive solution is to limit on every site the list of synchronized replicas, and thus to produce packets only for those. This has undesirable side effects:

- It requires manual maintenance of the lists and coordination between the sites
- It stops updating certain parts of the epoch tables, leading to the fact that all replicas are not equal anymore: in case of need, one might not be able to use certain replicas in order to re-create one, which one way or another became corrupted

These first-level issues may be fixed in ways which further break the fairness of the original model, but let's examine our alternatives.

Export

First, let's note that the `sync_export_list` script does not take advantage of the existing ability of `mt syncreplica -export` to create a single packet for multiple replicas. This is the first enhancement we make, and we make it to this script: instead of creating one packet for every replica in the argument list (including the `-all` case), create a common one. This will be reasonable if the replicas are all about at the same epoch level: otherwise, the packet will contain useless oplogs for the replicas nearly up-to-date, as driven by the needs of the most outdated one.

We'll return to exporting oplogs to multiple replicas in the further section, while considering the *hub* function.

In any case we keep the original script and rename our customized version as `sync_export_list_hub`. This leaves the original version for manual use (and backup), and protects our version from being overwritten during an upgrade.

Shipping/routing

The next question is this of shipping: the default behavior of the shipping server is to dispatch the packet to its destinations. We do not change this, but only rely on the `ROUTE` settings in the `shipping.conf` file: if several destinations share a route, the packets will be sent only once and dispatched at the point where the routes split. We consider now setting up a *hub* (hence the suffix appended to the script name). Let's call our hub host *proxima*. The `shipping.conf` files on all the vob servers other than the hub would then contain a single default route:

```
ROUTE proxima -default
```

Shipping servers are needed to cross firewalls, at least if one intends to restrict the range of open ports. The setting in `shipping.conf` allowing us to define a port range must be matched with one in the `albd_server` environment, in the `clearcase` startup script:

```
$ grep 'M.*PORT' /var/adm/rational/clearcase/config/shipping.conf
CLEARCASE_MIN_PORT    49152
CLEARCASE_MAX_PORT    65535

$ grep 'M.*PORT' /opt/rational/clearcase/etc/clearcase
CLEARCASE_MIN_PORT=49152
CLEARCASE_MAX_PORT=65535
export CLEARCASE_MIN_PORT
export CLEARCASE_MAX_PORT
```

This affects all ClearCase processes on the host.

It is obvious that for crossing firewalls, as well as on long-distance connections, the bandwidth cannot be wasted in shipping redundant data.

The names of the packets produced by default by `mt syncreplica -export`, invoked by the `sync_export_list` script, contain the name of the source replicas but not the vob tags. The rationale is that the same vobs may be tagged differently in different sites. Many administrators work around the resulting inconvenience by duplicating the vob tag in the replica names. We note that this practice suffers from the effect described in the previous argument (the vob tags may be different). Next that it creates a pressure to rename the replicas if one re-tags the vobs; finally that it requires some conventions to deal with the case of structured tags (the path separator not being a legal character in replica names), and that longer names have a price in terms of clarity of the outputs. For all these reasons, we cannot recommend this practice and prefer to trade for using a common site name for all replicas.
The vob uuid is anyway found in the packets and easily read from there with the `mt lspacket` command:

```
$ mt lspacket sync_* | grep family | awk '{print $5}' | sort -u
c6052140.08e511d5.b124.00:01:80:e6:b1:a0
$ ct lsvob -fam c6052140.08e511d5.b124.00:01:80:e6:b1:a0
* /vob/foo /vobstg/foo.vbs public (replicated)
```

Where the command is not available (nor the registry to map it to the local tag), i.e., on shipping servers on the way, the packet bears a transient name. We should note that this transient name is not normally exposed, as it is not recorded in the shipping logs, where it is replaced by the original shipping packet name. It is only in the exceptional cases, when the shipping fails, that one can see the shipping packets with names such as `sh_d_...` in the outgoing bay:

```
[shiphost]$ pwd
/opt/rational/clearcase/shipping/ms_ship
[shiphost]$ ll
-r--r--r-- 1 root root 372 Oct 10 14:16 sh_d_99185
-rw-rw-rw- 1 root root 597 Oct 10 14:16 ##################################
                              sh_o_sync_s1_2010-10-10T14.16.09+03.00_9517

[shiphost]$ scp sh_d_99185 vobsrv:/tmp
[vobsrv]$ mt lspacket /tmp/sh_d_99185
Packet is: ######################################################################
         /var/adm/rational/clearcase/shipping/ms_ship/outgoing/sh_d_99185
Packet type: Update
Packet fragment: 1 of 1
VOB family identifier is: eccf73e6.e44e11dc.9acd.00:16:35:7f:04:52
Comment supplied at packet creation is:
Packet intended for the following targets:
 s1 [ local to this network ] tag: /vob/foo
```

And from the shipping order, we can see both the original and the transient packet names:

```
$ cat sh_o_sync_s1_2010-10-10T14.16.09+03.00_9517
# ClearCase MultiSite Shipping Order
# Version 1.0
%IDENTIFIER d65843f5.d4a311df.894b.00:12:3f:93:d3:e9
%CREATION-DATE 1286738169 2010-10-10T14:16:09+03:00
%EXPIRATION-DATE 1287947770 2010-10-24T14:16:10+03:00
%ORIGINAL-DATA-PATH "/opt/rational/clearcase/shipping/ms_ship/outgoing/ #
                                  sync_s1_2010-10-10T14.16.09+03.00_9517"
%LOCAL-DATA-PATH "/opt/rational/clearcase/shipping/ms_ship/outgoing/ ####
                                                              sh_d_99185"
%DESTINATIONS
 s1
%ARRIVAL-ROUTE
 s2 1286738169 2010-10-10T14:16:09+03:00
%DELIVERIES
%FAILED-ATTEMPTS
 R 1277634571 2010-06-27T13:29:31+03:00 s2 s1
%NOTIFICATION-ADDRESSES
%COMMENT
```

Now, we must admit that when grepping remote shipping logs, one has nothing more than the packet names at one's disposal. One can consider implementing a change to `sync_export_list_hub` to use the `-out` option to add a vob identification to the packet name, and use `mkorder` to ship the packet. Taking into account the argument of the possible disparity of vob tags, one might use instead of the tag a special `packet_name` attribute: it would be attached to the vob object and thus replicated, which would guarantee a common value on all sites. This would combine the advantages of the various solutions: save the replica name from having to depend on the vob tag, and publish a shared and stable name for the packets.

Import

Let's now consider the hub. What behavior would be optimal there? What should it do on receiving packets?

Our goal is to reduce the number of packets received by the destinations. The hub will receive packets from multiple origins. How can it combine them so that the number of packets received at any final destination does not grow linearly with the number of replicas? The solution is simple: the hub must itself host a replica, import the packets, refrain from forwarding them to their other destinations (again, altering the default behavior, which is to forward packets that are not for the local replica), and re-export packets containing the compound oplogs on its own schedule.

This implements a version of an *import/export hub*, as opposed to this of a *forwarding hub*, already available in the default `shipping_server`.

More precisely, our changes affect two steps: first, when the script is used as a **proactive receipt handler** (see below), we set a flag `skip_ship` (here is an excerpt from the diffs):

```
+ my $skip_ship = 0;
 if ($CFG::MScfg_proactive_receipt_handler && $CFG::receipt_handler) {
 $cmd = qq/$CFG::MultiTool syncreplica -import -receive -invob $tag/;
+ if ($actual_shiporder and ($sclass ne 'express')) {
+  dbgprint "Do not ship this: $actual_shiporder,\nif the import #####
                                                        is successful. ".
+ "It would be redundant with the sync packet produced later by ######
                                                              the hub\n";
+  $skip_ship = 1;
+ }
... [ other branch not altered ]
```

Then, on import success, and with the flag set, we delete the packet and shipping order to prevent the forwarding (with this second excerpt, we nearly showed all the changes we made!):

```
+ if ($skip_ship) {
+  dbgprint "About to delete $pkt and $actual_shiporder\n";
+  if (open SHORDER, "<$actual_shiporder") {
+   my ($line, $pkt) = ();
+   foreach $line (<SHORDER>) {
+     next if $line =~ /^\s*\#.*/; # skip comment lines
+      if ($line =~ /^\%LOCAL-DATA-PATH \"(.*)\"/) {
+       $pkt = $1;
+      last;
+   }
+  }
+  close SHORDER;
+  if ($pkt) {
+   unlink $pkt, $actual_shiporder;
+   dbgprint "Deleted $pkt and $actual_shiporder\n";
+   $actual_shiporder = 0;
... [ only error reports in the other branches ]
```

Note here that the packets first arrive in the *outgoing* bay, and are only hard linked to the *incoming* one: this is to support the case when the packets would have several destinations and would thus have to be forwarded after being imported. This sets a requirement that the two bays are co-located on the same file system. Note also how the handling of compressed packets gets simplified now (since v7.0), as the `syncreplica` command supports directly the `-compress` switch! Previously, this had to be handled by the `sync_export_list` script.

We must, however, guard against some cases. Before the hub replica itself has been imported, packets must be forwarded to their destination. The same must happen if the local import fails: an error on the hub should not block the replication.

Then, when exporting changes for all the other replicas, we shall often meet the case when the changes came in fact exclusively from one among them (at least for the current export, if the schedule is tight enough). Clearly, there is no point in re-exporting to this replica its own changes!

This is handled in our `sync_export_list_hub` script by the following excerpt:

```
my %remepo = GetReplicaEpochs($vob, $sib);
my $skip = 1;
my $key;
foreach $key (keys %remepo) {
  if ($remepo{$key} < $locepo{$key}) {
    $skip = 0;
    last;
  }
}
next if $skip;
```

We do not attempt anything fancier than skipping a destination if there is nothing new for its replica.

Next, we must be prepared to distribute the hub function among several hosts: one single server holding a replica for all known vobs would soon become a bottleneck. This may however require a dedicated shipping server performing intelligent dispatching on the hub site. The problem here is that the `ROUTE` settings on the various sites are shared between all vobs, without a syntax to discriminate between them. On the other hand, one cannot mandate that the same vobs would be co-located on the same vob servers on all sites. It is thus best if one host may be designated on the hub site to act as the next hop for all routes using the hub. This host will need to refrain from dispatching the packets, before their import has been attempted on the host holding the replica within the hub site.

Finally, we must consider redundancy: one must be able to ship packets via different routes, at least in certain cases. If this can be restricted to certain destinations, the strategy depicted so far is fully sufficient: one only needs to add the route settings to the `shipping.conf`. Supporting this on a per vob basis however requires further adjustments to the scripts, or maybe the use of yet another dedicated shipping server, this time to dispatch packets before shipping them to remote sites.

Receipt handler

The export phase must be scheduled.

The import phase, on the contrary, is better handled as soon as the packet arrives: there is no advantage to wait. This is by the way true as well on a shipping server for the purpose of purely dispatching or forwarding packets. This doesn't preclude the use of a scheduled job to process such packets that, for one reason or another, would have been dropped by the receipt handler.

The receipt handler is invoked by `shipping_server`. One may invoke different handlers per storage class, but it is not obvious how to make use of this feature at its best. We shall assume here a `-default` handler. In `shipping.conf`:

```
RECEIPT-HANDLER -default /opt/rational/clearcase/config/scheduler/ ###
                                                    tasks/sync_receive_hub
```

We also add a `MSimport_export.conf` file under `/var/adm/rational/clearcase/config`, and there we have:

```
proactive_receipt_handler = 1
```

This allows the receipt handler to process other packets than the one just received, if they are for the same vob. This lowers the probability of dropping packets because of missing oplogs.

Shipping server

There is a special ClearCase shipping server installation on UNIX. This one is lighter, and doesn't, for instance, require an access to licenses. It comes therefore neither with *cl*eartool nor multitool. However, the ClearCase *scheduler* functionality is enabled on the shipping server, and indeed, it may be very useful to schedule the following tasks:

- `sync_export_list -poll` (*Daily MultiSite Shipping Poll* job) to poll all the packets in the outgoing bay and shipping them to the next hop
- `sync_receive` (*Daily MultiSite Receive* job), the functionality of which is reduced on a shipping server to moving received packets from the incoming to the outgoing bay

Note that the use of a receipt handler is also possible, and even encouraged. We even propose a trivial receipt handler adapted to the needs of a shipping server. This will forward packets until the maximum number of ports in the firewall range is exhausted. It is thus meant to coexist with a scheduler job, which will process the packets dropped under bursts of activity.

Setting up the scheduler on a shipping server

As we already mentioned, the shipping server installation does not include cleartool. Hence, the scheduler cannot be set up locally using the `cleartool sched` command as described in *Chapter 10, Administrative Concerns*. One can however read the shipping server schedule from a full ClearCase client, using the following command:

```
$ ct sched -host shipping_server_hostname -get -sched
```

One won't be able to change it though, until the scheduler ACLs are set up adequately on the shipping server. The default ACL settings on any ClearCase host are:

```
$ ct sched -get -acl
# Scheduler ACL:
Everyone: Read
```

This grants write permissions only to the local `root` account, and must thus be modified locally first:

```
$ ct sched -edit -acl
```

One can set them as desired, for example, to:

```
# Scheduler ACL:
Everyone: Read
User:<unknown>/joe Full
```

One can then copy the `/var/adm/rational/clearcase/scheduler/db` file from the local host to the shipping server, and with it goes the ACL just set.
After that the shipping server scheduler can be edited remotely (as user `joe`):

```
$ ct sched -edit -host shipping_server_hostname
```

Monitoring

Because `mt lsepoch -actual` won't work through firewalls, one can enable remote hosts replica monitoring with the help of the ***repoch*** script.

Client side (remote host)

The `repoch` script running on the remote host creates a replica monitoring report containing the current epoch number of the replicas hosted on that site and sends the report to the other site by creating MultiSite shipping order for it.

The setup on the client site includes the following steps:

1. Copy `repoch` to `/var/adm/rational/clearcase/scheduler/tasks` directory (make sure it has execution permission for everyone)
2. Edit `/var/adm/rational/clearcase/scheduler/tasks/task_registry` and add the following to the end of it:

```
Task.Begin
Task.Id: 102
Task.Name: "Replica monitoring"
Task.Pathname: repoch
Task.End
```

3. Add a new job to the cleartool scheduler (`ct sched -edit`):

```
Job.Begin
Job.Name: "Daily Replica Monitoring Poll"
Job.Description.Begin:
Send replica monitoring log to the next host.
Job.Description.End:
Job.Schedule.Daily.Frequency: 1
Job.Schedule.FirstStartTime: 00:00:00
Job.Schedule.StartTimeRestartFrequency: 04:00:00
Job.DeleteWhenCompleted: FALSE
Job.Task: 102
Job.Args: --dest destination_host_name
Job.NotifyInfo.OnEvents: JobEndFail
Job.NotifyInfo.Using: email
Job.NotifyInfo.Recipients: root
Job.End
```

Server side (local host)

The local host receives replicas monitoring reports from the remote host using the MultiSite shipping infrastructure and makes them available to the standard `getlog` mechanism. The setup on the server site includes the following steps. We took this in Chapter 10 as an example of *scheduler* configuration, and shall not reproduce the details here.

1. Create a simple shell script (like below) for the purpose of periodically moving the received `repoch` logs from the incoming bay:

```
#!/bin/bash
mv /usr/atria/shipping/ms_ship/incoming/*.repoch* /tmp
```

2. Name it `repoch_mv.sh` and place it in the `/var/adm/rational/clearcase/scheduler/tasks` directory.
3. Add a new scheduler *task* using the `repoch_mv.sh` script.

4. Add the corresponding job invoking the task periodically.
5. Add a new *repoch* entry to the log database named `/opt/rational/clearcase/config/services/log_classes.db`, allowing access to the collected logs with `ct getlog repoch`.

Troubleshooting

Here we will give a few examples of troubleshooting sessions, with the hope of demonstrating routines and tools, which apply to other cases as well.

Missing oplogs

A site may fail to import some packets because they depend on previous ones not found in any available packets. The cure is to generate again the missing oplogs, and to ship and import the packets. For this, one must use the `chepoch` command to set the epochs back in time to the situation required to generate the oplogs, and use `syncreplica` to recreate the missing packets. The appropriate values match the actual situation on the destination site, thus for the destination replica, the epoch number before the missing oplog. Note however that one may also have to change the oplogs on more than one line of the row: if one forgets some, the import will fail and tell which. The information about the epoch values at the various export times is actually also available in the history of the replica object (for the remote replica): see the section titled *History of exports*. This is what the `recoverpacket` command uses.

Just resetting the oplogs may however result in a huge amount of data right away, with no guarantee that it will be directly importable.
We recommend taking a more careful approach:

- Exclude the destination from the scheduled synchronization; this involves using a list of replicas as job argument, whether or not one does it by default.
- Reset the epoch tentatively.
- Send one packet (with multitool syncreplica -export `-max 50k -lim 1`) and record the new epoch value.
- Set the epochs back, so that the avalanche doesn't start, even if the scheduler is reset to its normal state, for example, to synchronize other replicas of the same vob.
- Try to import the packet created and shipped, and record the error if any.
- If the import succeeds, then the epochs will grow, and you may set them to the new value. Note, however, that existing packets in the incoming bay on the destination will become importable at some stage, so there's no need to send all the packets again if only few are needed.

- If the import failed, fix the condition and try again.

Let's illustrate the described approach with an example:

```
$ ct getlog -last 100 sync_import
multitool: Error: Sync. packet /opt/rational/clearcase/shipping/ms_ship/
  incoming/sync_s2_2010-08-04T10.44.12+05.30_23515 was not applied to VOB
  /vobstg/foo.vbs
 - packet depends on changes not yet received
Packet requires changes up to 1369900; VOB has only 1369884 from ########
                                                                replica: s2
...
multitool: Error: Sync. packet /opt/rational/clearcase/shipping/ms_ship/
  incoming/sync_s2_2010-08-04T11.16.19+05.30_1256 was not applied to VOB
  /vobstg/foo.vbs
 - packet depends on changes not yet received
Packet requires changes up to 1482751; VOB has only 1369884 from ########
                                                                replica: s2
```

On the `s1` site, the epoch table looks as follows:

```
$ mt lsepoch -invob /vob/foo
For VOB replica "/vob/foo":
Oplog IDs for row "s1" (@ v1.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1369884 (s2)
Oplog IDs for row "s2" (@ v2.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1369884 (s2)
```

So, it looks like just a tiny missing packet (the missing epoch numbers from `1369884` to `1369900`) has resulted in a huge number of non-imported sync packet in the incoming bay, with epoch numbers up to `1482751` (and actually above that).

Now consider the `s2` site:

```
$ mt lsepoch -invob /vob/foo
For VOB replica "/vob/foo":
Oplog IDs for row "s1" (@ v1.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1489641  (s2)
Oplog IDs for row "s2" (@ v2.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1489641 (s2)
```

Here, the `s2` replica recorded that the sync packets with epoch numbers up to `1489641` have been exported for the `s1` replica and it presumes they were applied there.

The quick and dirty solution would be to reset the `1489641` epoch number to `1369884` and leave it like that until the replicas get synchronized:

```
$ mt chepoch s2@/vob/foo s1=1369884
$ mt lsepoch -invob /vob/foo
For VOB replica "/vob/foo":
Oplog IDs for row "s1" (@ v1.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1369884 (s2)
Oplog IDs for row "s2" (@ v2.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1489641 (s2)
```

But that would result in the "avalanche" export of all the epoch numbers from `1369884` to `1489641` into sync packets in the `s2` replica, and avalanche shipping them all (once again) to `s1` host. This produces a huge traffic of unnecessarily duplicated sync packets that have already been delivered to the `v1` host.

So, a smarter solution would be to temporarily disable `sync_export` (or do it well between its scheduled executions), then reset the epoch number to `1369884` as specified above, generate a small sync packet, and check whether the new increased epoch number is equal to or more than the desired `1369900` number (the point where importing the first packet from the stuck pile can start):

```
$ mt chepoch s2@/vob/foo s1=1369884
$ mt sync -export -max 50k -lim 1 -nc -fship s1@/vob/foo
$ mt lsepoch s1@/vob/foo
For VOB replica "/vob/foo":
Oplog IDs for row "s1" (@ v1.uk):
oid:3b56807b.489a11de.9517.00:21:5e:40:81:8c=3493 (s1)
oid:76d45938.f99a11dd.b047.00:02:c4:65:3f:4c=1369902 (s2)
```

This looks fine now, so we can set the original epoch number `1489641` back to avoid generating any more sync packets, as the small packet we exported should be enough for importing all the sync packets that have accumulated in the incoming bay on the `v1` host:

```
$ mt chepoch s2@/vob/foo s1=1489641
```

History of exports

There is a good deal of information in the history of the replica objects: in the local replica, the history of imports, and in the remote replicas, the history of exports to the respective destinations. This is an important tool to troubleshoot synchronization problem: when did you last successfully import a packet from a given replica? What were the epochs prior to the import? The epochs after the import, you get from the next import event, or if this was the last one, from the current epochs.

The only problem is the verbosity of the records. They contain the epoch for all the replicas, including the deleted ones (which safely serves the needs of the `recoverpacket` command; see earlier in *Missing oplogs*). Here is our last export to the `andromeda` replica:

```
$ ct lshis -last replica:andromeda
--10-05T18:25 root export sync from replica "wonderland" to replica #####
                                                                  "andromeda"
 "Exported synchronization information for replica "andromeda".
 Row at export was: centauri=10758 andromeda=67 wonderland=1377 #########
                                                      centauri.deleted=3"
```

It is typically more convenient to restrict the output to one or a few relevant replicas:

```
$ ct lshis -last 8 -fmt "%d %Nc\n" replica:andromeda | \
  perl -n0777e \
   'while (/^([\dT:+-]+) .*?wonderland=(\d+)[^\n]*$/gms){
     print "$1: $2\n"}'
2010-10-05T18:20:01+05:30: 1377
2010-10-05T14:50:02+05:30: 1374
2010-10-05T11:50:24+05:30: 1371
2010-10-04T18:40:01+05:30: 1370
2010-10-04T17:10:09+05:30: 1368
2010-10-01T19:30:14+05:30: 1365
2010-10-01T19:00:13+05:30: 1360
2010-10-01T18:50:03+05:30: 1354
```

This is again a job for Perl post-processing. This time, we use the "slurp" mode, the paragraph mode with a non-existing character (octal `0777`) as separator. We treat the multiline output as scalar, as a large string. We then repeatedly retrieve the interesting records, from which we every time extract and print, the time stamp and the starting epoch value concerning our own replica. Note the use of the `g` modifier (*Global matching*) to retrieve all the occurrences of matching the regular expression, updating the position to the beginning of the next line at every step.

Consequences of replicas being out of sync

The window of opportunity for creating *evil twin types* (label, attribute, and so on) grows. When the vobs get eventually synched, the names of the remote types clash at import with the ones created locally. They get renamed.

Now, the funny detail is that the format for the renaming has changed at some point (presumably between 7.0.1.1 and 7.0.1.4).

It used to be `<replica_name>:<type_name>` and now changed to `<replica_name>_<type_name>`.

Here is a scenario. We are using a script that creates when needed in the current vob a shared `BldDir` attribute type and applies it. It doesn't matter which site masters the type (which one needed it first), but it matters that both sites use the same if they use any. We used this script at our local replica, `wonderland`, before we noticed that recent packets from the `andromeda` replica had not been imported. When the synchronization is restored, we notice that a `BldDir` attribute type had been created at `andromeda`, the name of which now clashed with the type we created at `wonderland`. During the import, the remote type was automatically renamed to `andromeda_BldDir`. Note that at `andromeda`, the situation is mirrored; the local type is named `BldDir` and the remote one as `wonderland_BldDir`:

```
[wonderland]$ ct lstype -fmt "%n %[type_mastership]p %[master]p\n" \
 -kind attype | grep BldDir
BldDir shared wonderland@/vob/foo
andromeda_BldDir shared andromeda@/vob/foo


[andromeda]$ ct lstype -fmt "%n %[type_mastership]p %[master]p\n" \
 -kind attype | grep BldDir
wonderland_BldDir shared wonderland@/vob/foo
BldDir shared andromeda@/vob/foo
```

How do we rename the type back at the `wonderland` replica? One must first remove the offending local type `BldDir`:

```
[wonderland]$ ct rmtype attype:BldDir
Removed attribute type "BldDir".
```

But even after that one cannot rename `wonderland_BldDir` back locally because it is mastered by the remote `wonderland`.

The solution is to request that the `andromeda` site generate two rename events:

```
[andromeda]$ ct rename attype:BldDir Foo
[andromeda]$ ct rename attype:Foo BldDir
```

On importing these events at the wonderland site, the remote type `andromeda_BldDir` will get back its original name—`BldDir`. In the meantime, you may of course create attributes using the renamed name; they will eventually be of type `BldDir`, upon synchronization between the replicas:

```
$ ct mkattr andromeda_BldDir '"foo/bar"' lbtype:FOO
Created attribute "andromeda_BldDir" on "FOO".
```

Export failures

Here is just an example of a very simple `sync_export` error: `Protect Container failed`. Let's explore how it is presented, how to dig up the essential information, and how to fix it.

```
~> ct getlog -last 34 sync_export
============================================================
Log Name: sync_export Hostname: beyond  Date: 2010-10-06T16:42:14+05:30
Selection: Last 34 lines of log displayed
------------------------------------------------------------
Target replica(s) up to date. No export stream generated.
Generating synchronization packet /stg/shipping/ms_ship/outgoing/ #######
                          sync_wonderland_2010-10-06T15.01.18+05.30_17101
multitool: Error: Vob server operation "Protect Container" failed.
Additional information may be available in the vob_log on host "beyond"
multitool: Error: Unable to change permissions of "c/cdft/18/42/ ########
            5640ca84a17511df80d300018503cbba": No such file or directory.
multitool: Error: Vob server operation "Protect Container" failed.
Additional information may be available in the vob_log on host "beyond"
multitool: Error: Unable to change permissions of "c/cdft/18/42/ ########
            5640ca84a17511df80d300018503cbba": No such file or directory.
multitool: Warning: ../vob_export.cxx:222: operation ####################
                    'vob_ver_get_data' failed: No such file or directory.
multitool: Error: Could not get statistics of the version data file #####
                                                        for this operation.
2153121:
op= checkin_do
replica_oid= 0d33a902.6d6f11df.98d6.00:30:6e:5d:e0:86
oplog_id= 766
op_time= 2010-08-07T04:54:16Z create_time= 2010-08-07T04:54:16Z
version_oid= 5640ca84.a17511df.80d3.00:01:85:03:cb:ba
event comment= ""
cr_info= 0xc7f00
data size= 148 data= 0xb5d08
------------
ver_oid= 5640ca84.a17511df.80d3.00:01:85:03:cb:ba
ver_num= 2
ver_fstat= ino: 0; type: 1; mode: 00
 usid: DONTCARE
 gsid: DONTCARE
 nlink: 0; size: 14154
 atime: Thu Jan 1 05:30:00 1970
 mtime: Sat Aug 7 10:24:16 2010
 ctime: Sat Aug 7 10:24:16 2010
ckout_ver_oid= 5640ca84.a17511df.80d3.00:01:85:03:cb:ba
nsdir_elem_oid= 00000000.00000000.0000.00:00:00:00:00:00
name_p= ""
multitool: Error: Removing incomplete packet /stg/shipping/ms_ship/ #####
                 outgoing/sync_wonderland_2010-10-06T15.01.18+05.30_17101
```

```
ERROR: command './bin/multitool syncreplica -export -maxsize 50m ########
       -fship -limit 1 replica:andromeda@/vob/foo >&2' encountered error.
~> ct getlog -aro 15:00 vob
============================================================
Log Name: vob Hostname: beyond Date: 2010-10-06T16:20:25+05:30
Selection: Lines between 2010-10-06T14:50:00+05:30 and 2010-10-
06T15:10:00+05:30 displayed
------------------------------------------------------------
2010-10-06T15:01:35+05:30 vob_server(1766): Error: unable to access file
 c/cdft/18/42/5640ca84a17511df80d300018503cbba: No such file or directory
2010-10-06T15:01:35+05:30 vob_server(1766): Error: Unable to chmod
 container /vobstg/foo.vbs/c/cdft/18/42/5640ca84a17511df80d300018503cbba:
 No such file or directory
...
```

We take the oid shown in the log, which is also the base name of the container, just formatted differently. This allows us to find the exact version corresponding to this oid, by executing a `describe` command in a view context, from a directory in the vob:

```
$ ct des -s oid:5640ca84.a17511df.80d3.00:01:85:03:cb:ba
/vob/foo/bar/example@@/main/mg/2
$ file /vob/foo/bar/example@@/main/mg/2
/vob/foo/bar/example@@/main/mg/2: commands text
```

Back to the vob storage:

```
$ ls -l /vobstg/foo.vbs/c/cdft/18/42/5640ca84a17511df80d300018503cbba
-r-xr-xr-x 1 vobown jgroup  14154 Oct 6 16:24 /vobstg/foo.vbs/c/cdft/ ###
                                    18/42/5640ca84a17511df80d300018503cbba
```

Now the once missing cleartext container is found where expected. It is of course the `file` command that forced the generation of the container, and this alone was enough to make the export succeed! The reason for the error was that exporting the event required protecting the cleartext container, but this one had been scrubbed, and there was no instruction to recreate it. Such a recreation happens automatically, but requires a view context, as in our example above.

Incompatibility between ClearCase releases

It may happen that in a ClearCase MultiSite setup, different sites run different ClearCase versions; even such versions that support different feature levels (for example, 7.0 with FL 5 and 2003.06.00 with FL 4). When a new vob is created on a site with FL 5, and replicated to a site with a lower feature level, the replica package is not importable at the destination.

To support MultiSite inter-operability between such sites, the feature level of new vob must explicitly be set low at vob creation time, by using the `-flev target_site_fl` option of `ct mkvob`.

```
[joe@v1 ~]$ ct hostinfo -l | grep Product
 Product: ClearCase 7.0.1.6
[joe@v1 ~]$ ct hostinfo -host v2 -l | grep Product
 Product: ClearCase 2003.06.10+
```

So, the site `s1` has a default feature level of 5, and the site `s2` supports only FL 4. Here is how to create a new vob on site `s1`, with the intention to replicate it to site `s2` (to a vob server `v2`):

```
$ ct mkvob -tag /vob/foo -flev 4 -nc -stgloc -auto
$ ct des -fmt "%[flevel]p\n" vob:/vob/foo
4
```

We can now create a replica for `s2` site, as usual:

```
$ mt mkrep -exp -work /tmp/foo -nc -fship v2:s2@/vob/foo
```

MultiSite shipping problems—a tricky case

Here is a known problem with ClearCase MultiSite shipping, easy enough to produce accidentally, but tricky to debug.

This will be the occasion to show the use of `mkorder` and of some under-documented debugging features of ClearCase: special environment variables, and the debug option of `shipping_server`. All of these apply for sure to a large range of ClearCase MultiSite administration cases.

The MultiSite configuration in this case is the following: two sites with a vob server and a shipping server on each side of a firewall. A range of ports is open in the firewall between the two shipping servers, so that they may contact each other, but not the other site's vob server through the firewall.

Let's call the vob servers `v1` and `v2`, and the shipping servers `sh1` and `sh2`.

The problem symptoms: shipping goes successfully in one direction (for example, `v1-> sh1 -> sh2 -> v2`), but fails in the other with the following messages:

```
$ cleartool getlog -since yesterday -host sh2 shipping
07/06/2009 02:56:12 PM shipping_server(23352): 4691): Error: unable to ##
              deliver/forward order /usr/atria/shipping/ms_ship/outgoing/
              sh_o_s2.ddd_1_46
07/06/2009 02:56:12 PM shipping_server(23352): 4691): Error:
A shipping_server RPC failed. Additional information may be ##############
                    available in the albd_log on the destination machine.
```

```
07/06/2009 02:56:12 PM shipping_server(23352): 4691): Error: ############
                  shp_forward_request_V1: RPC: Unable to receive; errno =
                  Connection reset by peer
07/06/2009 02:54:51 PM shipping_server(23352): 4691): Error: ############
                              Unable to contact albd_server on host sh1
07/06/2009 02:54:31 PM shipping_server(23352): 4691): Error: ############
                                    connect failed: Connection timed out
```

We check that `sh2` can contact the `albd_server` on `sh1`:

```
[sh2]$ albd_list sh1
albd_server addr = 155.111.22.33, port= 371 PID 10962:
shipping_server, tcp socket 49158: version 1; BUSY PID 29560:
shipping_server, tcp socket 49164: version 1; BUSY PID 30105:
Albd_list complete
```

So, let's create a test packet and ship it with `mkorder` from the shipping server `sh2` to the vob server `v1` (advantage: it will not get imported):

```
$ hostname
sh2
$ touch /tmp/foo
$ mkorder -data /tmp/foo -fship v1
Shipping order "/tmp/sh_o_foo" generated.
Attempting to forward/deliver generated packets...
---- NOTE: consult log file ##############################################
        (/var/adm/rational/clearcase/log/shipping_server_log) for errors.
mkorder(19770): Error: Store-and-forward server ##########################
       "/opt/rational/clearcase/etc/shipping_server" failed with status 1
mkorder(19770): Warning: Unable to send packet /tmp/foo ##################
                                                (see store-and-forward log)
```

We check that the error is logged:

```
$ cleartool getlog -host sh2 shipping
[...] shp_forward_request_V1: RPC: Unable to receive; errno = ############
                                                      Connection reset by peer
```

We verify that shipping from `sh1` to `v2` works:

```
[sh1]$ mkorder -data /tmp/foo -fship v2
Shipping order "/tmp/sh_o_foo" generated.
Attempting to forward/deliver generated packets...
-- Forwarded/delivered packet /tmp/foo
$ tail -1 /var/adm/atria/log/shipping_server_log
07/06/09 16:25:37 shipping_server(6515): Ok: Forwarded order #############
                                    /tmp/sh_o_foo (data "foo") to host "v2"
```

Let's now start `shipping_server` with options `-server -d`, and set the ClearCase environment variables—`TRACE_SUBSYS` and `TRACE_VERBOSITY`—to monitor the shipping session. Note that `TRACE_SUBSYS` may be used with any subsystem to restrict the verbosity to the one of current interest. This is the server side, at `sh1` host in our case, where we start the shipping server process in the verbose debug mode:

```
$ export TRACE_SUBSYS=shp_subsys
$ export TRACE_VERBOSITY=4
$ /usr/atria/etc/shipping_server "" -server -d
>>> (shipping_server(4956)): Configuration
>>> (shipping_server(4956)): -------------
>>> (shipping_server(4956)): max data size 2097151
>>> (shipping_server(4956)): minimum port setting 49152
>>> (shipping_server(4956)): maximum port setting 49171
>>> (shipping_server(4956)): notify_prog = /usr/atria/bin/notify
>>> (shipping_server(4956)): storage classes
>>> (shipping_server(4956)): -default ---> /usr/atria/shipping/ms_ship
>>> (shipping_server(4956)): -default ---> /usr/atria/shipping/ms_rtn
>>> (shipping_server(4956)): routes[...]
>>> (shipping_server(4956)): sh2: -default
>>> (shipping_server(4956)): administrators (1)*** listening on #########
   port 49153 ***And we record the port number it is listening on: 49153.
```

Now, from our client side, on `sh2`, we can open a shipping session, specifying the exact port used on the server side (`49153`):

```
$ mkorder -copy -ship -data /tmp/foo v1
Shipping order "/opt/rational/clearcase/shipping/ms_ship/outgoing/ ######
                                                    sh_o_foo" generated.
$ export TRACE_VERBOSITY=4
$ export TRACE_SUBSYS="*"
$ shipping_server /opt/rational/clearcase/shipping/ms_ship/outgoing/ ####
                                                  sh_o_foo -d sh1:49153
shipping_server(692): Error: Operation "connect" failed: No such file ###
                                                        or directory.
$ cat /opt/rational/clearcase/shipping/ms_ship/outgoing/sh_o_foo
# ClearCase MultiSite Shipping Order
# Version 1.0
%IDENTIFIER c5dfcedc.263247dd.a40e.f3:83:5e:db:d9:3b
%CREATION-DATE 1082014994 06-Jul-09.10:43:14
%EXPIRATION-DATE 1083224595 06-Jul-09.10:43:15
%ORIGINAL-DATA-PATH /opt/rational/clearcase/shipping/ms_ship/outgoing/ ##
                                                              sh_o_foo
%LOCAL-DATA-PATH /opt/rational/clearcase/shipping/ms_ship/outgoing/ #####
                                                              sh_o_foo
%DESTINATIONS v1
%ARRIVAL-ROUTE%DELIVERIES
%FAILED-ATTEMPTS
```

```
R 1082015248 06-Jul-09.10:47:28 sh1 v1
%NOTIFICATION-ADDRESSES
%COMMENT
```

On the server side one receives the following output:

```
>>> (shipping_server(4956)): socket addr is 0.0.0.0
shipping_server(4956): Error: timeout waiting for transfer RPC request
```

The investigation showed that the final destination matters—shipping only to `sh1` worked:

```
[sh2]$ mkorder -data /tmp/foo -fship sh1
Shipping order "/tmp/sh_o_foo" generated.
Attempting to forward/deliver generated packets...
-- Forwarded/delivered packet /tmp/foo
$ tail -1 /var/adm/atria/log/shipping_server_log
07/06/09 16:25:37 shipping_server(6515): Ok: Forwarded order ############
                                  /tmp/sh_o_foo (data "foo") to host "sh1"
```

But mentioning `v1` as the destination caused a failure, as we demonstrated earlier.

In this case, the cause of the problem was a misconfiguration of the DNS, mentioning a decommissioned server:

- The shipping server always attempts to resolve the destination, and this implies by default to issue a DNS request
- In the case that the DNS server doesn't reply, the shipping session times out
- The quick work-around was to add the destination to `/etc/hosts` (as `/etc/nswitch.conf` instructed to use *files* before *DNS* to resolve host names)

The actual solution was of course to fix the DNS servers in `/etc/resolv.conf`.

The tricky problem here is that for shipping purposes and routing via a shipping server, resolving the actual destination (the vob server behind a firewall) is, of course, completely useless: the host name is likely to be non resolvable, its IP address non reachable. The difference in the shipping server behavior is dramatically different though depending on whether it does receive a negative answer from a configured DNS server (such as "`destination host is not resolvable`") within a certain time limit, or it keeps waiting for the pending DNS answer (in case the DNS itself is misconfigured) and the shipping session terminates then after a timeout.

Summary

Once again, we were able to show some scenarios in which users could solve their problems by themselves without the need for administrative rights or special skills, just by paying careful attention to the error messages. In the last case of course, the administrator was eventually required, but the decisive step was to use the `mkorder` tool to analyze the shipment into simpler steps that could be shown to succeed. Even the MultiSite architecture, with its routing tables and shipping servers, is not as complex as it might seem, and information may be brought closer to the users as we shown with the remote epoch monitoring.

It is our experience that end users are willing to understand whether a delay in synchronization is normal, or the symptom of a problem to report or to investigate.