MOSIX

Cluster Management System

Administrator's, User's and Programmer's Guides and Manuals

Revised for MOSIX-4.4.3

March 2017

Preface

MOSIX[1] is a cluster management system targeted for distributed computing on x86_64 based Linux clusters and multi-clusters private clouds. MOSIX incorporates dynamic resource discovery and proactive workload distribution. In a MOSIX system, users can run applications by creating multiple processes, then let MOSIX seek resources and automatically migrate processes among nodes to improve the overall performance, without changing the run-time environment of the migrated processes.

Audience

This document provides general information. It is intended for system administrators, users and programmers who are familiar with general Linux concepts. Users of MOSIX are advised to rely on the documents provided with their specific MOSIX distribution.

Main changes in version 4

- The MOSIX 4 distribution is in user-mode and no longer requires a kernel patch.
- MOSIX 4 can run on any Linux kernel version 3.12 or higher; on Linux distributions based on such kernels and on the standard kernel from openSUSE version 13.1 or higher.
- Supports for batch jobs and queuing no longer available. These can be provided by an external package such as SLURM.
- MOSIX Reach the Clouds (MRC) is not included.
- The code was simplified.
- Rarely-used options were removed.
- Updated guides and manuals.

Organization

This document consists of five parts. The first part provides general information about MOSIX, terminology and system requirements. The administrator guide, in the second part, includes chapters about configurations, storage allocation, management of processes and security. The user's guide, in the third part, explains how to run and manage MOSIX programs. The programmer's guide, in the forth part, explains how MOSIX is viewed from the perspective of programmers. The last part includes the MOSIX manuals.

Further information is available at $http://www.MOSIX.org$.

[1] MOSIX® is a registered trademark.

Contents

Part I
General

Chapter 1

Terminology

The following terms are used throughout this document:

Node - a participating computer (physical or virtual), whose unique IP address is configured to be part of a MOSIX cluster or multi-cluster cloud.

Processor - a CPU (Central Processing Unit or a Core): most recent computers have several processors. (Hyper-Threads do not constitute different processors).

Process - a unit of computation that is started by the "fork" (or "vfork") system call and maintains a unique identifier (PID) throughout its life-time (for the purpose of this document, units of computation that are started by the "clone" system call are called "threads" and are not included in this definition).

Program - An instance of running an executable file: a program can result in one or more processes.

Home-node - The node to which a migratable process "belongs": a migratable process sees the world (file-systems, network, other processes, etc.) from the perspective of this node.

Home-cluster - the cluster to which the home-node of a process belongs.

Local process - a process that runs in its home-node.

Guest process - a process whose home-node is elsewhere, but is currently running here (on the node being administered).

Cluster - one or more computers - workstations, servers, blades, multi-core computers, etc. possibly of different speeds and number of processors, called "nodes", that are owned and managed by the same entity (a person, a group of people or a project) in which all the nodes run the same version of MOSIX and are configured to work tightly together. Note that a MOSIX cluster can at times be different than hardware clusters. For example, it can consist of several hardware-clusters or just part of a hardware-cluster.

Multi-cluster private cloud (Multi-cluster) - A collection of clusters whose owners trust each other and wish to share some computational resources among them.

Your - cluster, addresses, nodes, computers, users, etc. that the sysadmin currently administer or configure.

Chapter 2

What MOSIX is and is not

2.1 What MOSIX is

MOSIX is a package that provides load-balancing by migrating processes within clusters and multi-cluster private clouds.

MOSIX is intended primarily for distributed, concurrent computing, such as used for intensive computing.

The main tool employed by MOSIX is **preemptive process migration** - a process may start on one node, then transparently move to other nodes. Process migration is utilized to optimize the overall performance. Migration of a process occurs automatically and transparently, in response to resource availability. It is repeated as necessary (possibly even returning to where it started).

2.1.1 The main cluster features of MOSIX

- Provides a single-system image.
  - Users can login on any node and do not need to know where their programs run.
  - No need to modify or link applications with special libraries.
  - No need to copy files to remote nodes.
- Automatic resource discovery and workload distribution:
  - Load-balancing by process migration.
  - Migrating processes from slower to faster nodes.
  - Migrating processes from nodes that run out of free memory.
- Migratable sockets for direct communication between migrated processes.
- Provides a secure run time environment (sandbox) for guest processes.
- Supports checkpoint and recovery.
- Supports 64-bit x86 architectures.
- Includes tools for automatic installation and configuration.
- Includes an on-line monitor.

2.1.2 Additional multi-cluster cloud features

- Guest processes can move from one cluster to another.
- Clusters can be shared symmetrically or asymmetrically.
- Cluster owner can assign different priorities to guest processes from other clusters.
- Supports disruptive configurations:
  - Clusters can join or leave the multi-cluster cloud at any time.
  - Guest processes move out before disconnecting a cluster.

2.2 What MOSIX is not

- A cluster set-up and installation tool.
- A batch/queuing system. For that, we recommend a different package such as SLURM.
- In the Linux kernel or part thereof.

2.2.1 MOSIX does not

- Improve performance of intensive I/O processes.
- Improve performance of non-computational applications, such as web or mail servers.
- Support high-availability.
- Support all the Linux system-calls (see the "mosrun" manual for details).

Chapter 3

System requirements

All nodes must have the x86_64 (64-bit) architecture.

All cores of the same node must have the same speed.

All the nodes must be connected by a network that supports TCP/IP and UDP/IP. Each node should have a unique IP address in the range 0.1.0.0 to 255.255.254.255 that is accessible to all the other nodes.

TCP/IP ports 252 and 253 and UDP/IP ports 249 and 253 should be reserved for MOSIX (not used by other applications or blocked by a firewall).

The Linux kernel should be of version 3.12 or higher, but also acceptable are the standard OpenSUSE kernels since version 13.1 and any Linux kernel patched by an older MOSIX distribution, version 3.4.0.0 or higher.

While any Intel or AMD processors can be used, specific CPUs have added non-standard features, in particular Intel's SSE4.1 and SSE4.2 standards, which are not present on either older Intel processors or in AMD processors. Newer versions of the "glibc" library (after glibc-2.8) detect those features during process-initialisation and record their presence for later optimisations. Unfortunately, this means that a process which started on a node that has those features will be unable to migrate to nodes that do not have them (if it does, it could crash with an "Illegal Instruction" fault). Accordingly, you should not mix newer Intel processors in the same MOSIX cluster (or multi-cluster) with older Intel processors or with AMD computers. Alternately, you may re-compile the "glibc" library with the "–disable-multi-arch" flag or obtain such a library from your Linux distribution. If your MOSIX cluster(s) are on virtual- machines, you may also circumvent the problem by using VMWare's "Enhanced Vmotion Compatiblity".

MOSIX can be installed on top of any Linux distribution and mixing of different Linux distributions on different nodes is allowed.

Part II
Administrator's Guide

Chapter 4

Configuration

4.1 General

The "mosconf" script will lead you step-by-step through the MOSIX configuration.

Mosconf can be used in two ways:

1. You can configure (or re-configure) MOSIX on the cluster-node where it should run. If so, just press <Enter> at the first question that "mosconf" presents.
2. In clusters (or parts of clusters) that have a central repository of system-files, containing their root image(s), you can make changes in the central repository instead of having to manually update each node separately.

   This repository can for example be NFS-mounted by the cluster as the root file-system, or it can be copied to the cluster at boot time, or perhaps you have some cluster-installation package that uses other methods to reflect those files to the cluster. Whichever method is used, you must have a directory on one of your servers, where you can find the hierarchy of system-files for the clusters (in it you should find subdirectories such as /etc, /bin, /sbin, /usr, "/lib", "/mnt", "/proc" and so on).

   At the first question of "mosconf", enter the full pathname to this repository.

When modifying the configuration there is no need to stop MOSIX - most changes will take effect within a minute. The only exception is that if you are modifying any of the following:

- The list of nodes in the cluster (/etc/mosix/mosix.map).
- The IP address used for MOSIX (/etc/mosix/mosip) for a cluster (option 2 above).

then you must run "mossetpe" on all the cluster-nodes (or restart MOSIX, whichever is convenient).

The MOSIX configuration is maintained in the directory "etc/mosix".

4.2 Configuring the single cluster

4.2.1 Participating nodes

The most important configuration task is to inform MOSIX which nodes participate in your cluster. In "mosconf" you do this by selecting "Which nodes are in this cluster".

Nodes are identified by their IP address (see the advanced options below if they have more than one): commonly the nodes in a cluster have consecutive IP addresses, so it is easy to define them using the IP address of the first node followed by the number of nodes in the range, for example, if you have 10 nodes starting from 192.168.3.1 to 192.168.3.10, type "192.168.3.1" followed by "10". If there are several such ranges, you need to specify all of them and if there are nodes with an isolated IP address, you need to specify them as ranges of 1.

If your IP addresses are mostly consecutive, but there are a few "holes" due to some missing computers, it is not a big deal - you can still specify the full range, including the missing computers (so long as the IP addresses of the "holes" do not belong to other computers elsewhere).

Specifying too many nodes that do not actually exist (or are down) has been known to produce excessive ARP broadcasts on some networks due to attempts to contact the missing nodes. This was found to be due to a bug in some routers, but unfortunately many routers have this bug.

It is always possible to add or delete nodes without stopping MOSIX: if you do it from a central repository, you need to run "mossetpe" on all your cluster nodes for the changes to take effect.

4.2.2 Automatic configuration of the participating nodes

MOSIX provides a tool that can automatically detect and configure the participating nodes in most clusters. You can invoke this tool either within "mosconf" by selecting 'a' under "Which nodes are in this cluster", or by running "mos_autoconf". This will detect all the nodes on the local TCP/IP subnet which run the same or a very close version of MOSIX (with the same first two digits of the version number), then configure them all accordingly.

The following conditions apply:

1. You should be able to identify an IP subnet (no larger than a Class B) that contains all the nodes which you intend to have in your cluster, but no others (such as nodes in your multi-cluster).

2. All the above nodes must be up and running.

3. An identical MOSIX protection key (password) must be already configured on all the above nodes (but NOT on other nodes that do not belong to this cluster).

4. Huge clusters (about 30000 nodes and over) cannot be detected using this tool.

5. The advanced options (below) are not automatically configured - if you require those options, then this tool is not recommended because you will need to manually re-configure them afterwards on all nodes.

This tool will interactively ask the relevant questions about the cluster and can also be used to automatically configure the cluster's logical node numbers.

4.2.3 Advanced options

The following are advanced options (if no advanced options were previously configured, type "+" in "mosconf"). As above, it is not necessary to stop MOSIX for modifying advanced options, just run "mossetpe" after making the changes from a central repository.

Nearby or distant nodes

To optimize process migration, for each range of nodes, you can define whether they are "distant" or "near" the nodes that you are configuring. The reason is that when networking is slow, it is better to compress the memory image of migrating processes: it takes CPU time, but saves on network transfer time and volume. If however the nodes are near, it is better not to compress. As a general guideline, specify "distant" if the network is slower than 1GB/sec, or is 1GB/sec and the nodes are in different buildings, or if the nodes are several kilometers away.

Aliases

Some nodes may be connected to more than one network, thus have several IP addresses. Messages from another node could thus seem to arrive from an IP address other than the one used to send messages to that node. Aliases allow MOSIX to identify different IP addresses through which valid MOSIX messages could arrive and associate them with one of its configured nodes.

For example, two physical clusters can each be internally connected by a faster Infiniband link, but only linked between them by Ethernet. To use the Infiniband, configure nodes within the physical cluster, INCLUDING THE LOCAL NODE with their Infiniband address and outside the physical cluster with their Ethernet address. As nodes from the other cluster would identify themselves using an unreachable Infiniband address, these addresses must be aliased to the respective Ethernet IP addresses of the other cluster.

Another example is of a junction node which apart from being part of a logical MOSIX cluster also serves as a router between physical clusters. As the junction node can have only one MOSIX IP address, which belongs (at most) to only one of the clusters, the other cluster(s) must use an alias to identify the junction node as it appears on their own network.

Unusual circumstances with IP addresses

There are rare cases when the IP address of a node does not appear in the output of "ifconfig" and even more rare cases when more than one IP address that belongs to a node is configured as part of the MOSIX cluster AND appears in the output of "ifconfig" (for example, a node with two Network-Interface-Cards sometimes boots with one, sometimes with the other and sometimes with both, so MOSIX has both addresses configured "just in case"). When this happens, you need to manually configure the main MOSIX address (using "Miscellaneous policies" of "mosconf").

4.3 Configuring the multi-cluster cloud

4.3.1 Partner-clusters

Now is the time to inform MOSIX which other clusters (if any) are part of your MOSIX multi-cluster private cloud.

In a MOSIX multi-cluster, there is no need for each cluster to be aware of all the other clusters, but only of those partner-clusters that we want to send processes to or are willing to accept processes from.

You should identify each partner-cluster with a name: usually just one word (if you need to use more, do not use spaces, but '-' or '_' to separate the words). Note that this name is for your own use and does not need to be identical across the multi-cluster. Next you can add a longer description (in a few words), for better identification.

4.3.2 Which nodes are in a partner-cluster

Nodes of partner-clusters are defined by ranges of IP addresses, just like in the local cluster - see above. As above, a few “holes” are acceptable.

For each range of nodes that you define, you will be asked: ”Are these nodes distant [Y/n]?” (or ”Is this node distant [Y/n]?” in the case of a single node). ”nearby” and ”distant” are defined in section 4.2.2.A above, but unlike the local cluster, the default here is ”distant”.

4.3.3 Partner-cluster relationship

By default, migration can occur in both directions: local processes are allowed to migrate to partner-clusters and processes from partners-clusters are allowed to migrate to the local cluster (subject to priorities, see below). As an option, you can allow migration only in one direction (or even disallow migration altogether if all you want is to be able to view the load and status of the other cluster).

4.3.4 Priorities

Each cluster is given a priority: this is a number between 0 and 65535 (0 is not recommended as it is the local cluster’s own priority) - the lower it is, the higher the priority. When one or more processes originating from the local cluster, or from partner-clusters of higher priority (lower number), wish to run on a node from our cluster, all processes originating from clusters of a lower priority (higher number) are immediately moved out (evacuated) from this node (often, but not always, back to their home cluster). When you define a new partner-cluster, the default priority is 50.

4.3.5 Priority stabilization

The following option is suitable for situations where the local node is normally occupied with privileged processes (either local processes, processes from your own cluster or processes from more privileged clusters), but repeatedly becomes idle for short periods.

If you know that this is the pattern, you may want to prevent processes from other clusters from arriving during these short gaps when the local node is idle, only to be sent away shortly after. You can define a minimal gap-period (in seconds) once all higher-privileged processes terminated (or left). During that period processes of less-privileged clusters cannot arrive: use “Miscellaneous policies” of “mosconf” to define the length of this period.

4.3.6 Maximum number of guests

While the number of guest processes from your own cluster is unlimited, the maximal number of simultaneous guest-processes from partner-clusters can be limited and you can change that maximum by using “Miscellaneous policies” in “mosconf”.

4.4 Configuring the processor speeds

In the past, MOSIX used to detect the processor’s speed automatically. Due to the immense diversification of processors and their special features which accelerate certain applications, but not others, this is no longer feasible. Even the processor’s frequency-clock is no longer a reliable indicator for the processor’s speed (and it can even be variable).

It is therefore recommended that you set the processor speeds manually, based either on information about your computers, benchmark web-sites, or measuring the performance of actual applications that your users intend to run on the MOSIX cluster(s).

The nodes with your most typical processor should usually have the speed of 10,000, which is also the default speed set by MOSIX, whereas faster nodes should have proportionally higher speeds and slower nodes should have proportionally lower speeds. However, if you add new computers to your cluster(s), you are not required to adjust the speeds of existing nodes - you may simply configure the new computers relative to the ones you had before.

4.5 Configuring the freezing policies

4.5.1 Overview

When too many processes are running on their home node, the risk is that memory will be exhausted, the processes will be swapped out and performance will decline drastically. In the worst case, swap-space may also exhausted and then the Linux kernel will start killing processes. This scenario can happen for many reasons, but the most common one is when another cluster shuts down, forcing a large number of processes to return home simultaneously. The MOSIX solution is to freeze such returning processes (and others), so they do not consume precious memory, then restart them again later when more resources become available.

Below we discuss how to set up the policy for automatic freezing to handle different scenarios of process-flooding. This policy does not affect:
A) Processes that are manually frozen ("mosmigrate pid freeze").
B) Processes which the user deemed "unfreezable" ("mosrun -g").
C) Guest processes: they migrate out instead when the load is high.

4.5.2 Freezing-policy details

In this section, the term "load" refers to the local node.

The policy consists of:

- The "Red-Mark": when the load reaches above this level, processes will start to be frozen until the load drops below this mark.
- The "Blue-Mark": when the load drops below this level, processes start to un-freeze. Obviously the "Blue-Mark" must be significantly less than the "Red-Mark".
- "Home-Mark": when the load is at this level or above and processes are evacuated from other clusters back to their home-node, they are frozen on arrival (without consuming a significant amount of memory while migrating).
- "Cluster-Mark": when the load is at this level or above and processes from this home-node are evacuated from other clusters back to this cluster, they are instead brought frozen to their home-node.
- Whether the load for the above 4 load marks ("Red", "Blue", "Home", "Cluster") is expressed in units of processes or in standardized MOSIX load: The number of processes is more natural and easier to understand, but the MOSIX load is more accurate and takes into account the number and speed of the processors: roughly, a MOSIX load unit is the number of processes divided by the number of processors (CPUs) and by their speed

relative to a "standard" processor. Using the MOSIX standardized load is recommended in clusters with nodes of different types - if all nodes in the cluster have about the same speed and the same number of processors/cores, then it is recommended to use the number of processes.

- Whether to keep a given, small number of processes running (not frozen) at any time despite the load.
- Whether to allow only a maximum number of processes to run (that run on their home-node - not counting migrated processes), freezing any excess processes even when the load is low.
- Time-slice for switching between frozen processes: whenever some processes are frozen and others are not, MOSIX rotates the processes by allowing running processes a given number of minutes to run, then freezing them to allow another process to run instead.
- Policy for killing processes that failed to freeze, expressed as memory-size in MegaBytes: in the event that freezing fails (due to insufficient disk-space), processes that require less memory are kept alive (and in memory) while process requiring the given amount of memory or more, are killed. Setting this value to 0, causes all processes to be killed when freezing fails. Setting it to a very high value (like 1000000 MegaBytes) keeps all processes alive.

When defining a freezing policy, the default is:

RED-MARK	= 6.0 MOSIX standardized load units
BLUE-MARK	= 4.0 MOSIX standardized load units
HOME-MARK	= 0.0 (e.g., always freeze evacuated processes)
CLUSTER-MARK	= -1.0 (e.g., never freeze evacuated processes)
MINIMUM-UNFROZEN	= 1 (process)
MAXIMUM-RUNNING	= unlimited
TIME-SLICE	= 20 minutes
KILLING-POLICY	= always

4.5.3 Disk-space for freezing

Next, you need inform MOSIX where to store the memory-image of frozen processes, which is configured as directory-name(s): the exact directory name is not so important (because the memory-image files are unlinked as soon as they are created), except that it specifies particular disk partition(s).

The default is that all freeze-image files are created in the directory (or symbolic-link) "/freeze" (please make sure that it exists, or freezing will always fail). Instead, you can select a different directory(/disk-partition) or up to 10 different directories.

If you have more than one physical disk, specifying directories on different disks can help speeding up freezing by writing the memory-image of different processes in parallel to different disks. This can be important when many large processes arrive simultaneously (such as from other clusters that are being shut-down).

You can also specify a "probability" per directory (e.g., per disk): This defines the relative chance that a freezing process will use that directory for freezing. The default probability is 1 (unlike in statistics, probabilities do not need to add up to 1.0 or to any particular value).

When freezing to a particular directory (e.g., disk-partition) fails (due to insufficient space), MOSIX will try to use the other freezing directories instead, thus freezing fails only when all directories are full. You can specify a directory with probability 0, which means that it will be used only as a last resort (it is useful when you have faster and slower disks).

4.5.4 Ownership of freezing-files

Freezing memory-image files are usually created with Super-User ("root") privileges. If you do your freezing via NFS (it is slow, but sometimes you simply do not have a local disk), some NFS servers do not allow access to "root": if so, you can select a different user-name, so that memory-image files will be created under its privileges.

4.6 Configuring parameters of 'mosrun'

Some system-administrators prefer to limit what their users can do, or at least to set some defaults for their less technically-inclined users. You can control some of the options of "mosrun" by using the "Parameters of 'mosrun'" option of "mosconf". The parameters you can control are:

1. Selection of the best node to start on: either let the user decide; make "mosrun -b" the default when no other location parameter is specified; or force the "mosrun -b" option on all ordinary users.
2. Handling of unsupported system-calls: either leave the default of killing the process if an unsupported system-call is encountered (unless the user specifies "mosrun -e" or "mosrun -w"; making "mosrun -e" the default; or making "mosrun -w" the default.
3. Whether or not to make the "mosrun -m{mb}" parameter mandatory: this may burden users, but it can help protecting your computers against memory/swap exhaustion and even loss or processes as a result.

To begin with, no defaults or enforcement are active when MOSIX is shipped.

Chapter 5

Storage allocation

5.1 Swap space

As on a single computer, you are responsible to make sure that there is sufficient swap-space to accommodate the memory demands of all the processes of your users: the fact that processes can migrate does not preclude the possibility of them arriving at times back to their home-node for a variety of reasons: please consider the worst-case and have sufficient swap-space for all of them.

You do not need to take into account programs having their home-node elsewhere in your cluster.

5.2 MOSIX files

During the course of its operation, MOSIX creates and maintains a number of small files in the directory "etc/mosix/var". When there is no disk-space to create those files, MOSIX operation (especially load-balancing and queuing) will be disrupted.

When MOSIX is installed for the first time (or when upgrading from an older MOSIX version that had no "etc/mosix/var"), you are asked whether you prefer "etc/mosix/var" to be a regular directory or a symbolic link to "var/mosix". However, you can change it later.

Normally the disk-space in the root partition is never exhausted, so it is best to let "etc/mosix/var" be a regular directory, but some disk-less cluster installations do not allow modifications within "etc": if this is the case, then "etc/mosix/var" should be a symbolic link to a directory on another partition which is writable and have the least chance of becoming full. This directory should be owned by "root", with "chmod 755" permissions and contain a sub-directory "multi/".

5.3 Freezing space

MOSIX processes can be temporarily frozen for a variety of reasons: it could be manually using the command: "migrate {pid} freeze" (which as the Super-User you can also use to freeze any user's processes), or automatically as the load increases, or when evacuated from another cluster. In particular, when another cluster(s) shuts down, many processes can be evacuated back home and frozen simultaneously.

Frozen processes keep their memory-contents on disk, so they can release their main-memory image. By default, if a process fails to write its memory- contents to disk because there is

insufficient space, that process is killed: this is done in order to save the system from filling up the memory and swap-space, which causes Linux to either be deadlocked or start killing processes at random.

As the system-administrator, you want to keep the killing of frozen processes only as the last resort: use either or both of the following two methods to achieve that:

1. Allocate freezing directory(s) on disk partitions with sufficient free disk-space: freezing is by default to the "freeze" directory (or symbolic-link), but you can re-configure it to any number of freezing directories.
2. Configure the freezing policy so that processes are not killed when freeze-space is unavailable unless their memory-size is extremely big (specify that threshold in MegaBytes - a value such as 1000000MB would prevent killing altogether)

5.4 Private-file space

MOSIX users have the option of creating private files that migrate with their processes. If the files are small (up to 10MB per process) they are kept in memory - otherwise they require backing storage on disk and as the system-administrator it is your responsibility to allocate sufficient disk-space for that.

You can set up to 3 different directories (therefore up to 3 disk partitions) for the private files of local processes; guest processes from the same cluster; and guest processes from other clusters. For each of those you can also define a per-process quota.

When a guest process fails to find disk-space for its private files, it will transparently migrate back to its home-node, where it is more likely to find the needed space; but when a local process fails to find disk-space, it has nowhere else to go, so its "write()" system-call will fail, which is likely to disrupt the program.

Efforts should therefore be made to protect local processes from the risk of finding that all the disk-space for their private files was already taken by others: the best way to do it is to allocate a separate partition at least for local processes (by default, space for private files is allocated in "/private" for both local and guest processes).

For the same reason, local processes should usually be given higher quotas than guest processes (the default quotas are 5GB for local processes, 2GB for guests from the cluster and 1GB for guests from other clusters).

Chapter 6

Managing processes

As the system administrator you can make use of the following tools:

6.1 Monitoring (mosmon)

mosmon ("man mosmon"): monitor the load, memory-use and other parameters of your MOSIX cluster or even the whole multi-cluster cloud.

6.2 Listing MOSIX processes (mosps)

mosps ("man mosps"): view information about current MOSIX processes. In particular, **"mosps a"** shows all users, and **"mosps -V"** shows guest processes. Please avoid using "ps" because each MOSIX process has a shadow son process that "ps" will show, but you should only access the parent, as shown by "mosps".

6.3 Controlling running processes (mosmigrate)

mosmigrate ("man mosmigrate"): you can manually migrate the processes of all users - send them away; bring them back home; move them to other nodes; freeze; or un-freeze (continue) them, overriding the MOSIX system decisions as well as the placement preferences of users. Even though as the Super-User you can technically do so, you should never kill (signal) guest processes. Instead, if you find guest processes that you don't want running on one of your nodes, you can use "mosmigrate" to send them away (to their home-node or to any other node).

6.4 Controlling the MOSIX node (mosctl)

mosctl ("man mosctl"): this utility provides a variety of functions. The most important are:

"mosctl stay" - prevent automatic migration away from this node.
("mosctl nostay" to undo).

"mosctl lstay" - prevent automatic migration of local processes away from this node.
("mosctl nolstay" to undoa.)

"mosctl block" - do not allow further migrations into this node.
("mosctl noblock" to undo).

"mosctl expel" - send away all guest processes. You would usually combine it with using "mosctl block" first.

"mosctl bring" - bring back all processes from this home-node. You would usually combine it with using "mosctl lstay" first.

"mosctl isolate" - isolate the node from the multi-cluster (but not from its cluster). ("mosctl rejoin" to undo).

"mosctl cngpri {partner} {newpri}" - modify the guest-priority of another cluster in the multi-cluster (the lower the better).

"mosctl shutdown" - isolate the node from all others (but first expel all guest processes and bring back all processes from this home-node).

"mosctl localstatus" - check the health of MOSIX on this node.

6.5 If you wish to limit what users can run

Some installations want to restrict access to "mosrun" or force its users to comply with a local policy by using (or not using) some of mosrun's options. For example:

- Force users to specify how much memory their program needs.
- Limit the number of "mosrun" processes that a user can run simultaneously (or per day).
- Log all calls to "mosrun" by certain users.
- Limit certain users to run only in their local cluster, but not in their multi-cluster.

etc.

Here is a technique that you can use to achieve this:

1. Allocate a special (preferably new) user-group for mosrun (we shall call it "mos" in this example).
2. Run: "chgrp mos /bin/mosrun"
3. Run: "chmod 4750 /bin/mosrun"

   (steps 2 and 3 must be repeated every time you upgrade MOSIX)
4. Write a wrapper program (we shall call it "bin/wrapper" in this example), which receives the same parameters as "mosrun", checks and/or modifies its parameters according to your desired local policies, then executes:

   "bin/mosrun -p {mosrun-parametrs}".

   Below is the "C" code of a primitive wrapper prototype that passes its arguments to "mosrun" without modifications:

```
#include <malloc.h>
main(int na, char *argv[])
{
        char **newargs = malloc((na + 2) * sizeof(char *));
        int i;
        newargs[0] = "mosrun";
        newargs[1] = "-p";
        for(i = 1 ; i < na ; i++)
                newargs[i+1] = argv[i];
        newargs[i+1] = (char *)0;
        execv("/bin/mosrun", newargs);
}
```

5. chgrp mos /bin/wrapper
6. chmod 2755 /bin/wrapper
7. Tell your users to use "wrapper" (or any other name you choose) instead of "mosrun".

Chapter 7

Security

7.1 Abuse by gaining control of a node

A hacker that gains Super-User access on any node of any cluster could intentionally use MOSIX to gain control of the rest of the cluster and multi-cluster private cloud. Therefore, before joining into a MOSIX multi-cluster private cloud, trust needs to be established among the owners (Super-Users) of all clusters involved (but not necessarily among ordinary users). In particular, system- administrators within a MOSIX multi-cluster need to trust that all the other system-administrators have their computers well protected against theft of Super-User rights.

7.2 Abuse by connecting hostile computers

Another risk is of hostile computers gaining physical access to the internal cluster's network and masquerading the IP address of a friendly computer, thus pretending to be part of the MOSIX cluster/multi-cluster. Normally within a hardware cluster, as well as within a well-secured organization, the networking hardware (switches and routers) prevents this, but you should especially watch out for exposed Ethernet sockets (or wireless connections) where unauthorized users can plug their laptop computers into the internal network. Obviously, you must trust that the other system-administrators in your multi-cluster private cloud maintain a similar level of protection from such attacks.

7.3 Multi-cluster password

Part of configuring MOSIX ("Authentication" of "mosconf") is selecting a multi-cluster-protection key (password), which is shared by the entire multi-cluster cloud. Please make this key highly-secure - a competent hacker that obtains it can gain control over your computers and thereby the entire multi-cluster private cloud.

7.4 Keep the multi-cluster within the same organization

The above level of security is usually only achievable within the same organization, hence we use the term "multi-cluster private cloud", but if it can exist between different organizations, then from a technical point of view, nothing else prevents them from sharing a MOSIX multi-cluster.

Chapter 8

Using MOSIX with SLURM

8.1 Installation guide

Here is a prototype method of combining SLURM with MOSIX - feel free to adjust and modify the scripts in this directory to suit your specific needs.

1. Select a directory that is accessible to the whole cluster and writable only to SlurmUser. Any directory name will do. Below, it is assumed that you selected "usr/local/slurm/scripts".

   In this directory install the following files, with execute permission:

   gmprolog, mprolog, mepilog

2. Create the file "usr/local/slurm/scripts/slurm_mosix.conf", with the following lines:

   COMMONDIR=directory A full pathname to a directory, writable by SlurmUser but only readable to other users on all nodes (this will be used as a scratch directory and will only contain small files).

   SLURMBIN=directory The full directory path where the SLURM binaries live, (usually COMMONDIR=/usr/local/slurm/bin)

3. Create the file "usr/local/slurm/scripts/world.conf" Each line in this file should have a host-name, optionally followed by the number of hosts with sequential IP addresses.

   The host-name should preferably be an IP address (when possible), as this will save the prologs time, calling "gethostbyname()", so jobs can start quicker.

   Host-names should include all the nodes where MOSIX is installed within the cluster(s) managed by the local SLURM controller, but are not limited to that and may also include nodes that are not configured by SLURM at all, or are managed independently by other SLURM controllers (e.g., of other departments).

4. Install MOSIX on your cluster.

5. Configure "slurm.conf":

   ```
   PrologSlurmctld=/usr/local/slurm/scripts/gmprolog
   Prolog=/usr/local/slurm/scripts/mprolog
   Epliog=/usr/local/slurm/scripts/mepilog
   ```

Then add to all lines that describe a MOSIX-enabled node or a range of MOSIX-enabled nodes the following:
`Feature=mosix`

6. Run "scontrol reconfigure".

7. Instruct your users:

   - In the allocation phase ("srun", "sbatch" or "salloc"), use the flags "O -Cmosix × MHN", where "MHN" is the desired number of MOSIX home-nodes for the job. Note that since SLURM cannot assign more than 128 tasks per node, even with over-subscribing, a sufficient number of MOSIX home-nodes should be specified.
   - All commands should be preceded with "mosrun [mosrun-parameters]". The mosrun-parameters should normally include "b" and "mmem".
   - No memory-requirements should be specified in "srun", because SLURM does not over-commit memory.

   If users find the above too hard, then you may prefer to write small wrapper script(s) for their convenience.

Part III
User's Guide

Chapter 9

What can MOSIX do for me

MOSIX allows your programs to be migrated within a computer-cluster, or even within a multi-cluster private cloud (as configured by your system-administrator). Each of your processes can migrate either automatically (in order to achieve load-balancing and similar optimizations), or by your manual request. No matter how many times a process migrates, it continues to behave as if it was still running where you launched it (this means for example that it will only use the files on that computer).

MOSIX is best for computational programs and becomes less effective when your programs are Input/Output oriented or perform system-calls.

Some programs are not suitable for MOSIX, including programs that use shared-memory or threads (threads use shared-memory) and some system/administration-programs. The full list of unsupported features is listen under the "LIMITATIONS" sections of the "mosrun" manual. You can also read it using:

```
man mosrun | sed -e '1,/^LIMITATIONS/d' -e '/^SEE ALSO/,$d' | more
```

If you are not sure whether your program can run under MOSIX, use "mosrun -w" (see below) and check the error messages.

Chapter 10

Running MOSIX programs

10.1 The basics

MOSIX allows your programs to be migrated within a computer-cluster, or even within a multi-cluster cloud. A process can migrate either automatically such as to achieve load-balancing, or by manual user requests. No matter how many times a process migrates, it continues to transparently behave as if it was running on its home-node, which is usually where it was launched. For example, it only uses the files, sockets and similar resources of its home-node.

To run a MOSIX program, use:

```
mosrun [flags] {program} [arguments]
```

Following are the basic flags to use:

-b
: Start your program on the best available node (computer). If unsure, use it!

-m {mb}
: Tell MOSIX how much memory your program may require, in Megabytes. If the amount changes during execution, use the maximum. Use this option when you can, also as a courtesy to other users.

-e OR -w
: If "mosrun" complains about unsupported functions, then these can help relax some restrictions so you can run your program anyway. In almost all cases, if your program runs, it will still run correctly. Running the C-shell ("csh") for example, requires this option. The difference between '-e' and '-w' is that '-w' also logs all occurrences of unsupported functions to the standard-error and is therefore useful for debugging, whereas '-e' is silent. -S{maxprograms}
If you want to run a large set of programs, you should create your own private queue, using:

```
mosrun -S{maxprograms} [other-flags] {commands-file}
```

where {commands-file} contains one command per line. "mosrun" will then run {maxprograms} commands at any given time until all programs complete.

Example 1: Running a program that is estimated to require 400MB of memory at most:

```
% mosrun -b -m400 simple_program
```

Example 2: Running a large set of programs. Each program requires a memory of 500MB, the private queue is used to limit the maximum number of simultaneous programs to 200.

```
% cat my_script
      my_program -x -y -z < /home/myself/file1 > /home/myself/output1
      my_program -x -y -z < /home/myself/file2 > /home/myself/output2
      my_program -x -y -z < /home/myself/file3 > /home/myself/output3
      my_program -x -y -z < /home/myself/file4 > /home/myself/output4
      my_program -x -y -z < /home/myself/file5 > /home/myself/output5

% mosrun -b -m500 -e -S200 my_script
```

10.2 Advance options

You can tell "mosrun" where to start your program, using one of the following arguments:

-b	Allow MOSIX to choose the best place to start.
-h	On your own computer.
-r{host}	On the given computer.
-r{IP-address}	On the computer with the given IP-address.
-{number}	On the given node-number (set by the system-administrator).

If you request to start your program on a computer that is down or refuses to run it, the program will fail to start unless you also provide the '-F' flag, in which case the program will then start on the launching node.

The '-L' flag prevents automatic migrations. Your program may still be migrated manually and will still forcefully migrate back to its home-node in certain emergencies. The '-l' flag cancels '-L': this is useful, for example, when "mosrun" is called by a shell-script that already runs under "mosrun -L".

Certain Linux features, such as shared-memory, certain system-calls and certain system-call parameters, are not supported. By default, when such features are encountered, the program is terminated. The '-e' (silent) or '-w' (verbose) options allow the program to continue and instead only fail the unsupported system-calls as they occur. The '-u' option cancels both '-e' and '-w'.

MOSIX processes may be automatically frozen when the load is too high, subject to policies configured by your system-administrator. If you don't want your process to be frozen automatically (for example because it communicates with other processes), use the '-g' option. The '-G' option cancels '-g'.

When using your private queue, if you want to find which programs (if any) failed, you can specify a second file-name:

```
mosrun -S{number} {script-file},{fail-file}
```

Command-lines that failed will then be listed in {fail-file}.

Periodic automatic checkpoints can be produced using the "A{minutes}" argument. By default, checkpoints are saved to files whose names start with "ckpt.{process-ID}": the "C{filename}" argument can be used to select different file-names.

Checkpoint-files have a numeric extension to determine the checkpoint-number, such as "myckpt.1", "myckpt.2", "myckpt.3". The "N{max}" argument can be used to limit the number of checkpoint- files: once that maximum is reached, checkpoint-numbers will start again at 1, so new checkpoints will override the earlier checkpoints.

To resume running from a checkpoint file, use "mosrun -R{checkpoint- file}".

It is also possible to resume a program with different opened-files than the files that were open when the checkpoint was taken - for details, see "man mosrun".

Private temporary-directories can be specified with the '-X' option.

Other advanced options ('-c', '-n', '-d') can affect the automatic migration-considerations of a program (see "man mosrun" for details).

10.3 Spawning Native Linux child programs

Once a program (including a shell) runs under "mosrun", all its child-processes will be migratable as well, but also subject to the limitations of migratable programs (such as being unable to use shared-memory and threading). If your shell, or shell-script, is already running under "mosrun" and you want to run some program as a standard Linux program, NOT under MOSIX, use the command:

```
mosnative {program} [args]...
```

10.4 Using pipes between migrated programs

It is common to run two or more programs in a pipeline, so that the output of the first becomes the input of the second, etc. You can do this using the shell:

```
program1 | program2 | program3 ...
```

If your shell (or shell-script) that generates the pipeline is running under "mosrun" and the amount of data transferred between the programs is large, this operation can be quite slow. Efficiency can be gained by using the MOSIX direct-communication feature. Use:

```
mospipe ‘‘program1 [args1]..." ‘‘program2 [args2]..." program3...
```

"mospipe" can substitute "mosrun", so you do not need to use "mosrun mospipe" and the arguments that inform "mosrun" where to run can be given to "mospipe" instead. Complete details can be found in "man mospipe".

10.5 Error messages

The following group of errors indicate that the program encountered a feature that is not supported by "mosrun":

system-call '{system-call-name}' not supported under MOSIX

Shared memory (MAP_SHARED) not supported under MOSIX

Attaching SYSV shared-memory not supported under MOSIX

Prctl option #{number} not supported under MOSIX

IPC system-call #{number} not supported under MOSIX

Sysfs option '{number}' not supported under MOSIX

Ioctl 0x{hexadecimal-number} not supported under MOSIX

Mapping special character files not supported under MOSIX

getpriority/setpriority supported under MOSIX only for self

If you see any of the above errors you may either:

1. Use "mosrun -e" (or "mosrun -w") to make the program continue anyway (although the unsupported feature will fail)
2. Modify your program so it does not use the unsupported feature.

Other errors include:

kernel does not support full ptrace options - Make sure that the kernel is Linux version 3.12 or higher.

failed allocating memory - There is not enough memory available on this computer: try again later.

illegal system call #{number} - The program attempted to run a system-call with a bad number: there could be a bug in the program, or the MOSIX version is very old and new system-calls were added since.

sysfs detected an unreasonably long file-system name - The size of the buffer provided to the "sysfs()" system-call is unreasonably large (more than 512 bytes - probably a fault in the library).

WARNING: setrlimit(RLIMIT_NOFILE) ignored by MOSIX - MOSIX does not allow programs to change their open-files limit.

File-descriptor #{number} is open (only 1024 files supported under MOSIX) - "mosrun" was called with a open file-descriptor numbered 1024 or higher: this is not supported.

Failed reading memory-maps - Either "/proc" is not mounted, or the kernel is temporarily out of resources.

Failed opening memory file - Either "/proc" is not mounted, or the kernel is temporarily out of resources.

Kernel too secure to run MOSIX (by non-Super-User) - The Linux kernel was compiled with extreme security limitations: ¡ please check the security options in your Linux kernel

failed migrating to {computer}: {reason} - Failed attempt to start the program on the requested computer.
Reasons include:

- "not in map"
  - other computer is not recognized as part of this cluster or multi-cluster.
- "no response"
  - perhaps MOSIX is not running on the requested computer? or perhaps a fire-wall is blocking TCP/IP ports 253 on the requested computer?

- "other node refused"
  - the requested computer was not willing to accept the program.
- "did not complete (no memory there?)"
  - there were probably not enough resources to complete the migration, or perhaps the requested computer just crashed or was powered-off.

{computer} is too busy to receive now - The requested computer refused to run the program.

connection timed out - The other computer stopped responding while preparing to run a batch-job. Perhaps it crashed, or perhaps it runs a different version of MOSIX, or perhaps even a different daemon is listening to TCP/IP port 250.

batch refused by other party ({computer}) - The requested computer refused to run the batch-job from this computer.

Lost communication with {computer} - The TCP/IP connection with the computer that was running the program was severed. Unfortunately this means that the program had to be killed.

Process killed while attempting to migrate from {computer1} to {computer2} - Connection was severed while the program was migrating from one remote computer to another. Unfortunately this means that the program had to be killed.

Un-freeze failed - The program was frozen (usually due to a very high load), then an attempt to un-freeze it failed, probably because there was not enough memory on this computer. Recovery was not possible.

Failed decompressing freeze file - The program was frozen (usually due to a very high load), but there were not enough resources (memory/processes) to complete the operation and recovery was not possible.

Re-freezing because un-freeze failed - The program was frozen (usually due to a very high load), then an attempt to un-freeze it failed, probably because there was not enough memory on this computer. Recovery was possible by re-freezing the program: you may want to manually un-freeze it later when more memory is available.

No space to freeze - The disk-space that was allocated for freezing (usually by the system-administrator) was insufficient and so freezing failed. The MOSIX configuration indicated not to recover in this situation, so the program was killed.

Security Compromised - Please inform this to your system-administrator and ask them to run "mosconf", select the "Authentication" section and set new passwords immediately.

Authentication violation with {computer} - The given computer does not share the same password as this computer: perhaps someone connected a different computer to the network which does not really belong to the cluster? Please inform your system-administrator!

Target node runs an incompatible version - Your computer and the computer on which you want to start your batch job do not run the same (or a compatible) version of MOSIX.

{program} is a 32-bit program - will run in native Linux mode - 32-bit programs are not migratable: your program will run instead as a standard Linux program: consider re-compiling your program for the 64-bit architecture.

remote-site ({computer}) seems to be dead - No "heart-beat" detected from the computer on which your program runs.

Corrupt or improper checkpoint file - Perhaps this is the wrong file, or was tempered with, or the checkpoint was produced by an older version of MOSIX that is no longer compatible.

Could not restart with {filename}: {reason} - Failed to open the checkpoint-file.

File-descriptor {file-number} was not open at the time of checkpoint! - When continuing from a checkpoint, attempted to redirect a file (using "mosrun -R -O", see "man mosrun") that was not open at the time of checkpoint.

Restoration failed: {reason} - Insufficient resources to restart from the checkpoint: try again later.

checkpoint file is compressed - but no /usr/bin/lzop here! - The program "usr/bin/lzop" is missing on this computer (perhaps the checkpoint was taken on another computer?).

WARNING: no write-access in checkpoint directory! - You requested to take checkpoints, but have no permission to create new checkpoint-files in the specified directory (the current-directory by default).

Checkpoint file {filename}: {reason} - Failed to open the checkpoint-file for inspection.

Could not restore file-descriptor {number} with '{filename}': {reason} - When continuing from a checkpoint, the attempt to redirect the given opened-file failed.

Restoration failed - The checkpoint file is probably corrupt.

Line #{line-number} is too long or broken! - The given line in the script-file ("mosrun -S") is either too long or does not end with a line-feed character.

Commands-file changed: failed-commands file is incomplete! - The script-file was modified while "mosrun -S" is running: you should not do that!

Failed writing to failed-commands file! - Due to some write-error, you will not be able to know from {fail-file} which of your commands (if any) failed.

Invalid private-directory name ({name}) - Private-directories (where private-temporary-files live) must start with '/' and not include "..".

Disallowed private-directory name ({name}) - Private-directories (where private-temporary-files live) must not be within "/etc", "/proc", "/sys" or "/dev".

Too many private directories - The maximum is 10.

Private directory name too long - The maximum is 256 characters.

Insufficient disk-space for private files - As your program migrated back to your computer, it was found that it used more private-temporary-file space than allowed. This is usually a configuration problem (has your system- administrator decreased this limit while your program was running on another computer?).

Chapter 11

Operating on your existing programs

11.1 Listing your MOSIX-related processes

The program "mosps" is similar to "ps" and shares many of its parameters. The main differences are that:

1. "mosps" shows only MOSIX and related processes.
2. "mosps" shows relevant MOSIX information such as where your processes are running.
3. "mosps" does not show running statistics such as CPU time and memory-usage (because this information is not readily available for processes that run on remote computers).

The most important information is under the column "WHERE", showing where your processes are running. This can be a node-number, an IP address, the word "here" (if the process is running on your own computer), the word "leaving" (if the process is leaving your computer) or the word "starting" (if the process has just started so it is still unclear where it will run). If you prefer IP addresses, use the '-I' flag; if you prefer the full host-name, use the '-h' flag; and if you prefer just the host-name (without the domain), use the '-M' flag.

The "FRZ" column shows whether your processes are frozen and if so why. The possible reasons are:

"A": Automatic freezing occurred.

"P": An external package requested to freeze the process (it is up to that package to un-freeze it).

"M": The process was frozen manually (by yourself or the system- administrator).

If you run "mosps -N", you can also see the "NMIGS" column, listing how many times your processes have ever migrated.

When running programs with a private queue, you can use "mosps -S" to find out how many programs completed and how many failed so far. Only programs that were started with "mosrun -S" will be shown.

11.2 Finding how long your program was running

Run "mostimeof {pid}" to find out the total user-level running-time that was accumulated by a MOSIX process. {pid} is the process-ID (which can be obtained by "mosps"). Several process-ID's may be specified at once ("mostimeof {pid1} {pid2} ...").

11.3 Migrating your programs manually

Use “mosmigrate {pid} {hostanme or IP-address or node-number}” to migrate a process to the given computer.

Use “mosmigrate {pid} {hostanme or IP-address or node-number}” to migrate a process to the given computer.

Use “mosmigrate {pid} home” to migrate a process back to its home- node.

Use “mosmigrate {pid} freeze” to freeze your process.

Use “mosmigrate {pid} continue” to un-freeze your process.

Use “mosmigrate {pid} checkpoint” to cause your process to generate a checkpoint.

Use “mosmigrate {pid} chkstop” to cause your process to generate a checkpoint and stop with a SIGSTOP signal.

Use “mosmigrate {pid} chkexit” to cause your process to generate a checkpoint and exit.

11.4 Killing your programs

Use “moskillall” to kill all your MOSIX programs (with the SIGTERM signal).

Use “moskillall -{signal}” to send a signal to all your MOSIX processes.

Chapter 12

Using SLURM

To use SLURM users should:

- In the allocation phase ("srun", "sbatch" or "salloc"), use the flags "O -Cmosix*{MHN}", where "MHN" is the desired number of MOSIX home-nodes for the job.

  Note that since SLURM cannot assign more than 128 tasks per node, even with over-subscribing, a sufficient number of MOSIX home-nodes should be specified.

- All commands should be preceded with "mosrun [mosrun-parameters]".

  The mosrun-parameters should normally include "-b" and "-m{mem}".

- No memory-requirements should be specified in "srun", because SLURM does not over-commit memory.

Part IV
Programmer's Guide

Chapter 13

Views from the perspective of programmers

13.1 What types of programs are suitable for MOSIX

MOSIX is suitable for computational programs that run on the x86_64 architecture, where computation is relatively high compared with the volume of Input/Output and Inter-Process-Communication. Such programs can benefit from process-migration.

13.2 What is not supported

All forms of shared memory, except "vfork", are not supported under MOSIX, including threads.

Also, several system-calls that are primarily designed to support threads and others that are primarily designed for system-utilities, are not supported. The full list of unsupported system-calls can be found in the "LIMITATIONS" section of the "mosrun" manual-page.

While the "vfork()" system-call is supported, it forces both the parent and child processes to migrate back to their home-node and process-migration is disabled for them until the son either calls "execve()" or exits.

13.3 How to optimize programs

Due to the need for transparency, where your program works in exactly the same manner regardless where it runs, all system calls are monitored: this means that system-calls are rather expensive in terms of performance.

A few system-calls are not a problem on today's modern computers, but if their number increase as they run within loops, their cost might become prohibitive.

So try to reduce the number of system-calls. You could for example read/write more data in one block instead of reading it byte-by- byte, or you could use "pread/pwrite" instead of lseek+read/write.

A common practice is to use "gettimeofday()" within a loop, often in order to check how efficient your program is - try to minimize this because it is expensive and would make your program inefficient indeed: if you call "gettimeofday()" because you need to do or check something else once in a while, it is better to use "setitimer()" instead.

Note that system-call monitoring occurs even when your program runs on its home-node.

If your program spawns a child-process that runs an executable that does much I/O or many system-calls, consider modifying your "execve()" call so the child runs under "mosnative": "mosnative program [args]..." takes the executable away from being under MOSIX control. On the one hand, the child will have to run on its home-node and not be migratable, but on the other hand it will not incur a performance-penalty on its I/O and system-calls.

13.4 Controlling the behavior of MOSIX from within programs

Some MOSIX behavior can be controlled from within your program. Your program can interact with MOSIX by loading the "open()" system call, "opening" files of the form "proc/self/function" (which do not exist otherwise, so all such calls will fail with errno set to ENOENT if your program does not run on MOSIX).

Here is what you can do:

- Find where your program is currently running.
- Find how many times your program has migrated so far.
- Find whether or not your program runs in your home-cluster.
- Migrate your program to another node of your choice.
- Perform a checkpoint.
- Control automatic-checkpoint parameters.
- Request to receive a signal when your program migrates.
- Inform MOSIX how much memory your program requires.
- Control load-balancing parameters.

You can also control:

- Whether your program may automatically migrate or not.
- Whether your program may be automatically frozen or not.
- Whether MOSIX should ignore unsupported system-calls.
- Whether MOSIX should report unsupported system-calls to stderr.

The details on using those functions is found in the "INTERFACE FOR PROGRAMS" section of the "mosix" manual-page.

Part V
Manuals

Chapter 14

Manuals

The manuals in this chapter are arranged in 3 sets and are provided for general information. Users are advised to rely on the manuals provided with each MOSIX distribution.

14.1 For users

mosbestnode - select the best node to run on.
mosmigrate - manual control of running MOSIX processes.
mosmon - MOSIX monitor.
moskillall - kill or signal all your MOSIX processes.
mospipe - run pipelined processes efficiently using Direct Communication.
mosps - list information about MOSIX processes.
mosrun - running MOSIX programs.
mostestload - MOSIX test program.
mostimeof - report CPU usage of migratable processes.

14.2 For programmers

direct communication - migratable sockets between MOSIX processes.
mosix - sharing the power of clusters and multi-clusters cloud.

14.3 For administrators

mosctl - miscellaneous MOSIX functions.
mossetpe - configure the local cluster.
mos_in_job - Configure MOSIX at the start/end of an external job.

NAME

MOSBESTNODE - Select best node to run on

SYNOPSIS

mosbestnode [-u] [-n] [-w] [-m{mb}]

DESCRIPTION

Mosbestnode selects the best node to run a new MOSIX process on.

Mosbestnode normally prints the selected node's IP address. If the -u argument is used and the node has an associated MOSIX node number, mosbestnode prints its MOSIX node number instead.

The selection is normally for the immediate sending of a job to the selected node by means other than mosrun(1) (such as "rsh", "ssh", or MPI). MOSIX is updated to assume that a new process will soon start on the selected node: when calling mosbestnode for any other purpose (such as information gathering), use the -n argument, to prevent misleading MOSIX as if a new process is about to be started.

The -m{mb} argument narrows the selection to nodes that have at least mb Megabytes of free memory.

When the -w argument is used, mosbestnode waits until an appropriate node is found: otherwise, if no appropriate node is found, mosbestnode prints "0" and exits.

SEE ALSO

mosrun(1), mosix(7).

NAME

MOSMIGRATE - Manual control of running MOSIX processes

SYNOPSIS

mosmigrate {pid} {node-number|IP-address|host}
mosmigrate {pid} home
mosmigrate {pid} out
mosmigrate {pid} freeze
mosmigrate {pid} continue
mosmigrate {pid} checkpoint
mosmigrate {pid} chkstop
mosmigrate {pid} chkexit

DESCRIPTION

Mosmigrate {pid} manually migrates or otherwise instructs a given MOSIX process (pid) to perform an action. Note that mosmigrate does not wait to ensure that the process in question indeed performed the requested action.

The first option, {node-number|IP-address|host}, specifies a target node to which the process should migrate.

The home option forces the process to migrate back to its home-node.

The out option forces a guest process to move out of this node (this option is available only to the Super-User).

The freeze option freezes the process (guest processes may not be frozen).

The continue option unfreezes the process.

The checkpoint option requests the process to take a checkpoint.

The chkstop option requests the process to take a checkpoint and stop (the process may then be resumed with SIGCONT).

The chkexit option requests the process to take a checkpoint and exit.

Except for the Super-User, one can normally migrate or affect only their own processes. The rule is: if you can kill it, then you are also allowed to migrate/affect it.

The best way to locate and find the PID of a MOSIX processes is by using mosps(1).

SEE ALSO

mosrun(1), mosps(1), mosix(7).

NAME

mosmon — MOSIX monitor

SYNOPSIS

mosmon [**-v** | **-V** | **-w**] [**-t**] [**-d**] [**-s** | **-m** | **-m -m** | **-f** | **-c** | **-l**]

DESCRIPTION

Mosmon displays useful current information about MOSIX nodes. The information is represented as a bar-chart, which allows a comparison between different nodes.

The display shows nodes that are assigned node numbers in `/etc/mosix/userview.map` (see `mosix(7)`).

The following options are available:

-w Select horizontal numbering: better display - but less nodes are visible.

-v Select vertical numbering: more nodes are visible - but denser display.

-V Select tight vertical numbering: maximal number of nodes are visible - but even denser display.

-t Display the number of operational nodes and number of CPUs.

-d Display also dead (not responding) nodes.

-s Display the CPU-speeds.

-m Display the used vs. total memory.

-m -m
Display the used vs. total swap space.

-f Display the number of frozen processes.

-c Display the number of cores.

-l Display the load (default).

While in `mosmon`, the following keys may be used:

`v` Select vertical numbering.

`V` Select tight vertical numbering.

`w` Select horizontal numbering.

`a` Select automatic numbering (vertical numbering will be selected if it would make the difference between viewing all nodes or not - otherwise, horizontal numbering is selected).

`s` Display CPU-speeds and number of nodes (if more than 1).
10000 units represent a standard 3GHz processor.

`m` Display used memory vs. total memory: the used memory is displayed as a solid bar while the total memory is extended with '+' signs. The memory is shown in Megabytes.

`m m` (pressing `m` twice) Display used swap-space vs. total swap-space: the used swap-space is displayed as a solid bar while the total swap-space is extended with '+' signs. Swap-space is shown in Gigabytes and is accurate to 0.1GB.

`f` Display the number of frozen processes.

`l` Display loads. The load represents one CPU-bound process that runs on a node with a single CPU under normal conditions. The load increases proportionally on slower CPUs, and decreases proportionally on faster CPUs and on nodes with more than one CPU.

`d` Display also dead (not-responding) nodes.

`D` Stop displaying dead nodes.

`t` Toggle displaying the count of operational nodes.

`Right-Arrow`
Move one node to the right (when not all nodes fit on the screen).

`Left-Arrow`
Move one node to the left (when not all nodes fit on the screen).

`n` Move one screen to the right (when not all nodes fit on one screen).

`p` Move one screen to the left (when not all nodes fit on one screen).

`h` Bring up a help screen.

`Enter` Redraw the screen.

`q` Quit **`Mosmon.`**

SEE ALSO

`mosix`(7).

NAME

MOSKILLALL - Kill or signal all your MOSIX processes

SYNOPSIS

moskillall [-{signum} | -{symbolic_signal}]

Symbolic signals:

HUP INT QUIT ILL TRAP ABRT BUS FPE KILL USR1 SEGV USR2 PIPE
ALRM TERM STKFLT CHLD CONT STOP TSTP TTIN TTOU URG XCPU XFSZ
VTALRM PROF WINCH POLL PWR SYS

DESCRIPTION

Moskillall kills or sends a signal to all the user's MOSIX processes, or if called by the Super-User, to all MOSIX processes of all users.

The default signal is SIGTERM, unless a numeric -{signum} or symbolic -{symbolic_signal} argument specifies a different signal. However, as MOSIX processes should not receive the SIGSTOP signal, SIGSTOP signals are converted to SIGTSTP.

Note that moskillall cannot provide an absolute guarantee that all processes will receive the signal, as a rare race condition is possible where one or more processes are forking at the exact time of running it.

DEFINITION OF MOSIX PROCESSES

MOSIX processes are processes that were started by the mosrun(1) utility on the current node, excluding those processes that invoked the mosnative utility.

SEE ALSO

mosps(1), mosrun(1), mosix(7).

NAME

MOSPIPE - Run pipelined jobs efficiently using Direct Communication

SYNOPSIS

mospipe [mosrun-options1] {program1+args1} [-e] [mosrun-options2]
{program2+args2} [[-e] [mosrun-options3] {program3+args3}]...

Mosrun options:
[-h|-a.b.c.d|-r{hostname}|-{nodenumber}] [-l|-L] [-F]
[-g|-G] [-m{mb}]

DESCRIPTION

Mospipe runs two or more pipelined programs just as a shell would run:

program1 [args1] | program2 [args2] [| program3 [args3]] ...

except that instead of pipes, the connection between the programs uses the MOSIX feature of direct_communication(7) that makes the transfer of data between migrated programs much more efficient.

Each program argument includes the program's space-separated arguments: if you want to have a space (or tab, carriage-return or end-of-line) as part of a program-name or argument, use the backslash(\) character to quote it ('\' can quote any character, including itself).

Each program may optionally be preceded by certain mosrun(1) arguments that control (see mosrun(1) for further details):

-h|-a.b.c.d|-r{hostname}|-{nodenumber}
On which node should the program start.
-F Whether to run even when the designated node is down.
-l|-L Allow/disallow automatic migration.
-G|-g Allow/disallow automatic freezing.
-m{mb} The amount of memory that the program requires.

The -e (or -E) argument between programs indicates that as well as the standard-output, the standard-error of the preceding program should also be sent to the standard-input of the following program.

If mospipe is not already running under mosrun(1), then in order to enable direct communication it places itself under mosrun(1). In that case it also turns on the -e flag of mosrun(1) for the programs it runs.

APPLICABILITY

Mospipe is intended to connect simple utilities and applications that read from their standard-input and write to their standard-output (and standard-error).

mospipe sets the MOSIX direct_communication(7) to resemble pipes, so applications that expect to have pipes or sockets as their standard-input/output/error and specifically applications that only use the stdio(3) library to access those, should rarely have a problem running under mospipe.

However, direct_communication(7) and pipes are not exactly the same, so sophisticated applications that attempt to perform complex operations on file-descriptors 0, 1 and 2 (such as lseek(2), readv(2), writev(2), fcntl(2), ioctl(2), select(2), poll(2), dup(2), dup2(2), fstat(2), etc.) are likely to fail. This regrettably includes the tcsh(1) shell.

The following anomalies should also be noted:

* `mosrun(1)` and `mosnative` (See `mosrun(1)`) cannot run under `mospipe`: attempts to run them will produce a "Too many open files" error.
* An attempt to write 0 bytes to the standard-output/error will create an End-Of-File condition for the next program.
* Input cannot be read by child-processes of the receiver (open direct-communication connections are not inheritable).
* `Direct_Communication(7)` should not be used by the applications (or at least extreme caution must be used) since `direct communication` is already being used by `mospipe` to emulate the pipe(s).

EXAMPLES

```
mospipe "echo hello world" wc
```
is like the shell-command:
```
echo hello world|wc
```
and will produce:
```
   1     2    12
```

```
mospipe "ls /no-such-file" -e "tr [a-z\ ] [A-Z+]"
```
is like the shell-command:
```
ls /no-such-file |& tr '[a-z ]' '[A-Z+]'
```
and will produce:
LS:+/NO-SUCH-FILE:+NO+SUCH+FILE+OR+DIRECTORY

```
b=`mosbestnode`
mospipe "echo hello world" -$b -L bzip2 -$b -L bzcat "tr [a-z] [A-Z]"
```
is like the shell-command:
```
echo hello world | bzip | bzcat | tr '[a-z]' '[A-Z]'
```

It will cause both compression (bzip) and decompression (lzop -d) to run and stay on the same and best node for maximum efficiency, and will produce:

HELLO WORLD

SEE ALSO

`direct_communication`(7), `mosrun`(1), `mosix`(7).

NAME

MOSPS - List information about MOSIX processes

SYNOPSIS

mosps [subset of ps(1) options] [-I] [-h] [-M] [-L] [-O] [-n] [-P] [-V] [-S]

Supported ps(1) options:

1. single-letter options:	TUacefgjlmnptuwx
2. single-letters preceded by '-':	-AGHNTUadefgjlmptuwx

DESCRIPTION

Mosps lists MOSIX processes in "ps" style, emphasizing MOSIX-related information - see the ps(1) manual about the standard options of ps. Since some of the information in the ps(1) manual is irrelevant, mosps does not display the following fields: %CPU, %MEM, ADDR, C, F, PRI, RSS, S, STAT, STIME, SZ, TIME, VSZ, WCHAN.

Instead, mosps can display the following:

WHERE Where is the process running, including the special values:
- 'here' the process runs on this node.
- 'leaving' the process is leaving this node.
- 'starting' the process just started, so it is still unclear where it will run. It also may be waiting for a node with sufficient available memory.

FROM The process' home-node, including the special value 'here'.

ORIGPID the original process ID in the process' home-node. "N/A" when the home-node is here.

FRZ Freezing status:
- \- not frozen
- A automatically frozen
- P preempted (by an external program)
- M manually frozen
- N/A cannot be frozen (including guest processes)

NMIGS the number of times the process (or its MOSIX ancesstors before it was forked) had migrated so far ("N/A" for guest processes).

Normally, if the nodes in the WHERE and FROM fields are listed in /etc/mosix/userview.map, then they are displayed as node-numbers: otherwise as IP addresses. The -I argument forces all nodes to be displayed as IP addresses and the -h argument forces all nodes to be displayed as host-names (when the host-name can be found, otherwise as an IP address). Similarly, the -M argument displays just the first component of the host-names. Regardless of those arguments, the local node is always displayed as "here".

When the -L argument is specified, only local processes (those whose home-node is here) are listed.

When the -O argument is specified, only local processes that are currently running away from home are listed.

The -n argument displays the number of migrations (NMIGS).

The ORIGPID field is displayed only when the -P and/or -V arguments are specified.

When the -V argument is specified, only guest processes (those whose home-node is not here) are listed: the listing includes ORIGPID, but not WHERE and FRZ (as those only apply to local processes).

The -S argument displays the progress of multiple-commands (mosrun -S). Instead of ordinary MOSIX processes. Only the main processes (that read commands-files) are displayed. The information provided is:

TOTAL total number of command-lines given.
DONE number of completed command-lines (including failed commands).
FAIL number of command-lines that failed.

IMPORTANT NOTES

1. In conformance to the ps standards, since guest processes do not have a controlling terminal on this node, in order to list such processes either use the -V option, or include a suitable ps argument such as -A, ax or -ax (it may depend on the version of ps installed on your computer).
2. the c option of ps(1) is useful to view the first 15 characters of the command being run under mosrun instead of seeing only "mosrun" in the command field.

SEE ALSO

ps(1), mosrun(1), moskillall(1), mosix(7).

NAME

MOSRUN - Running MOSIX programs

SYNOPSIS

mosrun [location_options] [program_options] program [args] . . .
mosrun -S{maxjobs} [location_options] [program_options] {commands-file}
[,{failed-file}]
mosrun -R{filename} [-O{fd=filename}[,{fd2=fn2}]]... [location_options]
mosrun -I{filename}
mosenv { same-arguments-as-mosrun }
mosnative program [args]...

Location options:

[-r{host} |-{a.b.c.d} |-{n} |-h |-b |[-L][-l][-G][-g][-F][-A{minutes}]
[-N{max}]

Program Options:

[-m{mb}][-e][-w][-u][-C{filename}][-z][-c][-n][-d {0-10000}]
[-X{/directory}]...

DESCRIPTION

Mosrun runs migratable programs that can potentially migrate to other nodes within a MOSIX(7) cluster (or a multi-cluster private cloud). All child processes of migratable programs can migrate as well, independently of their parent.

Migratable programs act strictly as if they were running on their home-node, from which they were launched. Specifically, all their file and socket operations (opening, reading, writing, sending, receiving) is performed on their home-node, so even when a migrated process is running on another node, it uses the network to perform the operation on the home-node, then returns the results to the program.

Certain Linux features, such as shared-memory, are not available to migratable programs (see the LIMITATIONS section below).

Following are the arguments of mosrun, beginning with where to start the program:

-b	attempt to run on the best available node
-r{hostname}	on the given host
-{a.b.c.d}	on the given IP address
-{n}	on the given MOSIX node-number
-h	in the home-node

(in some installations, the system-adminstrator may force the -b option, disallowing the rest of the above options)

-m{mb}
Specify the maximum amount of memory (in Megabytes) that the program requires, so that programs will not automatically migrate to nodes with insufficient available memory (except to their home-node in circumstances when a program must return home). Also, combined with the -b flag, the program will only consider to start running on nodes with sufficient memory and will not begin until at least one such node is found.

`-e`

Usually, a program that encounters an unsupported feature (see `LIMITATIONS` below) terminates. This flag allows the program to continue and instead behave as follows:

1. `mmap(2)` with (flags & MAP_SHARED) - but !(prot & PROT_WRITE), replaces the MAP_SHARED with MAP_PRIVATE (using MAP_SHARED without PROT_WRITE seems unusual or even faulty, but is unnecessarily used within some Linux libraries).
2. all other unsupported system-call return -1 and "errno" is set to ENOSYS.

`-w`

Same as `-e`, but whereas `-e` is silent, `-w` causes `mosrun` to print an error message to the standard-error whenever an unsupported system-call is encountered.

`-u`

Reverse the effect of `-e` or `-w`: this can be used when a program (especially a shell-script) wants to spawn a son-process that should terminate upon encountering an unsupported feature. Also, as the system-administrator can make the `-e` or `-w` arguments the default, the `-u` flag can be used to reverse their effect.

`-L`

Do not migrate the program automatically. The program may still be migrated manually or when circumstances do not allow it to continue running where it is and force it to migrate back to its home-node.

`-l`

reverse the effect of `-L` and allow automatic migrations.

`-g`

Prevents the program from being automatically frozen (this may be important for interactive programs or programs that must communicate frequently or periodically with other program).

`-G`

Reverse the effect of `-g`, re-allowing the program to be automatically frozen.

`-S{maxjobs}`

Queue a list of jobs (potentially a very long list).

Instead of a program and its arguments, following the list of arguments is a `commands-file`, containing one command per line (interpreted by the standard shell, `bash(1)`): all other arguments will apply to each of the command-lines.

This option is commonly used to run the same program with many different sets of arguments. For example, the contents of `commands-file` could be:

my_program -a1 < ifile1 > output1
my_program -a2 < ifile2 > output2
my_program -a3 < ifile3 > output3

Command-lines are started in the order they appear in `commands-file`. While the number of command-lines is unlimited, `mosrun` will run concurrently up to `maxjobs` (1-30000) command-lines at any given time: when any command-line terminates, a new command-line is started.

Lines should not be terminated by the shell's background ampersand sign ("&"). As `bash` spawns a son-process to run the command when redirection is used, when the number of processes is an issue, it is recommended to prepend the keyword `exec` before each command line that uses redirection. For example:

exec my_program -a1 < ifile1 > output1
exec my_program -a2 < ifile2 > output2
exec my_program -a3 < ifile3 > output3

The exit status of `mosrun -S{maxjobs}` is the number of command-lines that failed (255 if more than 255 command-lines failed).

As a further option, the `commands-file` argument can be followed by a comma and another file-name: `commands-file,failed-commands`. `Mosrun` will then create the second file and write to it the list of all the commands (if any) that failed (this provides an easy way to re-run only those commands that failed).

`-F`
Run the program even if the requested node is unavailable (otherwise, an error-message will be displayed and the program will not start).

`-z`
The program's arguments begin at argument #0 (usually the arguments, if any, begin at argument #1 and argument #0 is assumed to be identical to the program-name).

`-C{filename}`
Select an alternative file-basename for checkpoints (See `CHECKPOINTS` below).

`-N{max}`
Limit the number of checkpoint files (See `CHECKPOINTS` below).

`-A{minutes}`
Perform an automatic checkpoint every given number of minutes (See `CHECKPOINTS` below).

`-R{filename}`
Recover and continue to run from a saved checkpoint file (See `CHECKPOINTS` below).

`[-O{fd=filename}[,{fd2=filename2}]]...`
When using the `-R{filename}` argument to recover after a checkpoint, replace one or more file-descriptors (See `CHECKPOINTS` below).

`-I{filename}`
Inspect a checkpoint file (See `CHECKPOINTS` below).

`-X {/directory}`
Declare a private temporary directory (see `PRIVATE TEMPORARY FILES` below).

`-c`
System calls and I/O operations are monitored and taken into account in automatic migration considerations, tending to pull processes towards their home-nodes. Use this flag if you want to tell `mosrun` to not take system calls and I/O operations into the migration considerations.

`-n`
Reverse the effect of the `-c` flag: include system-calls and I/O operations into account in automatic migration considerations.

`-d{decay}`
Set the rate of decay of process-statistics for automatic migration considerations as a fraction of 10000 per second (see `mosix(7)`). `'decay'` must be an integer between 0 (immediate decay) and 10000 (no decay at all). The default decay is 9976.

Note that the following arguments may also be changed at run time by the program itself: `-m, -G, -e/-w/-u, -L/-l, -g/-G, -c, -t/-T, -C, -N, -A, -c/-n/-d` (See `mosix(7)`).

CHECKPOINTS

Most CPU-intensive processes running under `mosrun` can be checkpointed: this means that an image of those processes is saved to a file, and when necessary, the process can later recover itself from that file and continue to run from that point.

For successful checkpoint and recovery, the process must not depend heavily on its Linux environment. Specifically, the following processes cannot be checkpointed at all:

1. Processes with setuid/setgid privileges (for security reasons).
2. Processes with open pipes or sockets.

The following processes can be checkpointed, but may not run correctly after being recovered:

1. Processes that rely on process-ID's of themselves or other processes (parent, sons, etc.).
2. Processes that rely on parent-child relations (e.g. use `wait(2)`, Terminal job-control, etc.).
3. Processes that coordinate their input/output with other running processes.
4. Processes that rely on timers and alarms.
5. Processes that cannot afford to lose signals.
6. Processes that use system-V IPC (semaphores and messages).

The `-C{filename}` argument specifies where to save checkpoints: when a new checkpoint is saved, that file-name is given a consecutive numeric extension (unless it already has one). For example, if the argument `-Cmysave` is given, then the first checkpoint will be saved to `mysave.1`, the second to `mysave.2`, etc., and if the argument `-Csave.4` is given, then the first checkpoint will be saved to `save.4`, the second to `save.5`, etc. If the `-C` argument is not provided, then the checkpoints will be saved to the default: `ckpt.{pid}.1`, `ckpt.{pid}.2` ... The `-C` argument is NOT inherited by child processes.

The `-N{max}` argument specifies the maximum number of checkpoints to produce before recycling the checkpoint versions. This is mainly needed in order to save disk space. For example, when running with the arguments: `-Csave.4 -N3`, checkpoints will be saved in `save.4`, `save.5`, `save.6`, `save.4`, `save.5`, `save.6`, `save.4` . . .
The `-N0` argument returns to the default of unlimited checkpoints; an argument of `-N1` is risky, because if there is a crash just at the time when a backup is taken, there could be no remaining valid checkpoint file. Similarly, if the process can possibly have open pipe(s) or socket(s) at the time a checkpoint is taken, a checkpoint file will be created and counted - but containing just an error message, hence this argument should have a large-enough value to accommodate this possibility. The `-N` argument is NOT inherited by child processes.

Checkpoints can be triggered by the program itself, by a manual request (see `mosmigrate(1)`) and/or at regular time intervals. The `-A{minutes}` argument requests that checkpoints be automatically taken every given number of minutes. Note that if the process is within a blocking system-call (such as reading from a terminal) when the time for a checkpoint comes, the checkpoint will be delayed until after the completion of that system call. Also, when the process is frozen, it will not produce a checkpoint until unfrozen. The `-A` argument is NOT inherited by child processes.

With the `-R{filename}` argument, `mosrun` recovers and continue to run the process from its saved checkpoint file. Program options are not permitted with `-R`, since their values are recovered from the checkpoint file.

It is not always possible (or desirable) for a recovered program to continue to use the same files that were open at the time of checkpoint: `mosrun -I{filename}` inspects a checkpoint file and lists the open files, along with their modes, flags and offsets, then the `-O` argument allows the recovered program to continue using different files. Files specified using this option, will be opened (or created) with the previous modes, flags and offsets. The format of this argument is usually a comma-separated list of file-descriptor integers, followed by a '=' sign and a file-name. For example: `-O1=oldstdout,2=oldstderr,5=tmpfile`, but in case one or more file-names contain a comma, it is optional to begin the argument with a different separator, for example: `-O@1=file,with,commas@2=oldstderr@5=tmpfile`.

In the absence of the -O argument, regular files and directories are re-opened with the previous modes, flags and offsets.

Files that were already unlinked at the time of checkpoint, are assumed to be temporary files belonging to the process, and are also saved and recovered along with the process (an exception is if an unlinked file was opened for write-only). Unpredictable results may occur if such files are used to communicate with other processes.

As for special files (most commonly the user's terminal, used as standard input, output or error) that were open at the time of checkpoint - if mosrun is called with their file-descriptors open, then the existing open files are used (and their modes, flags and offsets are not modified). Special files that are neither specified in the -O argument, nor open when calling mosrun, are replaced with /dev/null.

While a checkpoint is being taken, the partially-written checkpoint file has no permissions (chmod 0). When the checkpoint is complete, its mode is changed to 0400 (read-only).

PRIVATE TEMPORARY FILES

Normally, all files are created on the home-node by migratable programs and all file-operations are performed there. This is important because programs often share files, but can be costly: many programs use temporary files which they never share - they create those files as secondary-memory and discard them when they terminate. It is best to migrate such files with the process rather than to keep them in the home-node.

The -X {/directory} argument tells Mosrun that a given directory is only used for private temporary files: all files that the program creates in this directory are kept with the process that created them and migrate with it.

The -X argument may be repeated, specifying up to 10 private temporary directories. The directories must start with '/'; can be up to 256 characters long; cannot include ".."; and for security reasons cannot be within "/etc", "/proc", "/sys" or "/dev".

Only regular files are permitted within private temporary directories: no sub-directories, links, symbolic-links or special files are allowed (but sub-directories can be specified by an extra -X argument).

Private temporary file names must begin with '/' (no relative pathnames) and contain no ".." components. The only file operations currently supported for private temporary files are: open, creat, lseek, read, write, close, chmod, fchmod, unlink, truncate, ftruncate, access, stat.

File-access permissions on private temporary files are provided for compatibility, but are not enforced: the stat(2) system-call returns 0 in st_uid and st_gid. stat(2) also returns the file-modification times according to the node where the process was running when making the last change to the file.

The per-process maximum total size of all private temporary files is set by the system-administrator. Different maximum values can be imposed when running on the home-node, in the local cluster and on other clusters in the multi-cluster - exceeding this maximum will cause a process to migrate back to its home-node.

ALTERNATIVE FREEZING SPACE

Migratable processes can sometimes be frozen (you can freeze your processes manually and the system-administrator usually sets an automatic-freezing policy - See mosix(7)).

The memory-image of frozen processes is saved to disk. Normally the system-administrator determines where on disk to store your frozen processes, but you can override this default and set your own freezing-space. One possible reason to do so is to ensure that your processes (or some of them) have sufficient freezing space regardless of what other users do. Another possible reason is to protect other users if you believe that your processes (or some of them) may require so much memory that they could disturb other users.

Setting your own freezing space can be done either by setting the environment-variable FREEZE_DIR to an alternative directory (starting with '/'); or if you wish to specify more than one freeze-directory, by creating a

file: `$HOME/.freeze_dirs` where each line contains a directory-name starting with '/'. For more details, read about "lines starting with '/'" within the section about configuring `/etc/mosix/freeze.conf` in the `mosix(7)` manual.

You must have write-access to the your alterantive freeze-directory(s). The space available in alternative freeze-directories is subject to possible disk quotas.

RECURSIVE MOSRUN

It is possible to invoke `mosrun` within a program that is already running under `mosrun`. This is common, for example, within shell-scripts or a `Makefile` that contains calls to `mosrun`.

Unless requesting explicitly otherwise, recursive programs will inherit the following (and only the following) arguments:

`-c, -d, -e, -L, -l, -g, -G, -m, -n, -T, -t, -u, -w`.

If you want a program running under `mosrun` (including a shell or shell-script) to fork a non-migratable child-program, use the utility:

```
mosnative {program} [args]...
```

`Mosnative` programs are run directly under Linux in their parent's home-node and are not subject to the limitations of migratable programs (but cannot migrate to other nodes either).

MOSENV

The variant `mosenv` is used to circumvent the loss of certain environment variables by the GLIBC library due to the fact that `mosrun` is a "setuid" program: if your program relies on the settings of dynamic-linking environment variables (such as LD_LIBRARY_PATH) or `malloc(3)` debugging (MALLOC_CHECK_), use `mosenv` instead of `mosrun`.

LIMITATIONS

Some system-calls are not supported by migratable programs, including system-calls that are tightly connected to resources of the local node or intended for system-administration. These are:

acct, add_key, adjtimex, afs_syscall, bdflush, capget, capset, chroot, clock_getres, clock_nanosleep, clock_settime, create_module, delete_module, epoll_create, epoll_create1, epoll_ctl, epoll_pwait, epoll_wait, eventfd, eventfd2, fanotify_init, fanotify_mark, futex, get_kernel_syms, get_mempolicy, get_robust_list, getcpu, getpmsg, init_module, inotify_add_watch, inotify_init, inotify_init1, inotify_rm_watch, io_cancel, io_destroy, io_getevents, io_setup, io_submit, ioperm, iopl, ioprio_get, ioprio_set, kexec_load, keyctl, lookup_dcookie, madvise, mbind, migrate_pages, mlock, mlockall, move_pages, mq_getsetattr, mq_notify, mq_open, mq_timedreceive, mq_timedsend, mq_unlink, munlock, munlockall, nfsservctl, perf_event_open, personality, pivot_root, prlimit64, prof_counter_open, ptrace, quotactl, reboot, recvmmsg, remap_file_pages, request_key, rt_sigqueueinfo, rt_sigtimedwait, rt_tgsigqueueinfo, sched_get_priority_max, sched_get_priority_min, sched_getaffinity, sched_getparam, sched_getscheduler, sched_rr_get_interval, sched_setaffinity, sched_setparam, sched_setscheduler, security, set_mempolicy, setdomainname, sethostname, set_robust_list, settimeofday, shmat, signalfd, signalfd4, swapoff, swapon, syslog, timer_create, timer_delete, timer_getoverrun, timer_gettime, timer_settime, timerfd, timerfd_gettime, timerfd_settime, tuxcall, unshare, uselib, vmsplice, waitid.

In addition, `mosrun` supports only limited options for the following system-calls:

`clone` The only permitted flags are CLONE_CHILD_SETTID, CLONE_PARENT_SETTID, CLONE_CHILD_CLEARTID, and the combination CLONE_VFORK|CLONE_VM; the child-termination signal must be SIGCLD and the stack-pointer (child_stack) must be NULL.

`getpriority`
: may refer only to the calling process.

`ioctl` The following requests are not supported: TIOCSERGSTRUCT, TIOCSERGETMULTI, TIOCSERSETMULTI, SIOCSIFFLAGS, SIOCSIFMETRIC, SIOCSIFMTU, SIOCSIFMAP, SIOCSIFHWADDR, SIOCSIFSLAVE, SIOCADDMULTI, SIOCDELMULTI, SIOCSIFHWBROADCAST, SIOCSIFTXQLEN, SIOCSMIIREG, SIOCBONDENSLAVE, SIOCBONDRELEASE, SIOCBONDSETHWADDR, SIOCBONDSLAVEINFOQUERY, SIOCBONDINFOQUERY, SIOCBONDCHANGEACTIVE, SIOCBRADDIF, SIOCBRDELIF. Non-standard requests that are defined in drivers that are not part of the standard Linux kernel are also likely to not be supported.

`ipc` the following SYSV-IPC calls are not supported: `shmat`, `semtimedop`, new-version calls (bit 16 set in call-number).

`mmap` MAP_SHARED and mapping of special-character devices are not permitted.

`prctl` only the PR_SET_DEATHSIG and PR_GET_DEATHSIG options are supported.

`setpriority`
: may refer only to the calling process.

`setrlimit`
: it is not permitted to modify the maximum number of open files (RLIMIT_NOFILES): `mosrun` fixes this limit at 1024.

Users are not permitted to send the SIGSTOP signal to programs run by `mosrun`. SIGTSTP should be used instead (and `moskillall(1)` changes SIGSTOP to SIGTSTP).

Attempts to run 32-bit programs by `mosrun` will result in the program running in native mode (as if it was run by `mosnative`).

SEE ALSO

`mosmigrate`(1), `moskillall`(1), `mosps`(1), `direct_communication`(7), `mosix`(7).

NAME

`mostestload` - MOSIX test program

SYNOPSIS

`mostestload [OPTIONS]`

DESCRIPTION

A test program that generates artificial load and consumes memory for testing the operation of MOSIX.

OPTIONS

`-t{seconds} | --time={seconds}`
: Run for a given number of CPU `seconds`: the default is 1800 seconds (30 minutes). A value of 0 causes `mostestload` to run indefinitely. OR:

`-t{min},{max} | --time={min},{max}`
: Run for a random number of seconds between `min` and `max`.

`-m{mb}, --mem={mb}`
: amount of memory to consume in Megabytes (by default, `mostestload` consumes no significant amount of memory).

`--random-mem`
: Fill memory with a random pattern (otherwise, memory is filled with the same byte-value).

`--cpu={N}`
: When testing pure CPU operations - perform `N` units of CPU work, then exit. When also doing system-calls (`--read`, `--write`, `--noiosyscall`) – perform `N` units of CPU work between chunks of system-calls.

`--read[={size}[,{ncalls}[,{repeats}]]]`

`--write[={size}[,{ncalls}[,{repeats}]]]`
: perform `read` OR `write` system calls of `size` KiloBytes (default=1KB). These calls are repeated in a chunk of `ncalls` times (default=1024), then those chunks are repeated `repeats` times (default=indefinitely), with optional CPU work between chunks if the `--cpu` option is also set.

`--noiosyscall={ncalls}[,{repeats}]`
: perform some other system call that does not involve I/O `ncalls` times (default=1024), repeat this `{repeats}` times (default=indefinitely), with optional CPU work in between if the `--cpu` option is also set.

`-d, --dir={directory}`

`-f, --file={filename}`
: select a directory OR a file on which to perform reading or writing (the default is to create a file in the /tmp directory).

`--maxiosize={SIZE}`
: Once the file size reaches `SIZE` megabytes, further I/O will resume at the beginning of the file.

`-v, --verbose`
: produce debug-output.

`--report-migrations`
: Report when `mostestload` migrates.

`-r, --report`
: Produce summary at end of run.

`--sleep SEC`
: Sleep for `SEC` seconds before starting

`-h, --help`
: Display a short help screen.

EXAMPLES

`mostestload -t 20`
run CPU for 20 seconds

`mostestload -l 10 -h 20`
runs CPU for a random period of time between 10 and 20 seconds.

`mostestload -f /tmp/20MB --write 32,640,1`
writes 32 KiloBytes of data 640 times (total 20 megabytes) to the file /tmp/20MB.

`mostestload -f /tmp/10MB --write 32,640 --maxiosize 10 --cpu=20`
writes 32 KiloBytes of data 640 times (total 20 megabytes) to the file /tmp/10MB, alternating this indefinitely with running 20 units of CPU. The file "/tmp/10MB" is not allowed to grow beyond 10 MegaBytes: once reaching that limit, writing resumes at the beginning of the file.

AUTHOR

Adapted from code by Lior Amar

NAME

MOSTIMEOF - Report CPU usage of migratable processes

SYNOPSIS

timeof {pid}...

DESCRIPTION

Mostimeof reports the amount of CPU-time accumulated by one or more MOSIX/migratable processes, no matter where they run. Its argument(s) are the process-IDs of processes to inspect.

NOTES

1. Mostimeof must run on the process' home-node.
2. The report is of user-level CPU-time: system-time is not included.
3. In clusters (or multi-clusters) where different nodes have different CPU speeds, the results could be the sum of CPU times from slower and faster processors. Such reslts cannot be used for determining how long the inspected process(es) are still expected to run.

SEE ALSO

mosps(1), mosrun(1), mosix(7).

NAME

DIRECT COMMUNICATION — migratable sockets between MOSIX processes

PURPOSE

Normally, migratable/MOSIX processes do all their I/O (and most system-calls) via their home-node: this can be slow because operations are limited by the network speed and latency. `Direct communication` allows processes to pass messages directly between them, bypassing their home-nodes.

For example, if process X whose home-node is A and runs on node B wishes to send a message over a socket to process Y whose home-node is C and runs on node D, then the message has to pass over the network from B to A to C to D. Using `direct communication`, the message will pass directly from B to D. Moreover, if X and Y run on the same node, the network is not used at all.

To facilitate `direct communication`, each MOSIX process (running under `mosrun(1)`) can own a "mailbox". This mailbox can contain at any time up to 10000 unread messages of up to a total of 32MB. MOSIX Processes can send messages to mailboxes of other processes anywhere within the multi-cluster (that are willing to accept them).

`Direct communication` makes the location of processes transparent, so the senders do not need to know where the receivers run, but only to identify them by their home-node and process-ID (PID) in their home-node.

`Direct communication` guarantees that the order of messages per receiver is preserved, even when the sender(s) and receiver migrate - no matter where to and how many times they migrate.

SENDING MESSAGES

To start sending messages to another process, use:

```
them = open("/proc/mosix/mbox/{a.b.c.d}/{pid}", 1);
```

where `{a.b.c.d}` is the IP address of the receiver's home-node and `{pid}` is the process-ID of the receiver. To send messages to a process with the same home-node, you can use `0.0.0.0` instead of the local IP address (this is even preferable, because it allows the communication to proceed in the rare event when the home-node is shut-down from its cluster).

The returned value (`them`) is not a standard (POSIX) file-descriptor: it can only be used within the following system calls:

```
w = write(them, message, length);
fcntl(them, F_SETFL, O_NONBLOCK);
fcntl(them, F_SETFL, 0);
dup2(them, 1);
dup2(them, 2);
close(them);
```

Zero-length messages are allowed.

Each process may at any time have up to 128 open `direct communication` file-descriptors for sending messages to other processes. These file-descriptors are inherited by child processes (after `fork(2)`).

When `dup2` is used as above, the corresponding file-descriptor (1 for standard-output; 2 for standard-error) is associated with sending messages to the same process as `them`. In that case, only the above calls (`write`, `fcntl`, `close`, but not `dup2`) can then be used with that descriptor.

RECEIVING MESSAGES

To start receiving messages, create a mailbox:

```
my_mbox = open("/proc/mosix/mybox", O_CREAT, flags);
```

where `flags` is any combination (bitwise OR) of the following:

1 Allow receiving messages from other users of the same group (GID).
2 Allow receiving messages from all other users.
4 Allow receiving messages from processes with other home-nodes.
8 Do not delay: normally when attempting to receive a message and no fitting message was received, the call blocks until either a message or a signal arrives, but with this flag, the call returns immediately a value of -1 (with `errno` set to EAGAIN).
16 Receive a `SIGIO` signal (See `signal(7)`) when a message is ready to be read (for assynchroneous operation).
32 Normally, when attempting to read and the next message does not fit in the read buffer (the message length is bigger than the `count` parameter of the `read(2)` system-call), the next message is truncated. When this bit is set, the first message that fits the read-buffer will be read (even if out of order): if none of the pending messages fits the buffer, the receiving process either waits for a new message that fits the buffer to arrive, or if bit 8 ("do not delay") is also set, returns -1 with `errno` set to EAGAIN.
64 Treat zero-length messages as an end-of-file condition: once a zero-length message is read, all further reads will return 0 (pending and future messages are not deleted, so they can still be read once this flag is cleared).

The returned value (`my_mbox`) is not a standard (POSIX) file-descriptor: it can only be used within the following system calls:

```
r = read(my_mbox, buf, count);
r = readv(my_mbox, iov, niov);
dup2(my_mbox, 0);
close(my_mbox);
ioctl(my_mbox, SIOCINTERESTED, addr);
ioctl(my_mbox, SIOCSTOREINTERESTS, addr);
ioctl(my_mbox, SIOCWHICH, addr);
```

(see FILTERING below)

Reading `my_mbox` always reads a single message at a time, even when `count` allows reading more messages. A message can have zero-length, but `count` cannot be zero.

A `count` of -1 is a special request to test for a message without actually reading it. If a message is present for reading, `read(my_mbox, buf, -1)` returns its length - otherwise it returns -1 with `errno` set to `EAGAIN`.

unlike in "SENDING MESSAGES" above, `my_mbox` is NOT inherited by child processes.

When `dup2` is used as above, file-descriptor 0 (standard-input) is associated with receiving messages from other processes, but only the `read`, `readv` and `close` system-calls can then be used with file-descriptor 0.

Closing `my_mbox` (or `close(0)` if `dup2(my_mbox, 0)` was used - whichever is closed last) discards all pending messages.

To change the `flags` of the mailbox without losing any pending messages, open it again (without using `close`):

```
my_mbox = open("/proc/mosix/mybox", O_CREAT, new_flags);
```

Note that when removing permission-flags (1, 2 and 4) from `new_flags`, messages that were already sent earlier will still arrive, even from senders that are no longer allowed to send messages to the current process. Re-opening always returns the same value (`my_mbox`) as the initial `open` (unless an error occurs and -1 is returned). Also note that if `dup2(my_mbox, 0)` was used, `new_flags` will immediately apply to file-descriptor 0 as well.

Extra information is available about the latest message that was read (including when the `count` parameter of the last `read()` was -1 and no reading actually took place). To get this information, you should first define the following macro:

```
static inline unsigned int GET_IP(char *file_name)
{
        int ip = open(file_name, 0);
        return((unsigned int)((ip==-1 && errno>255) ? -errno:
        ip));
}
```

To find the IP address of the sender's home, use:

```
sender_home = GET_IP("/proc/self/sender_home");
```

To find the process-ID (PID) of the sender, use:

```
sender_pid = open("/proc/self/sender_pid", 0);
```

To find the IP address of the node where the sender was running when the message was sent, use:

```
sender_location = GET_IP("/proc/self/sender_location");
```

(this can be used, for example, to request a manual migration to bring together communicating processes to the same node)

To find the length of the last message, use:

```
bytes = open("/proc/self/message_length", 0);
```

(this makes it possible to detect truncated messages: if the last message was truncated, `bytes` will contain the original length)

FILTERING

The following facility allows the receiver to select which types of messages it is interested to receive:

```
struct interested
{
        unsigned char conditions; /* bitmap of conditions */
        unsigned char testlen;    /* length of test-pattern (1-8 bytes) */
        int pid;          /* Process-ID of sender */
        unsigned int home;        /* home-node of sender (0 = same home) */
        int minlen;               /* minimum message length */
        int maxlen;               /* maximum message length */
        int testoffset;   /* offset of test-pattern within message */
        unsigned char testdata[8]; /* expected test-pattern */
        int msgno;                /* pick a specific message (starting from 1) */
        int msgoffset;            /* start reading from given offset */
};

/* conditions: */
#define INTERESTED_IN_PID     1
#define INTERESTED_IN_HOME 2
```

```
#define INTERESTED_IN_MINLEN        4
#define INTERESTED_IN_MAXLEN        8
#define INTERESTED_IN_PATTERN       16
#define INTERESTED_IN_MESSAGENO     32
#define INTERESTED_IN_OFFSET        64
#define PREVENT_REMOVAL             128

struct interested filter;

struct interests
{
        long number;                    /* number of filters */
        struct interested *filters;     /* filters to store */
} filters;

#define SIOCINTERESTED          0x8985
#define SIOCKSTOREINTERESTS 0x8986
#define SIOCWHICH 0x8987
```

A call to:

```
ioctl(my_mbox, SIOCINTERESTED, &filter);
```

starts applying the given `filter`, while a call to:

```
ioctl(my_mbox, SIOCINTERESTED, NULL);
```

cancels the filtering. Closing `my_mbox` also cancels the filtering (but re-opening with different flags does not cancel the filtering).

Calls to this `ioctl` return the address of the previous filter.

When filtering is applied, only messages that comply with the filter are received: if there are no complying messages, the receiving process either waits for a complying message to arrive, or if bit 8 ("do not delay") of the `flags` from `open("/proc/self/mybox", O_CREAT, flags)` is set, `read(my_mbox,...)` and `readv(my_mbox,...)` return -1 with `errno` set to EAGAIN. Filtering can also be used to test for particular messages using `read(my_mbox, buf, -1)`.

Different types of messages can be received simply by modifying the contents of the `filter` between calls to `read(my_mbox,...)` (or `readv(my_mbox,...)`).

`filter.conditions` is a bit-map indicating which condition(s) to consider:

When `INTERESTED_IN_PID` is set, the process-ID of the sender must match `filter.pid`.

When `INTERESTED_IN_HOME` is set, the home-node of the sender must match `filter.home` (a value of 0 can be used to match senders from the same home-node).

When `INTERESTED_IN_MINLEN` is set, the message length must be at least `filter.minlen` bytes long.

When `INTERESTED_IN_MAXLEN` is set, the message length must be no longer than `filter.maxlen` bytes.

When `INTERESTED_IN_PATTERN` is set, the message must contain a given pattern of data at a given offset. The offset within the message is given by `filter.testoffset`, the pattern's length (1 to 8 bytes) in `filter.testlen` and its expected contents in `filter.testdata`.

When `INTERESTED_IN_MESSAGENO` is set, the message numbered `filter.msgno` (numbering starts from 1) will be read out of the queue of received messages.

When `INTERESTED_IN_OFFSET` is set, reading begins at the offset `filter.msgoffset` of the message's data.

When `PREVENT_REMOVAL` is set, read messages are not removed from the message-queue, so they can be re-read until this flag is cleared.

A call to:

```
ioctl(my_mbox, SIOCSTOREINTERESTS, &filters);
```

stores an array of filters for later use by MOSIX: `filters.number` should contain the number of filters (0-1024) and `filters.filters` should point to an array of filters (in which the conditions `INTERESTED_IN_MESSAGENO`, `INTERESTED_IN_OFFSET` and `PREVENT_REMOVAL` are ignored). Successful calls return 0.

Closing `my_mbox` also discards the stored filters (but re-opening with different flags does not).

A call to:

```
ioctl(my_mbox, SIOCWHICH, &bitmap)
```

fills the given bitmap with information, one bit per filter, about whether (1) or not (0) there are any pending messages that match the filters that were previously stored by `SIOCSTOREINTERESTS` (above).

The number of bytes affected in `bitmap` depends on the number of stored filters. If unsure, reserve the maximum of 128 bytes (for 1024 filters).
Successful calls return the number of filters previously stored by `SIOCSTOREINTERESTS`.

ERRORS

`Sender errors:`

ENOENT
: Invalid pathname in `open`: the specified IP address is not part of this cluster/multi-cluster, or the process-ID is out of range (must be 2-32767).

ESRCH
: No such process (this error is detected only when attempting to send - not when opening the connection).

EACCES
: No permission to send to that process.

ENOSPC
: Non-blocking (O_NONBLOCK) was requested and the receiver has no more space to accept this message - perhaps try again later.

ECONNABORTED
: The home-node of the receiver is no longer in our multi-cluster.

EMFILE
: The maximum of 128 `direct communicaiton` file-descriptors is already in use.

EINVAL
: When opening, the second parameter does not contain the bit "1"; When writing, the length is negative or more than 32MB.

ETIMEDOUT
: Failed to establish connection with the mail-box managing daemon (`mospostald`).

ECONNREFUSED
: The mail-box managing (`mospostald`) refused to serve the call (probably a MOSIX installation error).

EIO Communication breakdown with the mail-box managing daemon (mospostald).

`Receiver errors:`

EAGAIN
No message is currently available for reading and the "Do not delay" flag is set (or count is -1).

EINVAL One or more values in the filtering structure are illegal or their combination makes it impossible to receive any message (for example, the offset of the data-pattern is beyond the maximum message length). Also, an attempt to store either a negative number or more than 1024 filters.

ENODATA
The INTERESTED_IN_MESSAGENO filter is used, and either "no truncating" was requested (32 in the open-flags) while the message does not fit the read buffer, or the message does not fulfil the other filtering conditions.

`Errors that are common to both sender and receiver:`

EINTR Read/write interrupted by a signal.

ENOMEM
Insufficient memory to complete the operation.

EFAULT
Bad read/write buffer address.

ENETUNREACH
Could not establish a connection with the mail-box managing deamon (mospostald).

ECONNRESET
Connection lost with the mail-box managing daemon (mospostald).

POSSIBLE APPLICATIONS

The scope of direct communication is very wide: almost any program that requires communication between related processes can benefit. Following are a few examples:

1. Use direct communication within standard communication packages and libraries, such as MPI.
2. Pipe-like applications where one process' output is the other's input: write your own code or use the existing mospipe(1) MOSIX utility.
3. Direct communiction can be used to implement fast I/O for migrated processes (with the cooperation of a local process on the node where the migrated process is running). In particular, it can be used to give migrated processes access to data from a common NFS server without causing their home-node to become a bottleneck.

LIMITATIONS

Processes that are involved in direct communication (having open file-descriptors for either sending or receiving messages) cannot be checkpointed and cannot execute mosrun recursively or mosnative (see mosrun(1)).

SEE ALSO

mosrun(1), mospipe(1), mosix(7).

NAME

MOSIX — sharing the power of clusters and multi-cluster private clouds

INTRODUCTION

MOSIX is a generic solution for dynamic management of resources in a cluster or in a multi-cluster private cloud. MOSIX allows users to draw the most out of all the connected computers, including utilization of idle computers.

At the core of MOSIX are adaptive resource sharing algorithms, applying preemptive process migration based on processor loads, memory and I/O demands of the processes, thus causing the cluster or the multi-cluster to work cooperatively similar to a single computer with many processors.

Unlike earlier versions of MOSIX, only programs that are started by the `mosrun(1)` utility are affected and can be considered "migratable" - other programs are considered as "standard Linux programs" and are not affected by MOSIX.

MOSIX maintains a high level of compatiblity with standard Linux, so that binaries of almost every application that runs under Linux can run completely unmodified under MOSIX. The exceptions are usually system-administration or graphic utilities that would not benefit from process-migration anyway. If a MOSIX program that was started by `mosrun(1)` attempts to use unsupported features, it will either be killed with an appropriate error message, or if a ‘‘do not kill’’ option is selected, an error is returned to the program: such programs should probably run as standard Linux programs.

In order to improve the overall resource usage, MOSIX processes may be moved automatically and transparently to other nodes within the cluster or even the multi-cluster private cloud. As the demands for resources change, processes may move again, as many times as necessary, to continue optimizing the overall resource utilization, subject to the inter-cluster priorities and policies. Manual-control over process migration is also supported.

`MOSIX` is particularly suitable for running CPU-intensive computational programs with unpredictable resource usage and run times, and programs with moderate amounts of I/O. Programs that perform large amounts of I/O should better be run as standard Linux programs.

REQUIREMENTS

1. All nodes must run a suitable Linux kernel, which can be either:

   A. Any Linux kernel version 3.12 or higher.
   B. Any kernel from a Linux distribution based on Linux kernel version 3.12 or higher.
   C. Any Linux kernel generated by an older MOSIX distribution, version 3.4.0.0 or higher.
   D. A standard kernel from OpenSUSE version 13.1 or higher.

2. All participating nodes must be connected to a network that supports TCP/IP and UDP/IP, where each node has a unique IP address in the range 0.1.0.0 to 255.255.254.255 that is accessible to all the other nodes.

3. TCP/IP ports 252-253 and UDP/IP ports 249 and 253 must be available for MOSIX (not used by other applications or blocked by a firewall).

4. The architecture of all nodes must be x86_64 (64-bit).

5. All the processors (cores) of a node must be of the same speed.

6. Specific CPUs have added non-standard features, in particular Intel’s SSE4.1 and SSE4.2 standards, which are not present on either older Intel processors or in AMD processors. Newer versions of the "glibc" library (> glibc-2.8) detect those features during process-initialisation and record their presence

for later optimisations. Unfortunately, this means that a process which started on a node that has those features will be unable to migrate to nodes that do not have them (if it does, it could crash with an "Illegal Instruction" fault). Accordingly, you should not mix newer Intel processors in the same MOSIX cluster (or multi-cluster) with older Intel processors or with AMD computers. Alternately, re-compile the "glibc" library with the "--disable-multi-arch" flag.

7. The system-administrators of all the connected nodes must be able to trust each other (see more on SECURITY below).

CLUSTER AND MULTI-CLUSTER PRIVATE CLOUD

The MOSIX concept of a "cluster" is a collection of computers that are owned and managed by the same entity (a person, a group of people or a project) - this can at times be quite different than a hardware cluster, as each MOSIX cluster may range from a single workstation to a large combination of computers - workstations, servers, blades, multi-core computers, etc. possibly of different speeds and number of processors and possibly in different locations.

A MOSIX multi-cluster private cloud (often shortened to "multi-cluster") is a collection of clusters that may belong to different, yet trusting, entities (owners) who wish to share their resources subject to certain administrative conditions. In particular, when an owner needs its computers - these computers must be returned immediately to the exclusive use of their owner. An owner can also assign priorities to guest processes of other owners, defining who can use their computers and when. Typically, an owner is an individual user, a group of users or a department that own the computers. The multi-cluster private cloud is usually restricted, due to trust and security reasons, to a single organization, possibly in various sites/branches or even across the world.

MOSIX supports dynamic multi-cluster configurations, where clusters can join and leave at any time. When resources become scarce (because other clusters leave or claim their resources and processes must migrate back to their home-clusters), MOSIX has a freezing feature that can automatically freeze excess processes to prevent memory-overload on the home-nodes.

CONFIGURATION

To configure MOSIX interactively, simply run `mosconf`: it will lead you step-by-step through the various configuration items.

`Mosconf` can be used in two ways:

1. To configure the local node (press <Enter> at the first question).

2. To configure MOSIX for other nodes: this is typically done on a server that stores an image of the root-partition for some or all of the cluster-nodes. This image can, for example, be NFS-mounted by the cluster-nodes, or otherwise copied or reflected to them by any other method: at the first question, enter the path to the stored root-image.

There is no need to stop MOSIX in order to modify the configuration - most changes will take effect within a minute. However, after modifying the list of nodes in the cluster (`/etc/mosix/mosix.map`) or `/etc/mosix/mosip`, you should run the command "`mossetpe`" (but when you are using `mosconf` to configure your local node, this is not necessary).

Below is a detailed description of the MOSIX configuration files (if you prefer to edit them manually).

The directory `/etc/mosix` should include at least the subdirectories `/etc/mosix/partners`, `/etc/mosix/var` and the following files:

`/etc/mosix/mosix.map`

This file defines which computers participate in your MOSIX cluster. The file contains up to 256 data lines and/or alias lines that can appear in any order. It may also include any number of comment lines beginning with a '#', as well as empty lines.

Data lines have 2 or 3 fields:

1. The IP ("a.b.c.d" or host-name) of the first node in a range of nodes with consecutive IPs.
2. The number of nodes in that range.
3. The optional letter 'p' for "proximate", meaning that the network to those nodes is fast, so data-compression should not be used during migration.

Alias lines are of the form:

```
a.b.c.d=e.f.g.h
```

or

```
a.b.c.d=host-name
```

They indicate that the IP address on the left-hand-side refers to the same node as the right-hand-side.

NOTES:

1. When using host names, the first result of `gethostbyname(3)` must return their IP address that is to be used by MOSIX: if in doubt - specify the IP address.
2. The right-hand-side in alias lines must either appear within the data lines or refer a node belonging to another cluster within the MOSIX multi-cluster.
3. IP addresses 0.0.x.x and 255.255.255.x are not allowed in MOSIX.
4. If you change `/etc/mosix/mosix.map` while MOSIX is running, you need to run `mossetpe` to notify MOSIX of the changes.

`/etc/mosix/secret`

This is a security file that is used to prevent ordinary users from interfering and/or compromizing security by connecting to the internal MOSIX TCP ports. The file should contain just a single line with a password that must be identical on all the nodes of the cluster/multi-cluster. This file must be accessible to ROOT only (chmod 600!)

The following files are optional:

`/etc/mosix/userview.map`

Although it is possible to use only IP numbers and/or host-names to specify nodes in your cluster (and multi-cluster), it is more convenient to use small integers as node numbers: this file allows you to map integers to IP addresses. Each line in this file contains 3 elements:

1. A node number (1-65535)
2. IP1 (or host-name, clearly identifiable by `gethostbyname(3)`)
3. Number of nodes in range (the number of the last one must not exceed 65535)

It is up to the cluster administrator to map as few or as many nodes as they wish out of their cluster and multi-cluster - the most common practice is to map all the nodes in one's cluster, but not in other clusters.

`/etc/mosix/speed`

If this file exists, it should contain a positive integer (1-10,000,000), providing the relative speed of the processor: the bigger the faster. It is recommended to set the speed of the most typical processor in your cluster(s) to 10,000, then the others accordingly.

If this file is absent, then the speed is assumed to be 10000.

As some processors do better than others with certain types of applications, in setting the processor's speed, the system-administrator should, when possible, take into account the typical applications that users intend to run on the MOSIX cluster(s).

`/etc/mosix/mosip`

This file contains our IP address, to be used for MOSIX purposes, in the regular format - `a.b.c.d`. This file is only necessary when the node's IP address is ambiguous: it can be safely omitted if the output of `ifconfig(8)` ("inet" or "inet addr:") matches exactly one of the IP addresses listed in the data lines of `/etc/mosix/mosix.map`.

`/etc/mosix/freeze.conf`

This file sets the freezing policies (affecting all MOSIX process originating in this node). The lines in this file can appear in any order.

An automatic freezing policy is defined by a line with the following space-separated fields:

1. the number '1'.
2. load-units:
   Used in fields #3-#6 below: 0=processes; 1=standard-load
3. RED-MARK (floating point)
   Freeze when load is higher.
4. BLUE-MARK (floating point)
   Unfreeze when load is lower.
5. minautofreeze (floating point)
   Freeze processes that are evacuated back home on arrival if load gets equal or above this.
6. minclustfreeze (floating point)
   Freeze processes that are evacuated back to this cluster on arrival if load gets equal or above this.
7. min-keep
   Keep running at least this number of processes - even if load is above RED-MARK.
8. max-procs
   Freeze excess processes above this number - even if load is below BLUE-MARK.
9. slice
   Time (in minutes) that a process is allowed to run while there are automatically-frozen process(es). After this period, the running process will be frozen and a frozen process will start to run.
10. killing-memory
    Freezing fails when there is insufficient disk-space to save the memory-image of the frozen process - kill processes that failed to freeze and have above this number of MegaBytes of memory. Processes with less memory are kept alive (and in memory). Setting this value to 0, causes processes to be killed when freezing fails. Setting it to a very high value (like 1000000 MegaBytes) keeps all processes alive.

`NOTES:`

1. The load-units in fields #3-#6 depend on field #2. If 0, each unit represents the load created by a CPU-bound process on this computer. If 1, each unit represents the load created by a CPU-bound process on a "standard" MOSIX computer (e.g. a 3GHz Intel Core 2 Duo E6850). The difference is that the faster the computer and the more processors it has, the load created by each CPU process decreases proportionally.
2. Fields #3,#4,#5,#6 are floating-point, the rest are integers.
3. A value of "-1" in fields #3,#5,#6,#8 means ignoring that feature.
4. The first 4 fields are mandatory: omitted fields beyond them have the following values: minautofreeze=-1,minclusterfreeze=-1,min-keep=0, max-procs=-1,slice=20.
5. The RED-MARK must be significantly higher than BLUE-MARK: otherwise a perpetual cycle of freezing and unfreezing could occur. You should allow at least 1.1 processes difference between them.

6. Frozen processes do not respond to anything, except an unfreeze request or a signal that kills them.
7. Processes that were frozen manually are not unfrozen automatically.

This file may also contain lines starting with '/' to indicate freezing-directory names. A "Freezing directory" is an existing directory (often a mount-point) where the memory contents of frozen process is saved. For successful freezing, the disk-partition of freezing-directories should have sufficient free disk-space to contain the memory image of all the frozen processes.

If more than one freezing directory is listed, the freezing directory is chosen at random by each freezing process. It is also possible to assign selection probabilities by adding a numeric weight after the directory-name, for example:

```
/tmp      2
/var/tmp  0.5
/mnt/tmp  2.5
```

In this example, the total weight is 2+0.5+2.5=5, so out of every 10 frozen processes, an average of 4 (10∗2/5) will be frozen to `/tmp`, an average of 1 (10∗0.5/5) to `/var/tmp` and an average of 5 (10∗2.5/5) to `/mnt/tmp`.

When the weight is missing, it defaults to 1. A weight of 0 means that this directory should be used only if all others cannot be accessed.

If no freezing directories are specified, all freezing will be to the `/freeze` directory (or symbolic-link).

Freezing files are usually created with "root" (Super-User) permissions, but if `/etc/mosix/freeze.conf` contains a line of the form:

```
U {UID}
```

then they are created with permissions of the given numeric `UID` (this is sometimes needed when freezing to NFS directories that do not allow "root" access).

`/etc/mosix/partners/*`

If your cluster is part of a multi-cluster private cloud, then each file in `/etc/mosix/partners` describes another cluster that you want this cluster to cooperate with.

The file-names should indicate the corresponding cluster-names (maximum 128 characters), for example: "geography", "chemistry", "management", "development", "sales", "students-lab-A", etc. The format of each file is a follows:

`Line #1:`
A verbal human-readable description of the cluster.

`Line #2:`
Three space-separated integers as follows:

1. Priority:
   0-65535, the lower the better.
   The priority of the local cluster is always 0. MOSIX gives precedence to processes with higher priority - if they arrive, guests with lower priority will be expelled.

2. Cango:
   0=never send local processes to that cluster.
   1=local processes may go to that cluster.
3. Cantake:
   0=do not accept guest-processes from that cluster.
   1=accept guest-processes from that cluster.

`Following lines:`

Each line describes a range of consecutive IP addresses that are believed to be part of the other cluster, containing 5 space-separated items as follows:

1. IP1 (or host-name):
   First node in range.
2. n: Number of nodes in this range.
3. Obsolete: should be '1'.
4. Obsolete: should be '1'.
5. Proximate:
   0=no Use compression on migration to/from that cluster.
   1=yes Do not use compression when migrating to/from that cluster (network is very fast and CPU is slow).

`NOTES:`

1. When using host names rather than IP addresses, the first result of `gethostbyname(3)` must return their IP address that is used by MOSIX: if in doubt - specify the IP address instead.
2. IP addresses 0.0.x.x and 255.255.255.x cannot be used in MOSIX.

`/etc/mosix/private.conf`

This file specifies where Private Temporary Files (PTFs) are stored: PTFs are an important feature of `mosrun(1)` and may consume a significant amount of disk-space. It is important to ensure that sufficient disk-space is reserved for PTFs, but without allowing them to disturb other processes by filling up disk-partitions. Guest processes can also demand unpredictable amounts of disk-space for their PTFs, so we must make sure that they do not disturb local operations.

Up to 3 different directories can be specified: for local processes; guest-processes from the local cluster; and guest-processes from other clusters in the multi-cluster private cloud. Accordingly, each line in this file has 3 fields:

1. A combination of the letters: 'O' (own node), 'C' (own cluster) and 'G' (other clusters). For example, `OC`, `C`, `CG` or `OCG`.
2. A directory name (usually a mount-point) starting with '/', where PTFs for the above processes are to be stored.
3. An optional numeric limit, in Megabytes, of the total size of PTFs per-process.

If `/etc/mosix/private.conf` does not exist, then all PTFs will be stored in "/private". If the directory "/private" also does not exist, or if `/etc/mosix/private.conf` exists but does not contain a line with an appropriate letter in the first field ('O', 'C' or 'G'), then no disk-space is allocated for PTFs of the affected processes, which usually means that processes requiring PTFs will not be able to run on this node. Such guest processes that start using PTFs will migrate back to their home-nodes.

When the third field is missing, it defaults to:
5 Gigabytes for local processes.
2 Gigabytes for processes from the same cluster.
1 Gigabyte for processes from other clusters.

In any case, guest processes cannot exceed the size limit of their home-node even on nodes that allow them more space.

`/etc/mosix/retainpri`
: This file contains an integer, specifying a delay in seconds: how early after all MOSIX processes of a certain priority (priorities are defined as above in `/etc/mosix/partners/*` and the current priority can be seen in `/proc/mosix/priority`) finish (or leave) to allow processes of lower priority (higher numbers) to start. When this file is absent, there is no delay and processes with lower priority may arrive as soon as there are no processes with a higher priority.

`/etc/mosix/maxguests`
: If this file exists, it should contain one line with an integer limit on the number of simultaneous guest-processes from other clusters (otherwise, the maximum number of guest-processes from other clusters is unlimited). A second integer may be added by `'mosctl closemulti'` in order to restore the former value in a subsequent `'mosctl openmulti'`.

`/etc/mosix/.log_mosrun`
: When this file is present, information about invocations of `mosrun(1)` and process migrations will be recorded in the system-log (by default "/var/log/messages" on most Linux distributions).

INTERFACE FOR PROGRAMS

The following interface is provided for programs running under `mosrun(1)` that wish to interface with their MOSIX run-time environment:

All access to MOSIX is performed via the "open" system call, but the use of "open" is incidental and does not involve actual opening of files. If the program were to run as a regular Linux program, those "open" calls would fail, returning -1, since the quoted files never exist, and `errno(3)` would be set to ENOENT.

`open("/proc/self/{special}", 0)`
reads a value from the MOSIX run-time environment.

`open("/proc/self/{special}", 1|O_CREAT, newval)`
writes a value to the MOSIX run-time environment.

`open("/proc/self/{special}", 2|O_CREAT, newval)`
both writes a new value and return the previous value.

(the `O_CREAT` flag is only required when your program is compiled with the 64-bit file-size option, but is harmless otherwise).

Some "files" are read-only, some are write-only and some can do both (rw). The "files" are as follows:

`/proc/self/migrate`
: writing a 0 migrates back home; writing -1 causes a migration consideration; writing the unsigned value of an IP address or a logical node number, attempts to migrate there. Successful migration returns 0, failure returns -1. (write only)

`/proc/self/lock`
: When locked (1), no automatic migration may occur (except when running on the current node is no longer allowed); when unlocked (0), automatic migration can occur. (rw)

`/proc/self/freezable`
: When 1, the process can be automatically frozen. When 0, it cannot. (rw)

`/proc/self/whereami`
: reads where the program is running: 0 if at home, otherwise usually an unsigned IP address, but if possible, its corresponding logical node number. (read only)

`/proc/self/nmigs`
: reads the total number of migrations performed by this process and its MOSRUN ancesstors before it was born. (read only)

`/proc/self/sigmig`
: Reads/sets a signal number (1-64 or 0 to cancel) to be received after each migration. (rw)

`/proc/self/needmem`
: Reads/modifies the process's memory requirement in Megabytes, so it does not automatically migrate to nodes with less free memory. Acceptable values are 0-262143. (rw)

`/proc/self/unsupportok`
: when 0, unsupported system-calls cause the process to be killed; when 1 or 2, unsupported system-calls return -1 with errno set to ENOSYS; when 2, an appropriate error-message will also be written to stderr. (rw)

`/proc/self/clear`
: clears process statistics. (write only)

`/proc/self/cpujob`
: Normally when 0, system-calls and I/O are taken into account for migration considerations. When set to 1, they are ignored. (rw)

`/proc/self/decayrate`
: Reads/modifies the decay-rate per second (0-10000): programs can alternate between periods of intensive CPU and periods of demanding I/O. Decisions to migrate should be based neither on momentary program behaviour nor on extremely long term behaviour, so a balance must be struck, where old process statistics gradually decay in favour of newer statistics. The lesser the decay rate, the more weight is given to new information. The higher the decay rate, the more weight is given to older information. This option is provided for users who know well the cyclic behavior of their program. (rw)

`/proc/self/checkpoint`
: When writing (any value) - perform a checkpoint. When only reading - return the version number of the next checkpoint to be made. When reading and writing - perform a checkpoint and return its version. Returns -1 if the checkpoint fails, 0 if writing only and checkpoint is successful. (rw)

`/proc/self/checkpointfile`
: The third argument (`newval`) is a pointer to a file-name to be used as the basis for future checkpoints (see `mosrun(1)`). (write only)

`/proc/self/checkpointlimit`
: Reads/modifies the maximal number of checkpoint files to create before recycling the checkpoint version number. A value of 0 unlimits the number of checkpoints files. The maximal value allowed is 10000000.

`/proc/self/checkpointinterval`
: When writing, sets the interval in minutes for automatic checkpoints (see `mosrun(1)`). A value of 0 cancels automatic checkpoints. The maximal value allowed is 10000000. Note that writing has a side effect of reseting the time left to the next checkpoint. Thus, writing too frequently is not recommended. (rw)

`open("/proc/self/in_cluster", O_CREAT, node);`
: return 1 if the given `node` is in the same cluster, 0 otherwise. The `node` can be either an unsigned, host-order IP address, or a node-number (listed in `/etc/mosix/userview.map`).

More functions are available through the `direct_communication(7)` feature.

STARTING MOSIX

To start MOSIX, run `/etc/init.d/mosix start`. Alternately, run `mosd`.

SECURITY

All nodes within a MOSIX cluster and multi-cluster private cloud must trust each other's super-user(s) - otherwise the security of the whole cluster or multi-cluster is compromized.

Hostile computers must not be allowed physical access to the internal MOSIX network where they could masquerade as having IP addresses of trusted nodes.

SEE ALSO

`mosrun`(1), `mosctl`(1), `mosmigrate`(1), `mossetpe`(1), `mosmon`(1), `mosps`(1), `mostimeof`(1), `moskillall`(1), `mosq`(1), `mosbestnode`(1), `mospipe`(1), `mos_in_job`(1), `direct_communication`(7).

NAME

MOSCTL - Miscellaneous MOSIX functions

SYNOPSIS

mosctl stay
mosctl nostay
mosctl lstay
mosctl nolstay
mosctl block
mosctl noblock
mosctl expel
mosctl bring
mosctl shutdown
mosctl isolate
mosctl rejoin [**{maxguests}**]
mosctl maxguests [**{maxguests}**]
mosctl openmulti [**{maxguests}**]
mosctl closemulti
mosctl whois [**{node_number}|IP-address|hostname**]
mosctl status [**{node_number}|IP-address|hostname**]
mosctl localstatus
mosctl rstatus [**{node_number}|IP-address|hostname**]

DESCRIPTION

Most Mosctl functions are for MOSIX administration and are available only to the Super-User. The exceptions are the whois, status and rstatus functions which provide information to all users.

mosctl stay prevents processes from migrating away automatically: mosctl nostay cancels this state.

mosctl lstay prevents local processes from migrating away automatically, but still allows guest processes to leave: mosctl nolstay cancels this state.

mosctl block prevents guest processes from moving in: mosctl noblock cancels this state.

mosctl expel expels all guest processes. It does not return until all guest processes are moved away (it can be interrupted, in which case there is no guarantee that all guest processes were expelled).

mosctl bring brings back all processes whose home-node is here. It does not return until all these processes arrive back (it can be interrupted, in which case there is no guarantee that all the processes arrived back).

mosctl shutdown shuts down MOSIX. All guest processes are expelled and all processes whose home-node is here are brought back, then the MOSIX configuration is turned off.

mosctl isolate disconnects the cluster from the multi-cluster private cloud, bringing back all migrated processes whose home-node is in the disconnecting cluster and sending away all guest processes from other clusters. To actually disconnect a cluster, this command must be issued on all the nodes of that cluster.

mosctl rejoin cancels the effect of mosctl isolate: an optional argument sets the number of guest processes that are allowed to move to this node or run here from outside the local cluster. When this argument is missing, no guest processes from outside the cluster will be accepted.

mosctl maxguests prints the maximum number of guests that are allowed to migrate to this node from other clusters. mosctl maxguests arg, with a numeric argument arg, sets that maximum.

`mosctl openmulti` sets the maximum number of guest processes from outside the local cluster to:

1. If an argument is provided - that argument.
2. Otherwise, if a positive maximum number of guest processes is already defined in `/etc/mosix/maxguests`, then it is not changed.
3. Otherwise, if a former call was made to `mosctl closemulti` – the maximum number of guest processes before that call.
4. Otherwise, 8 times the number of processors.

`mosctl closemulti` sets that maximum to 0 - preventing processes from other clusters to run on this node.

`mosctl whois`, depending on its argument, converts host-names and IP addresses to node numbers or vice-versa.

`mosctl status` outputs useful and user-friendly information about a given node. When the last argument is omitted, the information is about the local node.

`mosctl localstatus` is like `status`, but adds more information that is only available locally.

`mosctl rstatus` output raw information about a given node. When the last argument is omitted, the information is about the local node. This information consists of 10 integers:

1. `status`: a bit-map, where bits have the following meaning:

1	The node is currently part of our MOSIX configuration.
2	Information is available about the node.
4	The node is in "stay" mode (see above).
8	The node is in "lstay" mode (see above).
16	The node is in "block" mode (see above).
32	The node may accept processes from here. Reasons for this bit to NOT be set include: * We do not appear in that node's map. * That node is configured to block migration of processes from us. * Our configuration does not allow sending processes to that node. * That node is currently running higher-priority MOSIX processes. * That node is currently running MOSIX processes with the same priority as our processes, but is not in our cluster and already reached its maximum number of allowed guest-processes. * That node is blocked.
512	The information is not too old.
1024	The node prefers processes from here over its current guests.

2. `load`: a value of 100 represents a standard load unit.
3. `availability`: The lower the value the more available that node is: in the extremes, 65535 means that the node is available to all while 0 means that generally it is only available for processes from its own cluster.
4. `speed`: a value of 10000 represents a standard processor (Pentium-IV at 3GHz).
5. `ncpus`: number of processors.
6. `frozen`: number of frozen processes.
7. `available memory`: in pages.
8. `total memory`: in pages.
9. `free swap-space`: in 0.1GB units.
10. `total seap-space` in 0.1GB units.

11. `privileged memory`: in pages - pages that are currently taken by less privileged guests, but could be used by clusters of higher privilege (including this node when "1024" is included in the `status` above).
12. `number of processes`: only MOSIX processes are counted and this count could differ from the load because it includes inactive processes.

SEE ALSO

`mosix`(7).

NAME

MOSSETPE - Configure the local cluster

SYNOPSIS

mossetpe [-m mapfile] [-p our.ip.x.y]] [-A]
mossetpe -[r|R]
mossetpe -off [-A]

DESCRIPTION

Mossetpe modifies the MOSIX node configuration.

The -m argument may be used to override the cluster's map file (/etc/mosix/mosix.map - See mosix(7)).

The -p argument may be used to override the local node's IP address (otherwise taken from /etc/mosix/mosip or by calling ifconfig(8).

Normally, mossetpe waits until all guest processes that must not run here according to the new configuration are sent away and all processes originating from this home-node that run where they should not according to the new configuration, migrate elsewhere. The -A argument causes mossetpe to return without waiting for this to happen.
(but note that mossetpe could still delay if a previous call to mossetpe without the '-A' argument is still waiting for the above)

mossetpe -r lists all nodes currently in the local cluster.
mossetpe -R lists all nodes currently in the multi-cluster private cloud.
mossetpe -off disables MOSIX.

All users can read the MOSIX configuration, but only the Super-User can modify it.

SEE ALSO

mosix(7).

NAME

MOS_IN_JOB - Configure MOSIX at the start/end of an external job

SYNOPSIS

mos_in_job [-A] {world-file} [own-cluster]

DESCRIPTION

mos_in_job configures MOSIX to run in either of two modes: as part of a temporary logical cluster; or standing alone at the service of other clusters. Typically, the temporary cluster would be allocated for a particular 'job' by an external package.

The file {world-file} describes which nodes/computers should be configured in the multi-cluster. Each line of {world-file} has a host-name or an IP address, optionally followed (space-separated) by the number of hosts with consecutive IP addresses that should be configured (the default is 1). The local node must be included in that file.

When provided, {own-cluster} is a comma-separated list of nodes that should form the local cluster (the local node must be included in that list). The local cluster is then configured accordingly and all other nodes that appear in {world-file} form another cluster, to which local processes can be migrated, but from which no guest processes are accepted.

When the {own-cluster} parameter is not provided, the local node is configured as a cluster on its own and all other nodes that appear in {world-file} are configured as another cluster, from which guest processes may arrive, but to which no local processes may go.

The [-A] argument causes mos_in_job to return immediately after changing the configuration - otherwise mos_in_job waits until all guest processes that must no longer run on the local node according to the updated configuration, are expelled; and all local processes that run where they should not according to the updated configuration, are migrated elsewhere.

SEE ALSO

mossetpe(1), mosix(7).