

# Table of contents

<b>Definitions</b> .....	1
<b>Problem</b> .....	1
<b>Assumptions</b> .....	1
<b>Requirements</b> .....	1
<b>Algorithm overview</b> .....	2
<b>Replication state diagram</b> .....	4
<b>Protocol</b> .....	5
<b>Proof of the algorithm correctness (semi-formal)</b> .....	6
<b>Considerations for the chosen algorithm</b> .....	8
<b>Implementation</b> .....	8
<b>Commands sent between daemons</b> .....	8
<b>Configuration file additions</b> .....	9
<b>Class diagram</b> .....	12
<b>Classes' structure</b> .....	13
<b>Dependency matrices</b> .....	13
<b>Protocol implementation</b> .....	15
<b>HAD configuration without replication</b> .....	17
<b>Testing</b> .....	18
<b>To do and future development</b> .....	22

## Definitions

1. Semi-coupled (or alternatively semi-decoupled) design of HAD and replication modules – unidirectional protocol between the aforementioned modules, when the HAD module affects the replication one, but not the vice versa.
2. Replication leader / leader – machine, responsible for the system state replication
3. Replication backup / backup – machine, which is to be a substitute for the replication leader upon the latter's failure
4. Newly joined machine – machine, joining the pool, which is neither replication leader nor replication backup yet

## Problem

Certain elements of the system are critical to its operability (such as Accountantnew.log file, which is responsible for fairness in exploiting Condor's resources amongst different users)

## Assumptions

1. A machine is accessible iff its HAD is accessible
2. Negotiator runs on one machine only, HAD runs on every machine in the pool
3. State is characterized by one file, which is located on the negotiator machine
4. No shared file system in the pool
5. The pool connectivity is symmetric and transitive
6. The protected state file format is unknown to the replication service

## Requirements

1. At any given moment in stable system all the backups possess either the most updated state file version or the one that preceded to it

2. Upon active HAD failure the new active HAD will pick up the most updated version amongst all the versions in the pool
3. Upon joining new machine picks up the most updated pool version to that moment
4. Upon merging two networks (when 2 negotiators are able to communicate with each another), the remaining negotiator picks up the file, that is a merge between two most updated versions in both networks

## Algorithm overview

We decided to separate between the implementation of HAD module and replication module, since the HAD functionality is critical to the proper functioning of the whole system, whereas the replication can be seen as nice-to-have feature, consequently HAD module robustness becomes better after being semi-decoupled from the replication functionality. This explains the design decision to have two separate daemons called 'condor\_had' and 'condor\_replication'.

Another design decision is that all the pool machines (except for the joining ones and, of course, the replication leader itself) are backups for the leader. The advantage of this decision (vs. having limited amount of backups) is better system robustness. Its disadvantage, however, is degradation in performance, since more time is now spent on sending updates to backups.

Our algorithm employs state file versions, which designate the state file in different time points. The greater the version, the more updated the state file. The only machine that may change the version number is the replication leader. The only operation that backups and newly joining machines may do with version is to download it from the replication leader machine. Since we eventually want to take care of two networks reconciliation, we introduce the notion of gid, which stands for the belonging to certain pool of machines. Again the gid may be changed by replication leader only.

The algorithm makes differentiation between stable and unstable machines. Unstable machines are the joining ones, their version hasn't been determined yet. Stable machines are backups and replication leader. In our algorithm we preserve the following invariant: in a stable pool (no heavy network interruptions) the worst pool version and the best pool version differ by no more than one version number, which minimizes the damage of the replication leader machine's failure.

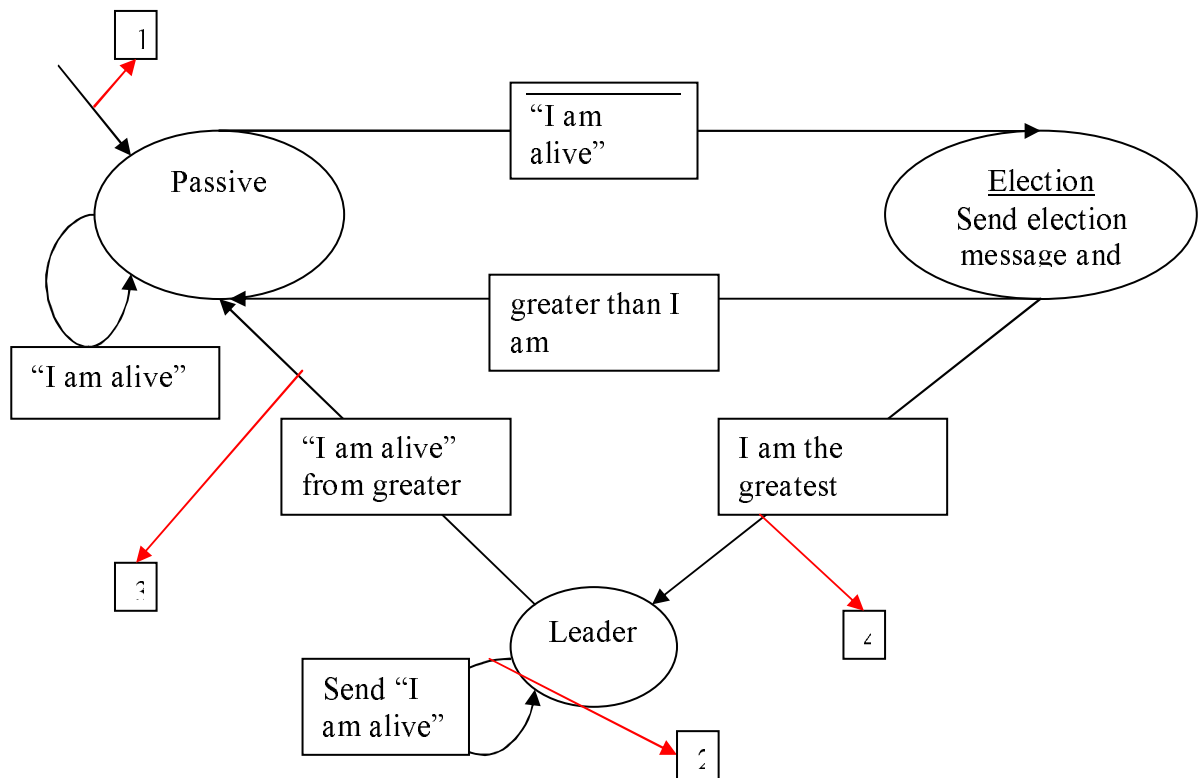
An important advantage of the algorithm is that it treats the protected state file as black box, thus not being dependent on its format.

All the communication between replication daemons and between HADs and replication daemons is performed by means of message sending, which increases the algorithm's robustness, since no shared file system is assumed.

Lest to stall the replication daemon, the downloading/uploading of state file is done by means of separate processes called transferers. Transferer can be uploading or downloading. It downloads/uploads the state file, sent from the leader machine to some backup, and work asynchronously to the replication daemon, thus preventing the replication daemon from being locked till the downloading/uploading finishes (liveness feature of the replication daemon).

Replication daemon has got its own state diagram, different from (though strongly affected by) the HAD one.

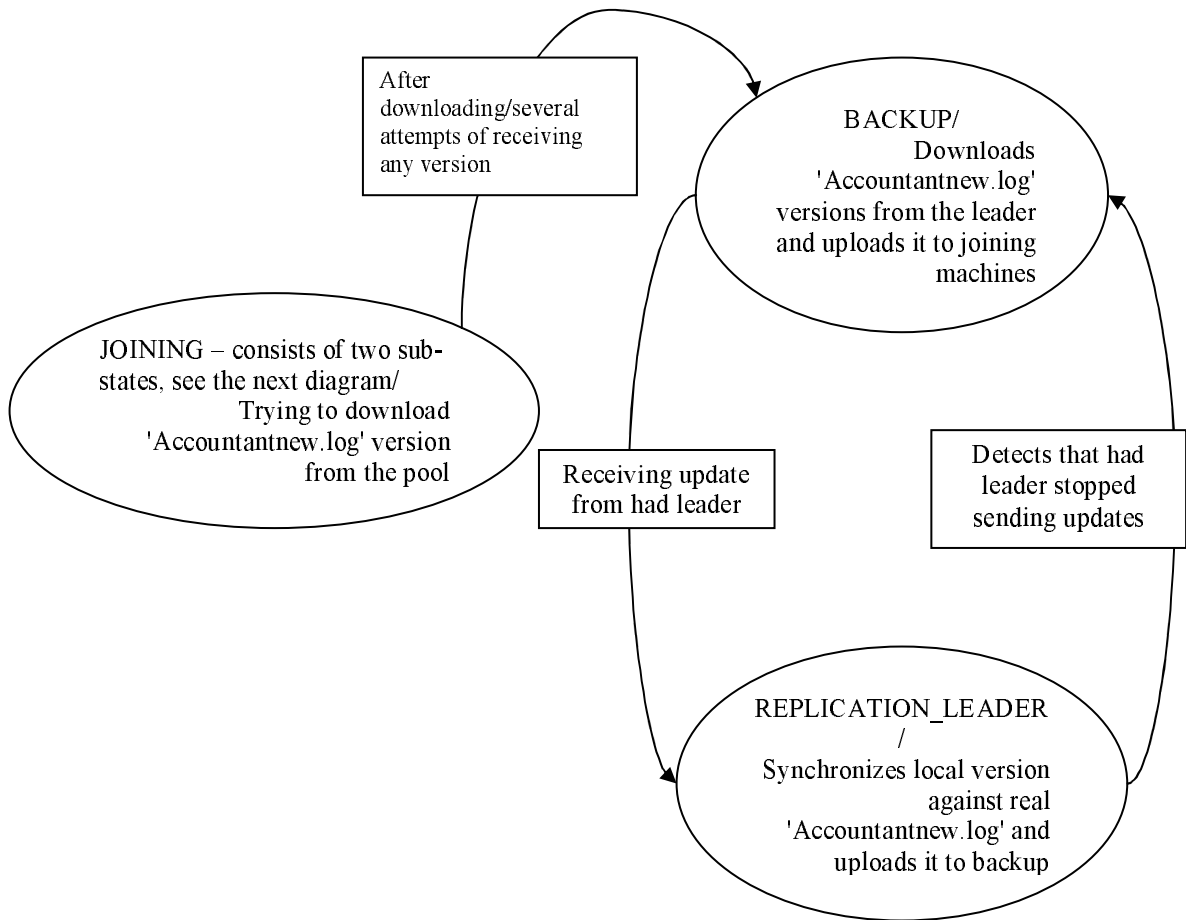
HAD daemons report their local replication daemons (by means of message sending), in which state transition they currently are. This can be best seen on the HAD state diagram:



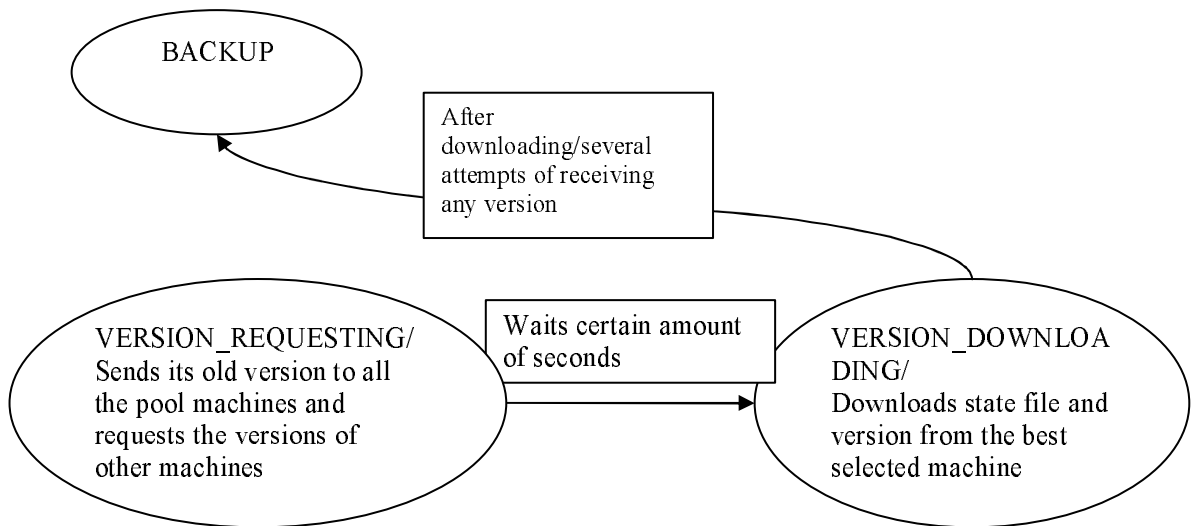
The messages, sent to replication daemon are:

- 1 – HAD\_BEFORE\_PASSIVE\_STATE
- 2 – HAD\_IN\_LEADER\_STATE
- 3 – HAD\_AFTER\_LEADER\_STATE
- 4 – HAD\_AFTER\_ELECTION\_STATE

# Replication state diagram



## JOINING state zoom in:



# Protocol

We decided to explain the whole algorithm by means of collection of interesting use cases, which altogether cover the functioning of the entire algorithm.

**Number** : 1

**Description** : replication daemon life cycle

**Flow** :

- 1) The replication daemon resets the replication timer
- 2) Joining machines and backups return
- 3) Leader checks its state file
- 4) If it is changed

4.1) It increments the version of the file

4.2) It broadcasts its version in the pool

Upon receiving the command:

\* leader and joining machine: ignore it

\* backup: receives the version, checks whether its own version is less updated, and if it is so, downloads the state file from the leader

5) Leader checks whether since last alive time of its HAD more than certain amount of seconds have passed

6) If so, it broadcasts its version in the pool and passes to BACKUP state

**Number** : 2

**Description** : new machine joins the pool

**Flow** :

- 1) Newly joined machine solicits all pool machines' versions

Upon receiving the command,

\* joining machines: ignore it

\* backup and leader: send their versions to the newly joined machine

3) Newly joined machine waits certain amount of seconds, in which it collects all the pool versions that were sent to it

4) Newly joined machine passes to VERSION\_DOWNLOADING state

5) Newly joined machine chooses the best version amongst those that were sent to it and downloads it from the remote machine

6) Newly joined machines passes to BACKUP state

**Number** : 3

**Description** : backup receives HAD\_AFTER\_ELECTION\_STATE message

**Flow** :

1) Backup sets last alive time of its HAD to the current time

2) Backup chooses new gid for the pool

3) Backup changes its state to REPLICATION\_LEADER

**Number** : 4

**Description** : leader receives HAD\_IN\_LEADER\_STATE message

**Flow** :

1) Leader sets last alive time of its HAD to the current time

**Number** : 5

**Description** : leader receives HAD\_AFTER\_LEADER\_STATE message

**Flow** :

1) The giving up leader broadcasts its version in the pool

Upon receiving the command

- \* joining machine and backup: ignore it
  - \* leader: chooses new gid for the pool (supplied with merging utility in the future, we will merge two state files here)
- 2) The giving up leader passes to BACKUP state

## Proof of the algorithm correctness (semi-formal)

### *Definitions:*

- Chronological log of state file changes – timestamps of changes to the accounting information file detected by the leader machine written in chronological order (of absolute time) and enumerated, i.e. the log of all the changes applied to state file ever
- Chronological number – the number of timestamp when some change to state file was detected by replication leader. Chronological number's increasing can be caused by replication leader only
- Replication interval – period of time between two successive awakenings of replication daemon

### *Assumptions:*

- Possible network interruptions are temporary
- Passing from state to state is atomic operation and its performance time can be neglected
- File uploading/downloading time is significantly less than replication interval
- No reconciliation with another network occurs in our machines cluster

### *Reservation:*

The worst and the best version of state file amongst cluster machines in BACKUP or REPLICATION\_LEADER states match to the two timestamps in chronological log. We claim that these timestamps' numbers do not differ by more than 1.

### *Proof:*

We prove by induction on possible events that change the cluster from the viewpoint of our replication system.

### *Basis:*

The induction basis is when there is no machine in BACKUP or REPLICATION\_LEADER in our cluster. It is vacuously true, because there are only joining machines in our cluster.

Let's assume that in the cluster state when no event occurs the reservation is true.

### *Possible events that can occur in our cluster (from the viewpoint of replication system):*

- 1) Replication leader machine/daemon failure
- 2) Replication backup machine/daemon failure
- 3) Joined machine passed to BACKUP state
- 4) Replication daemon is informed by the HAD leader, that it is the replication leader
- 5) Replication leader detects that HAD leader does not send the updates anymore
- 6) Replication leader detects that state file has changed and sends the updated version to the cluster machines (after being a leader at least one replication interval)
- 7) Downloading of the updated version from the leader of state file by backup machines

1-2. Upon replication leader/replication backup machine/daemon failure no change is made to the chronological log of state file and no change is made to any of the existing copies of state file (of machines in BACKUP and REPLICATION\_LEADER states) inside the cluster so the reservation remains true.

3. When a new machine joins the cluster, it keeps trying till it downloads the accounting information file version from the cluster. If the file has been downloaded from the replication leader, then its number in chronological log is equal to the best number inside the cluster. Otherwise if the file has been downloaded from some backup machine, then the chronological number of the newly joined machine is less by 1 than the best number inside the cluster in the worst case (in case when the replication leader failed after sending the most updated version to part of the cluster machines and there was no replication leader at the time of file downloading by the newly joined machine)

4-5. Either when the replication daemon is announced the replication leader by the HAD leader or when it detects that the updates are not sent anymore, no change to local state file occurs, so the reservation remains true. Besides, after the replication daemon is announced the replication leader by the HAD leader, it sends the versions updates to all the cluster machines, meaning that the cluster chronological numbers are equal inside the cluster after all the downloads finish

6. When the replication daemon detects a change inside state file (after being a leader at least one replication interval), all the versions inside the cluster are equal. After the modification is detected the chronological number of local state file grows by 1 (which means that the reservation is still true, because the worst chronological number is less by 1 than the leader's number). After that the leader sends the update to all the backups and the cluster chronological numbers are equalized again

7. When the state file is downloaded by the backup machine, its chronological number becomes equal to the replication leader's one (pay attention that the chronological number of the backup necessarily increases by 1). Since the replication leader's chronological number did not break the reservation, the backup machine's new chronological number does not break it either.

This finishes the proof of our replication system reservation. The reservation is the *safety* reservation, which means that, in order to accomplish the correctness proof, we have to provide the proof of *liveness* property of the system. It actually stems from the correctness of negotiator daemon functionality. That means that if the user decides to submit a job, the accounting information file will be eventually updated by the negotiator daemon, which turns on the detection mechanism of the replication system, resulting in updating all the backup machines with this new version of state file.

### *Conclusions:*

Optimizing the algorithm to cope with networks reconciliation case, means that cluster machines can sometimes receive more than one version, in which the 'am I replication leader' flag is set to 'true'. This is an evident conflict, which the cluster machine must be able to resolve. Any consistent resolution of the conflict is appropriate then, e.g. taking the latest modified file, merging the files altogether (being provided with the merging utility) or any another heuristic.

End of proof

## Considerations for the chosen algorithm

Several ideas were suggested for the replication algorithm. One of the considered options was the following algorithm:

### **Do forever**

**Broadcast local version of state file to the machines pool**  
**Asynchronously receive remote versions and amongst them choose the best one**

**If the best chosen version is better than the local one then**  
**Download the appropriate state file from the remote machine**

**If the machine is HAD leader then**  
**Check if the local state file modification time has changed and if so**  
**Increment the state file version by 1**

This specific algorithm failed to resolve properly the following case. Suppose, there was a pool of machines, the greatest version number of which was 1000. Then the pool of machines was terminated. After that another pool of machines started and its version reached 100 at the moment when one of the machines of the previously launched pool was revived. The version 1000 is much better than 100, so it turns out that the old state file overwrites the new state file, which is exactly the case we wish to avoid.

Although the presented algorithm does really avoid the above case, however, it may fail in quite similar cases: since the format of the protected state file is considered unknown, the algorithm cannot base its decisions upon the state file semantics. The solution for the problem is to construct a utility, which is able to understand the state file format. Merging two state files is good example for such a utility.

## Implementation

We decided to make the implementation generic, which means we defer certain functionality to specific implementation. So, for example, the handlers for messages, sent from HAD to replication daemon are all abstract functions and their implementation is postponed to concrete classes. Besides, the handler for choosing the best version out of local collection of versions and the handler for choosing the new gid of the pool are abstract too. We will present our own default implementations of these handlers shortly.

Besides, we provide functions of replicator that might be useful in any replication algorithm implementation, such as

- \* `broadcastVersion( command )` – sends command, containing a local version to all pool machines
- \* `requestVersions` – solicit other pool machines' versions
- \* `download( address )` – downloads state file from remote machine at the specified address
- \* `upload( address )` – uploads local state file to the remote machine at the specified address

## Commands sent between daemons

- 1) `REPLICATION_LEADER_VERSION` – replication leader sends this to all the machines, when its state file has changed
- 2) `REPLICATION_TRANSFER_FILE` – downloading 'condor\_transferer' sends the command to 'condor\_replication' daemon to start uploading 'condor\_transferer' (to start transferring the state files from the leader to the backup)

- 3) HAD\_BEFORE\_PASSIVE\_STATE  
HAD\_AFTER\_ELECTION\_STATE  
HAD\_AFTER\_LEADER\_STATE  
HAD\_IN\_LEADER\_STATE – commands from HAD, sent in different transitions between its states
- 4) REPLICATION\_NEWLY\_JOINED\_VERSION – sent by the newly joined machine to all the machines in the pool; contains the local version
- 5) REPLICATION\_GIVING\_UP\_VERSION - sent by the giving up replicator to all the machines in the pool; contains the local version
- 6) REPLICATION\_SOLICIT\_VERSION – sent by the newly joined machine to all the machines in the pool to get all the pool replicas  
REPLICATION\_SOLICIT\_VERSION\_REPLY – sent by pool machines to the newly joined machine; contains the local version and the sending replication daemon state

## Configuration file additions

The following entries are being added to the configuration file:

- 1) **REPLICATION\_LIST** – the comma-separated list of ip:port or host:port entries, optionally enclosed in <> branches

### **Description:**

The following list contains entries for all known replication daemons in the pool for the configuration machine, including the port numbers for each one of them. Normally each machine's replication list must contain the same entries, necessarily in the same order.

- 2) **STATE\_FILE** – name of the file from the SPOOL directory that will be replicated

### **Description:**

This file is protected by the replication mechanism. It is replicated between all the replication daemons listed in REPLICATION\_LIST configuration parameter.

**Default:** \$(SPOOL)/Accountantnew.log

- 3) **REPLICATION\_INTERVAL** – period of time between two successive awakenings of the replication daemon

### **Description:**

This parameter determines how frequently the replication daemon wakes up to do its periodic activities: probing for update of the state file, broadcasting the update to backups, monitoring and managing the downloading/uploading process by transferer processes etc. Since the accounting information file normally changes, as negotiator daemon wakes up, then REPLICATION\_INTERVAL value must be like NEGOTIATOR\_INTERVAL. That is why default REPLICATION\_INTERVAL equals to default NEGOTIATOR\_INTERVAL.

**Default:** 300

- 4) **MAX\_TRANSFER\_LIFETIME** – period of time, in which transferer daemons have to accomplish the downloading/uploading process

### **Description:**

This parameter is intended to handle the case of stuck transferer processes: be it a network interruptions or huge state file or any other reason. During this period of time the transferers must download the file from leader to backup machine. We advise this parameter be equal to

2 \* Average size of state file / network rate

**Default:** 300

5) **NEWLY\_JOINED\_WAITING\_VERSION\_INTERVAL** – period of time, which the newly joined machine waits from requesting pool versions to selecting the best of them

**Description:**

Before entering the pool, newly joining machine must download the best replica of the state file from the pool. In order to do that, it requests for the state file versions from all the pool machines and waits certain amount of time to let the machines react and send their versions to it. Its value depends on the effective network rate, set it to higher value than default, if the networking is really slow or if the security is enabled.

**Default:**  $2 * (\text{HAD\_CONNECTION\_TIMEOUT} + 1)$

6) **HAD\_UPDATE\_INTERVAL** – period of time between two successive sends of classads to the collector daemon

**Description:**

HAD sends classads to the collector once in a period of time, defined by this configuration parameter in order to let the collector know the most updated information about the state of HADs in the pool. Upon each change in the HAD state (like turning from active to passive or vice versa), the collector is being updated immediately.

**Default:** 300

7) **HAD\_USE\_REPLICATION** – whether the replication feature should be used by the HAD daemon or not

**Description:**

This configuration parameter enables administrator of the machine disable/enable the replication feature on Condor machine configuration level.

**Default:** No

8) **REPLICATION\_ARGS** – arguments passed to *condor\_replication* daemon by *condor\_master*, when it is started

**Description:**

Command line arguments passed by the *condor\_master* daemon as it invokes the *condor\_replication* daemon. To make replication work, the *condor\_replication* daemon requires the port number to use. This argument is of the form

-p \$(REPLICATION\_PORT\_NUMBER)

where REPLICATION\_PORT\_NUMBER is defined with the desired port number. Note that this port number must be the same value here as used in REPLICATION\_LIST.

**Default:** No

9) **REPLICATION** - path to the replication daemon executable

**Description:**

The path to the *condor\_replication* executable. Normally it is defined relative to \$(SBIN).

**Default:** No

10) **MAX\_REPLICATION\_LOG** – maximum length of the replication daemon log

**Description:**

Controls the maximum length in bytes to which the *condor\_replication* daemon log will be allowed to grow. It will grow to the specified length, then be saved to a file with the suffix .old. The .old file is overwritten each time the log is saved, thus the maximum space devoted to logging is twice the maximum length of this log file. A value of 0 specifies that this file may grow without bounds.

**Default:** 1 MB

11) **REPLICATION\_DEBUG** – replication daemon's logging level

**Description:**

Logging level for the *condor\_replication* daemon. See SUBSYS\_DEBUG for values.

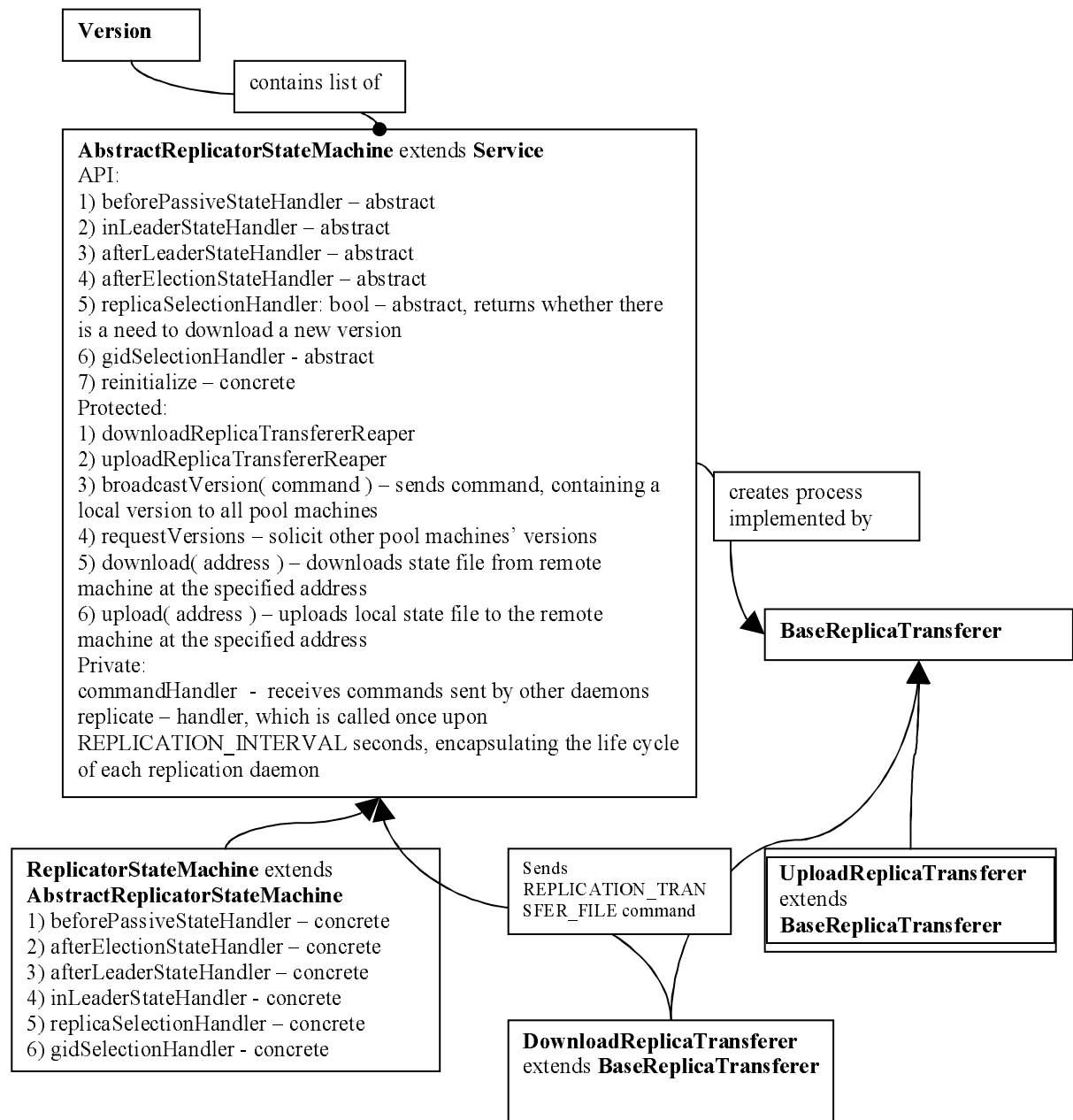
12) **REPLICATION\_LOG** – path to replication daemon's log file

**Description:**

Path to the log file

**Default:** No

# Class diagram



## Classes' structure

### **AbstractReplicatorStateMachine**

- 1) replicationDaemonsList: string list – list of sinful strings of all the replication daemons in the pool except for this very replication daemon
- 2) versionsList: list of Version – list of versions, sent from other replication daemons in the pool; used by 'replicaSelectionHandler' to select the appropriate state file replica and by 'gidSelectionHandler' to merge the state files from several reconciled networks
- 3) myVersion: Version – local state file version
- 4) state: enum – member of {VERSION\_REQUESTING, VERSION\_DOWNLOADING, BACKUP, REPLICATION\_LEADER}
- 5) uploadTransfererPidsList: list of integer – pids of running uploading condor\_transferer processes
- 6) downloadTransfererPid: integer – pid of running downloading condor\_transferer process
- 7) lastHadAliveTime: timestamp – the last time, that local HAD sent HAD\_ALIVE command

### **ReplicatorStateMachine**

- 1) versionRequestingTimerId – timer for passing from VERSION\_REQUESTING to VERSION\_DOWNLOADING state
- 2) versionDownloadingTimerId – timer for passing from VERSION\_DOWNLOADING to BACKUP state
- 3) replicationTimerId – life cycle timer of replication daemon

### **Version**

- 1) gid: integer – group identifier to differentiate between different pools
- 2) logicalClock: integer – ever-growing logical clock, designating how updated the local state files are
- 3) sinfulString: string – sinful string of the machine, to which the version belongs
- 4) stateFile: static string – path to file that is to be replicated amongst pool machines
- 5) lastModifiedTime: static integer – state file last modified time

### **BaseReplicaTransfer**

- 1) socket: ReliSock – TCP socket by means of which the information will be sent or received
- 2) version: Version – the version to be transferred
- 3) stateFile: string – path to file that is to be transferred
- 4) daemonSinfulString: string – address of another communication side

## Dependency matrices

### Influence of version commands on replication daemon in different states

State	Message	NEWLY JOINED	LEADER	GIVING_UP	SOLICIT	SOLICIT_REPLY
REQUESTING		ignore	ignore	ignore	ignore	collect version
DOWNLOAD		ignore	ignore	ignore	ignore	ignore

ING					
BACKUP	ignore(future merge)	download version	ignore(future merge)	send SOLICIT_REPLY	cannot be - assert
LEADER	ignore(future merge)	ignore	gid change(future merge)	send SOLICIT_REPLY	ignore

Influence of notifications from HAD on replication daemon in different states

State	Event	construction	AFTER_ELECTION	IN_LEADER	AFTER_LEADER
REQUESTING		- solicit versions - wait for reply - download best version	ignore	ignore	ignore
DOWNLOADING		N/A	ignore	ignore	ignore
BACKUP		N/A	- gid selection - last HAD alive time set - pass to LEADER	- gid selection - last HAD alive time set - pass to LEADER	ignore
LEADER		N/A	assert	last HAD alive time set	- broadcast version - pass to BACKUP

Influence of timers expiration on replication daemon in different states

State	Timer	version requesting	version downloading	replication
REQUESTING		- cancel timer - pass to DOWNLOADING - start downloading best version - start version downloading timer	N/A	- restart timer
DOWNLOADING		N/A	- cancel timer - pass to	- restart timer

		<b>BACKUP</b>	
<b>BACKUP</b>	N/A	N/A	- restart timer - kill stuck transferers
<b>LEADER</b>	N/A	N/A	- restart timer - kill stuck transferers - broadcast version upon state file update

## Protocol implementation

**Number** : 1

**Description** : replication daemon life cycle

**Flow** :

- 1) The replication daemon resets the replication timer
- 2) Joining machines return
- 3) Backups and leader check whether 'uploadTransfererPidsList' and 'downloadTransfererPid' contain processes that have been running for more than MAX\_TRANSFERER\_LIFETIME seconds and kill them
- 4) Backups return
- 5) Leader checks its state file
- 6) If it is changed, leader calls 'broadcastVersion(REPLICATION\_LEADER\_VERSION)'  
Upon receiving the command:
  - \* leaders and joining machines: ignore it
  - \* backups: see 7)
- 7) Backup checks, whether the received version is better than the local one
- 8) If the latter is true and there is no another downloading 'condor\_transferer' running, backup downloads the state file from the remote machine, to which the received version belongs
- 9) After the downloading is finished, the 'downloadReplicaTransfererReaper' is called:
  - 9.1) It rotates the downloaded temporary files with the local state file and version file (tries several times upon rotate failures)
  - 9.2) It synchronizes the local version
- 10) After broadcasting the version, leader checks whether  $currentTime - lastAliveHadTime > HAD\_ALIVE\_TOLERANCE$
- 11) If the latter is true, leader calls 'broadcastVersion(REPLICATION\_GIVING\_UP\_VERSION)' and passes to BACKUP state

**Number** : 1.1

**Description** : downloading/uploading process, point 8) of UC 1 zoomed in

**Precondition**: backup calls 'download' function

**Flow** :

1) Backup creates downloading 'condor\_transferer' process and passes it the address of remote replication daemon, the temporary state file and version file names

2) Downloading 'condor\_transferer' sends REPLICATION\_TRANSFER\_FILE to remote replication daemon only

Upon receiving the command the replication daemon creates uploading 'condor\_transferer' process and passes it the address of the downloading 'condor\_transferer' process, the state file and version file names

3) Uploading 'condor\_transferer' copies state file (tries several times upon copy failures) and version file to the temporary files and sends them to the downloading 'condor\_transferer', then deletes the temporary files

4) Downloading 'condor\_transferer' receives the state file and version and writes them to the temporary files

**Number** : 2

**Description** : New machine joins the pool

**Precondition**: new machine state is VERSION\_REQUESTING

**Flow** :

1) Newly joined machine calls

'broadcastVersion(REPLICATION\_SOLICIT\_VERSION)'

Upon receiving the command,

\* joining machines: ignore it

\* backups and leader: send

REPLICATION\_SOLICIT\_VERSION\_REPLY to the newly joined machine

Upon receiving the command, joining machine pushes the received version to 'versionsList'

3) Newly joined machine waits

NEWLY\_JOINED\_WAITING\_VERSION\_INTERVAL seconds

4) Newly joined machine passes to VERSION\_DOWNLOADING state

5) Newly joined machine calls 'replicaSelectionHandler' to select the best version in 'versionsList'

6) Newly joined machine downloads the selected version from its remote replication daemon (see UC 1.1)

7) Newly joined machine clears the 'versionsList'

8) Newly joined machines passes to BACKUP state

**Number** : 3

**Description** : backup receives HAD\_AFTER\_ELECTION\_STATE message ('afterElectionHandler')

**Flow** :

1) Backup sets 'lastHadAliveTime' to the current time

2) Backup calls 'gidSelectionHandler'

3) Backup changes its state to REPLICATION\_LEADER

**Number** : 4

**Description** : leader receives HAD\_IN\_LEADER\_STATE message ('inLeaderStateHandler')

**Flow** :

1) Leader sets 'lastHadAliveTime' to the current time

**Number** : 5

**Description** : leader receives HAD\_AFTER\_LEADER\_STATE message ('afterLeaderStateHandler')

**Flow** :

- 1) The giving up leader calls  
'broadcastVersion(REPLICATION\_GIVING\_UP\_VERSION)'  
Upon receiving the command
  - \* joining machine: ignores it
  - \* backup: ignores it (supplied with merging utility in the future, we will merge two state files here)
  - \* leader: chooses new gid for the pool (supplied with merging utility in the future, we will merge two state files here)
- 2) The giving up leader passes to BACKUP state

**Number** : 6

**Description** : our 'replicaSelectionHandler' implementation

**Flow** :

- 1) Depending on the replication daemon state, the following happens
  - \* VERSION\_REQUESTING: cannot be
  - \* VERSION\_DOWNLOADING: see 2)
  - \* BACKUP: see 3)
  - \* REPLICATION\_LEADER: cannot be
- 2) For joining machines the best version is being chosen from 'versionsList' and returns true unless 'myVersion' is the best  
The best version is chosen in the following way:
  - \* If all of the versions are comparable: return the leader's version or the version with the greatest 'logicalClock' value
  - \* Otherwise: amongst the versions with some equal gid return the leader's version or the version with the greatest 'logicalClock' (supplied with merging utility in the future, we will merge several state files here)
- 3) For backups the received remote version is compared vs. local version and 'false' is returned only if local version is greater/equal than the remote version

**Number** : 7

**Description** : our 'gidSelectionHandler' implementation

**Flow** :

- 1) Depending on the replication daemon state, the following happens
  - \* VERSION\_REQUESTING: cannot be
  - \* VERSION\_DOWNLOADING: cannot be
  - \* BACKUP: see 2)
  - \* REPLICATION\_LEADER: see 2)
- 2) For backups and leaders
  - \* If all of the versions (both inside 'myVersion' and 'versionsList') are comparable: do nothing
  - \* Otherwise: set the gid to be some random number, different from the previous one

## **HAD configuration without replication**

To be able to run HAD-enabled pool without replication feature, one must perform the following operations:

- 1) Comment out the following line inside  
\$CONDOR\_WORKSPACE/src/condor\_had/StateMachine.h:  
#define IS\_REPLICATION\_USED

and add the following line

```
#undef IS_REPLICATION_USED
```

and thus disable the replication on compilation level

2) Compile the 'condor\_had' binary from \$CONDOR\_WORKSPACE/src/condor\_had directory:

```
make release
```

3) Remove REPLICATION entry from DAEMON\_LIST and DC\_DAEMON\_LIST inside \$CONDOR\_CONFIG file

Alternatively, one may

1) set the HAD\_USE\_REPLICATION configuration parameter to 'false' and thus disable the replication on configuration level

2) Remove REPLICATION entry from DAEMON\_LIST and DC\_DAEMON\_LIST inside \$CONDOR\_CONFIG file

## Testing

We tested our replication algorithm both in Technion DSL pool (RedHat 8) and in Madison University NMI pool (RedHat 7.2, 8 and 9).

The testing included both manual and automatic tests. In manual tests some basic and important scenarios were tested, such as failing and recovering the primary HAD machine and memory profiling. In automatic tests more randomized scenarios were tested: we simulated jobs submission and machines failures, while checking the correctness of the following invariants all over the test:

Automatic tests:

- Memory profiling (testing that no leaks present)
- Testing in static pool, i.e. no submissions, no failures (ensuring that in normal working mode the new daemon functions properly)
- Testing in dynamic pool, i.e. with submissions and machines failures (stress testing)

Manual tests:

- Testing how the daemon reads the configuration parameters (testing how the input is being read)
- Testing how the system works with replication feature turned off
- Testing how the daemon reacts upon condor\_reconfig command
- Testing if the HAD sends its notification messages to the replication after its state is changed (interaction between HAD and replication)
- Testing that replication leader does send state file to backups when its state file's modification time is changed (testing basic functionality)
- Testing that the sent file and received files are identical
- Testing that the stuck processes are being killed by the replication
- Testing that after successful file transfer no temporary files remain in the state file directory (\$SPOOL by default)
- Testing that all the transferring processes are being reaped properly
- Testing that the version number progresses as the state file changes
- Testing that when a new machine joins the pool, it picks up the best state file version known to the pool (from the leader)
- Testing that after the active HAD is failed, the backups continue with the best version known to them

- Testing that after the primary HAD is failed and raised after some time, it continues with the latest state file of the pool
- Testing that after the primary HAD is failed and raised after some time, the previous active HAD sends message about giving up
- Testing that after the primary HAD is failed and raised after some time, the users sharing information is preserved like before the primary HAD failure
- Testing the the classad is being sent properly, containing all the fields, we wanted
- Testing that the classad is being sent to the collector, when the HAD turns from active to passive or vice versa
- Testing that the file can be sent without MAC (not in a secure manner, like using usual 'put\_file')
- Testing that the file can be sent with MAC and the machine that receives the file indeed ensures that the file's locally generated MAC is equal to the sent one.
- Evaluating how much time takes for the transferers to pass huge files
  
- Overall testing that all the paths in the program are reachable (testing that we enter all the possible ifs and the likes)
- Overall testing that the replication daemons send the right messages one to another after some event happens (such as joining the pool or election of new replication leader etc.)

And various other tests, performed just by tracing the log files and checking their consistency.

<b>Test name</b>	<b>Test description</b>	<b>Result</b>	<b>Fix</b>
Memory profiling	Testing that no leaks present	Success	
Testing in static pool	No submissions, no failures (ensuring that in normal working mode the new daemon functions properly)	Success	
Testing in dynamic pool	With submissions and machines failures (stress testing)	Success	Port was added to the name of HAD classad, since HADs, running on the same machine overwrote ClassAds of each another
Configuration parameters (NEGOTIATOR_STATE_FILE)	Testing default parameter Changing parameter value	Success	
Configuration parameters (REPLICATION_INTERVAL)	Testing default parameter Changing parameter value Trying to input wrong values for parameter (non-numeric or half-numeric)	Success	
Configuration parameters (MAX_TRANSFER_LIFETIME)	Testing default parameter Changing parameter value Trying to input wrong values for parameter	Success	

	(non-numeric or half-numeric)		
Configuration parameters (NEWLY_JOIN, ED_WAITING, VERSION_INTERVAL)	Testing default parameter Changing parameter value Trying to input wrong values for parameter (non-numeric or half-numeric)	Success	
Configuration parameters (HAD_UPDATE_INTERVAL)	Testing default parameter Changing parameter value Trying to input wrong values for parameter (non-numeric or half-numeric)	Bug in 'param_integer' function of Condor: assigning '30w' to HAD_UPDATE_INTERVAL in \$CONDOR_CONFIG resulted in assigning 30 to the respective software data member	Added manual processing of HAD_UPDATE_INTERVAL
Configuration parameters (HAD_USE_REPLICATION)	Testing default parameter Testing how instance works without replication Trying to input wrong values for parameter (not equal to 'true' or 'false')	Success	
Condor_reconfig	Testing how the daemon reacts upon condor_reconfig command	Success, considering that 'condor_reconfig' works in 'condor_restart' semantics for HAD and replication daemons	
Interaction between HAD and replication daemons	Testing if the HAD sends its notification messages to the replication after its state is changed	Success for all the messages	
Testing basic functionality	Testing that replication leader does send state file to backups when its state file's modification time is changed	Success	
Data integrity	Testing that the sent file and received files are identical	Success	
Neat processes managing	Testing that the stuck processes are being killed by the replication	Success for uploading transferer; failed in assert for downloading transferer	The assert wrongly assumed that in reaper the process metadata is valid; in fact, when the stuck process is killed, its metadata is cleaned before the reaper; commented the assert out
Neat file transfer	Testing that after	Success	

	successful file transfer no temporary files remain in the state file directory		
Neat file transfer	Testing that after unsuccessful file transfer no temporary files remain in the state file directory (by stalling the uploading/downloading processes)	Success	
Reapers	Testing that all the transferring processes are being reaped properly: processes metadata is erased by reapers	Success	
Version file	Testing that the version number progresses as the replication leader state file changes	Success	
Joining machine	Testing that when a new machine joins the pool, it picks up the best state file version known to the pool (from the leader)	Success	
Active HAD failure	Testing that after the active HAD is failed, the backups continue with the best version known to them	Success	
Raising of failed HAD	Testing that after the primary HAD is failed and raised after some time, it continues with the latest state file of the pool	Success	
Preserving right information	Testing that after the primary HAD is failed and raised after some time, the users sharing information is preserved like before the primary HAD failure (using 'condor_userprio')	Success	
Giving up message	Testing that after the primary HAD is failed and raised after some time, the previous active HAD sends message about giving up	Success	
Classads	Testing the the classad	Partial success	Initialization of

	is being sent properly, containing all the fields, we wanted		HadIsActive field did not match the format
Classads	Testing that the classad is being sent to the collector, when the HAD turns from active to passive or vice versa	Success	
Integrity	Testing that the file can be sent without MAC (not in a secure manner, like using usual 'put_file')	Success	
Integrity	Testing that the file can be sent with MAC and the machine that receives the file indeed ensures that the file's locally generated MAC is equal to the sent one	Success	
Stress testing with huge state files	Evaluating how much time takes for the transferers to pass huge files	With timeout = 10 sec, it is possible to send the state file of ~20-25 MB size with two backups; After killing stuck transferer the temporary files are not erased	Socket timeout was relatively small, we need it because the file copy may take some time; increased the socket waiting time to INT_MAX
Code reachability	Overall testing that all the paths in the program are reachable (testing that we enter all the possible ifs and the likes)	Success	
Messages	Overall testing that the replication daemons send the right messages one to another after some event happens (such as joining the pool or election of new replication leader etc.)	Success	

## To do and future development

- 1) Possible impacts of the CONTROLLER/CONTROLLEE issues in HAD files
- 2) Gid selection algorithm
- 3) Support for merging utility
- 4) Remove REPLICATION\_LIST configuration variable – the replication list may be deduced from HAD\_LIST