IST-FP6-034286 SORMA

D5.4

Market-based Grid OS & System Manual

Elicitation of requirements and resulting architecture

Contractual Date of Delivery to the CEC:	31 January 2009
Actual Date of Delivery to the CEC:	31 January 2009
Author(s):	**HUJI**
Workpackage:	5
Security:	public
Nature:	final
Version:	0.1
Total number of pages:	51

Abstract:
This document presents the implementation of the MOSIX-SORMA market based scheduling system on top of the MOSIX Organizational Grid system. We present a detailed description of the implemented components of the systems as well as modifications done to the original MOSIX system. A novel market scheduling simulation environment is also described. This simulator is a viable tool in researching the behavior of market scheduling algorithms in large systems. This document also provides the installation and configuration guide of the MOSIX-SORMA system, this is a step by step guide for installing the system from scratch. Another guide provided is the user guide of the system, which explain to normal MOSIX users how to use the new tools provided by the MOSIX-SORMA package to submit jobs to the market. At the end of this document we provide a description of the CD image containing the MOSIX-SORMA package. The system is delivered in 3 forms: A ready to use virtual machine disk image, a ready to install tar-ball of the system, and a snapshot of the system source code.

Keyword list: (optional)

Revision Table:

Revision & Date	Author	Comment
0.1 – 13/2/2009	Lior Amar (HUJI)	First version (moved from wiki to doc)
0.2 – 22/2/2009	Lior Amar (HUJI)	Adding the documentation of the componenets
0.3 – 23/2/2009	Tal Maoz (HUJI)	Testing and proofing
0.4 – 26/2/2009	Lior Amar (HUJI)	Description of MOSIX_SORMA modifications and added componenets
0.5 – 1/3/2009	Lior Amar (HUJI)	Final additions and fixups
0.6 – 2/3/2009	Tal Maoz (HUJI)	Proofing

Table 1: Document History

ACTION LIST – updated to 09/02/2009:

Table of Contents:

Table of Figures:

1. Introduction

Deliverable format description (from the description of work):

D 5.4: Market-based Grid OS & System Manual (Prototype/Report)**: "Description of the implementation and modifications undertaken, detailed definition of prototypical system components and manual are available"**

<u>This deliverable (5.4) includes the following:</u>

- Describes the MOSIX-SORMA system implementation (and modifications) and presents a detailed definition of the system components
  - Provides description of the market simulator environment (both architecture and implementation).
- Manuals:
  - Provides system administrator installation and configuration manual
  - Provides user (both resource buyers and resource providers) manuals
  - Provides market simulator manuals (in future)
- Software:
  - A pre-installed virtual machine with MOSIX and MOSIX-SORMA components installed there within.
  - Final MOSIX-SORMA package (to be installed on a clean node (binary form))
  - Source tree of all the MOSIX-SORMA components

1.1. Structure of this Document

This document is structured in the following way:

- **Chapter 0** provides relevant information about this document and its relationship within Work-package 5, and provides a high level overview of the MOSIX-SORMA system and its main end-users.
- **Chapter 2** describes the MOSIX-SORMA final prototypes' architecture and implementation undertaken.
  - Presents a detailed implementation view of the MOSIX-SORMA system.
  - Provides information about the market simulator system, which is a viable part of the MOSIX-SORMA system. This simulator allows researchers to test and develop market based scheduling algorithms and test them using real workloads.
- **Chapter 3** provides the system-administrator's manual of the system: how to install and configure the market-manager, resource-provider and user client components.
- **Chapter 4** provides the user's manuals of the system, both for resource buyers and resource providers.
- **Chapter 5** provides the user manual of the market simulator environment
- **Chapter 6** provides description of the MOSIX-SORMA delivered software

1.2. Explanatory Notes about WP5 Deliverables: D5.2, D5.3, D5.4

In order to avoid misunderstandings or ambiguities, it is useful to remember which deliverable is going to deliver the Final software prototypes and related documentation

- **D 5.2: Economic Grid middleware (Prototype/Report**): **"Description and implementation of the prototypical economic Grid middleware"**

This contains technical details about software components to be installed at **Providers** side. This deliverable IS NOT going to deliver such components' user guides, which will be delivered in D5.3

- **D 5.3: Integrated SORMA system & System Manual (Prototype/Report): "Description of the integrations undertaken, detailed definition of all prototypical system components and manuals are available."**

This deliverable includes a technical overview of the integrated SORMA system (technical architecture), the role of the components, and user's manuals for ALL SORMA components. This deliverable also includes a software prototype of the entire SORMA market and user agents.

- **D 5.4: Market-based Grid OS & System Manual (Prototype/Report): "Description of the implementation and modifications undertaken, detailed definition of prototypical system components and manuals are available"**

This deliverable includes the software prototypes and user manuals of the MOSIX-SORMA package. It contains a comprehensive description of the new software components developed as well as description of the modifications done to the MOSIX system in order to support the MOSIX-SORMA market based scheduling. This deliverable also includes the software of the MOSIX-SORMA package.

1.3. Overview of the MOSIX-SORMA System

The MOSIX-SORMA system was developed in order to offer alternative scheduling to the traditional load-balancing scheduling offered by MOSIX. In particular, the MOSIX-SORMA system provides an on-line preemptive market based scheduling for a MOSIX system. Figure 1-1 (**as presented in D2.2**) presents a conceptual view of the MOSIX-SORMA system integration within the main SORMA system. In the figure, the three main layers are presented:

- Application and Resource layer – representing users and resource owners
- Agent layer – which come as an intermediate layer between the users and the actual market/system
- Market layer – which performs the actual scheduling of resources to clients.

In the Mosix context, the market layer can be composed from two market types: A *future market* and a *spot market*. In the integrated view, the futures market will be responsible for the scheduling of resources where there is a need for reservation in advance, comprising hard deadlines while the spot market can be used for scenarios where no reservation is needed and *best effort* scheduling is good enough. Were by best effort we mean that the market will run the users jobs whenever it is possible. For example if user A submits a job with price 10 and user B submits a job with price 20 then user A does can not expect the market to run her processes instead of user B. But when user B finishes then user A can expect the market to run her processes.

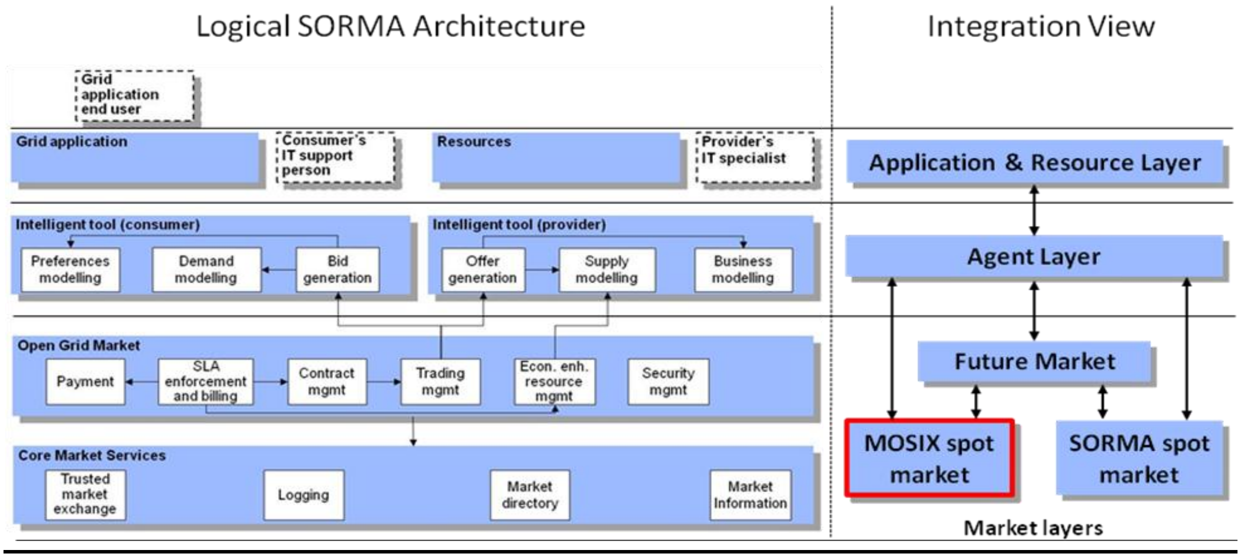


Figure 1-1 A comparison between the SORMA and MOSIX-SORMA conceptual architecture

The MOSIX-SORMA implementation, presented in this deliverable, provides an independent ***preemptive*** spot market. As we previously stated (in D2.2), the agent layer and the future market can be the ones delivered by the main SORMA system (with the necessary integration).

1.3.1. Market Operator

MOSIX-SORMA provides an ***online preemptive spot market*** to perform the brokerage of resources over the MOSIX organizational Grid, match the users' and brokers' requests and put them in contact. The market based scheduling is performed continuously allowing the market to respond swiftly to new demands or changing supply scenarios. The market operates by collecting consumer requests via the client component and obtaining information about participating recourses via the MOSIX information system. This information is relayed to a market solver component which computes the allocation of jobs to providers. The allocation is than carried on by the market which forward the details to the corresponding clients.

Using the MOSIX-SORMA system it is possible to easily switch between different market-based schedulers (and even non-market based schedulers). The MOSIX-SORMA system is supplied with several market-based schedulers and in particular one preemptive market-based scheduler, which is the default scheduler in the MOSIX-SORMA system. The flexibility to modify the market scheduler allows users of the system to develop their own scheduling policies, which might be more suitable to their installation.

During the development of the market based scheduling algorithm a market simulator was built. This simulator is also part of the MOSIX-SORMA package allowing researches to benefit from the developed simulation environment in further researches. The simulator can (and should) be used when a new scheduling algorithm is developed to verify the algorithm before deploying it on a real system.

MOSIX-SORMA mechanisms ensure a more rigorous allocation of resources and self-organizing resource management. A simple management interface for resources is provided; the surveillance of the running nodes is made through a shared information gathering layer to simplify the market task of discovering and keeping track of nodes and their status.

1.3.2. Resource Provider

The MOSIX-SORMA system enables resource providers to join a MOSIX-SORMA market in which the physical resources can be offered to clients in exchanged to payment. The system allows resource owners to regain full control over their resources immediately when needed. This can be achieved thanks to the usage of preemptive process migration, which allows a running process to be migrated out of the provider when the circumstances demand. The system is suitable for both dedicated resources (that are always in the market) and non-dedicated resources with an unknown time of participation.

MOSIX-SORMA provides the means for the resource providers to join the market, and leave the market. The resource fabrics register their presence in the MOSIX information infrastructure; the market collects the messages (about available resources and their status) sent by the resource fabric node, performs an allocation decision, and communicates it back to the user agent and to the provider;

MOSIX-SORMA provides resource owners with economically sound sustainable and customizable business models:

- methods and tools to express the business model of the resource owners
- methods and tools for capturing users' reserve prices for resources
- methods and tools for estimating and monitoring the quality of the resource management.

1.3.3. Consumer

MOSIX-SORMA offers a platform to the consumers of the MOSIX organizational Grid for economically efficient, market-based, identification and acquisition of needed resources. Users need not know about the availability and status of resources in the system. This is handled transparently by the MOSIX-SORMA system requiring users to only submit economy related parameters such as price and budget.

MOSIX-SORMA provides the means for submitting jobs and monitoring them. The users no longer have to be concerned with which resources their jobs are consuming as long as they are within the scope of the supplied economic parameters. The client agent acts on behave of the user and contacts the market to obtain resources. The client can modify the reported parameters (as long as the do not break the original parameters) to optimize the obtained performance.

The MOSIX-SORMA platform provides resource users with tools to access the spot market:

- methods and tools to submit a new job with its economical parameters
- methods and tools to estimate and monitor the quality of the resource management.

1.4. Note on Security

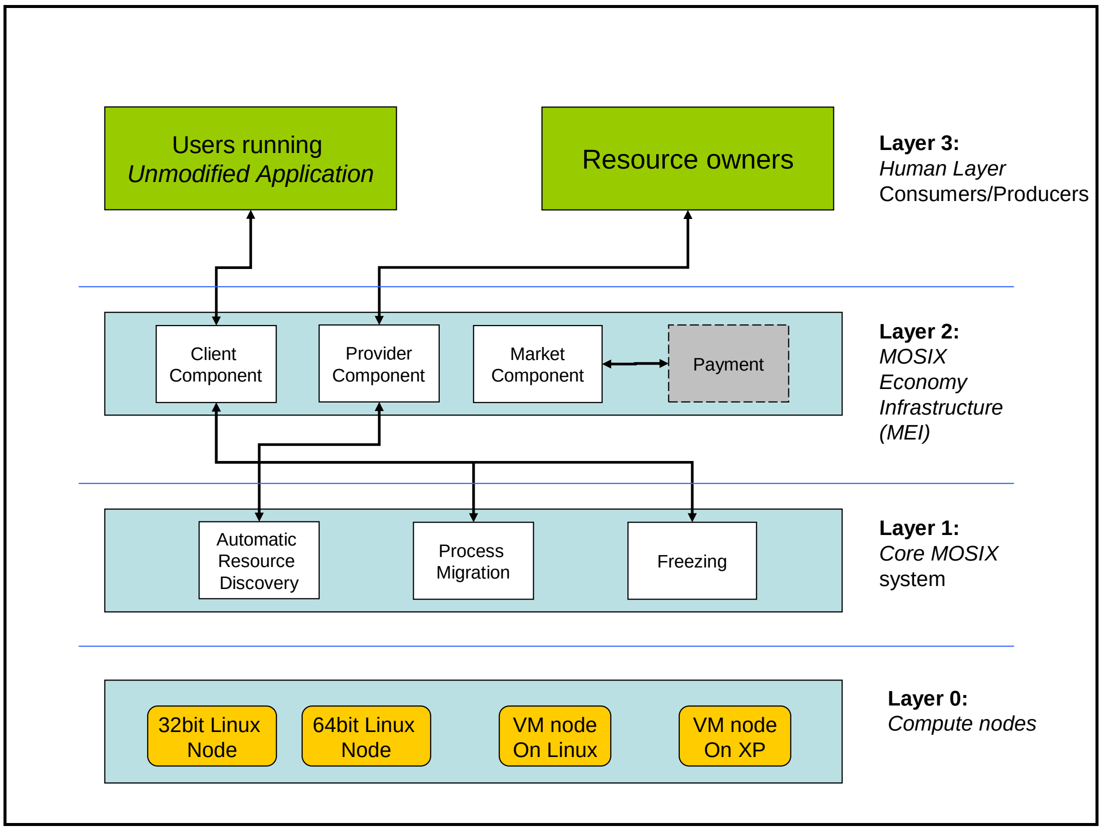
Since the MOSIX-SORMA system comes as an enhancement to the MOSIX system scheduling policy, the ***security model used is the one assumed by the MOSIX system***. In the MOSIX system, once a user logs in to a client machine, he can submit processes from that machine. This means that the administrator of the client machine(s) is responsible for setting the appropriate entrance control. It is also assumed that the administrator of the client machine is reliable and there is no need to authenticate the remote machine (assuming a secure network). From the provider's point of view, it is assumed that the administrator of the provider machine is reliable and that the provider machine can be trusted. **All those assumptions are part of the MOSIX organizational Grid assumptions**.

2. Description of the MOSIX-SORMA System

This chapter presents a detailed description of the prototype implementation of the MOSIX-SORMA system for performing market based scheduling in a MOSIX system.

The purpose of the MOSIX-SORMA system is to create an economically aware scheduling mechanism on top of the existing MOSIX system. The MOSIX-SORMA system provides a spot market for selling/buying CPU resources without guarantees. One of the important issues in the MOSIX-SORMA system is its responsiveness. The MOSIX-SORMA market is designed to perform allocation of jobs is a very short time (seconds). This is necessary in order to support the scheduling of large numbers of processes (even with short runtimes).

Figure 2-1 presents the component architecture of the MOSIX-SORMA system. Layer 0 represents the physical resources used. Those resources can be either 32bit Linux machine, 64bit Linux machines or virtual machines (either 32 or 64 bit Linux). Layer 1 presents the core MOSIX subsystems used by the MOSIX-SORMA system. Those subsystems of MOSIX were modified in order to interface with the MOSIX-SORMA componennts (presented in layer 2). Layer 2, shows the important componenets of the MOSIX-SORMA system (which is also referenced ad MEI (MOSIX Economy Infrastructure)): market, provider and client components.



Users running *Unmodified Application*
Resource owners
Layer 3: *Human Layer* Consumers/Producers
Client Component
Provider Component
Market Component
Payment
Layer 2: *MOSIX Economy Infrastructure (MEI)*
Automatic Resource Discovery
Process Migration
Freezing
Layer 1: *Core MOSIX* system
32bit Linux Node
64bit Linux Node
VM node On Linux
VM node On XP
Layer 0: *Compute nodes*

Figure 2-1 MOSIX-SORMA architecture view

2.1. Enhanced Core MOSIX Subsystems

The relevant features of MOSIX to the work presented in this paper are the information dissemination, the process migration and the freezing mechanisms.

2.1.1. Information Dissemination Service

Resource discovery in MOSIX is performed by an on-line information dissemination algorithm[1], providing each node with the latest information about the availability and state of Grid resources. The dissemination is based on a randomized gossip algorithm, in which each node regularly monitors the state of its resources, including the CPU speed, current load, free and used memory, etc. This information, along with similar information that has been recently received by that node, is routinely sent to a randomly chosen node, where a higher probability is given to choosing target nodes in the local cluster.

The outcome of this scheme is that each node maintains a local information-vector with information about all active nodes in the local cluster and the Grid. Any client requiring information about cluster or Grid nodes can simply access the local node's information-vector and use the stored information. Information about newly available resources (e.g. nodes that have just joined the Grid), is gradually disseminated across the active nodes, while information about nodes in disconnected clusters is quickly phased out.

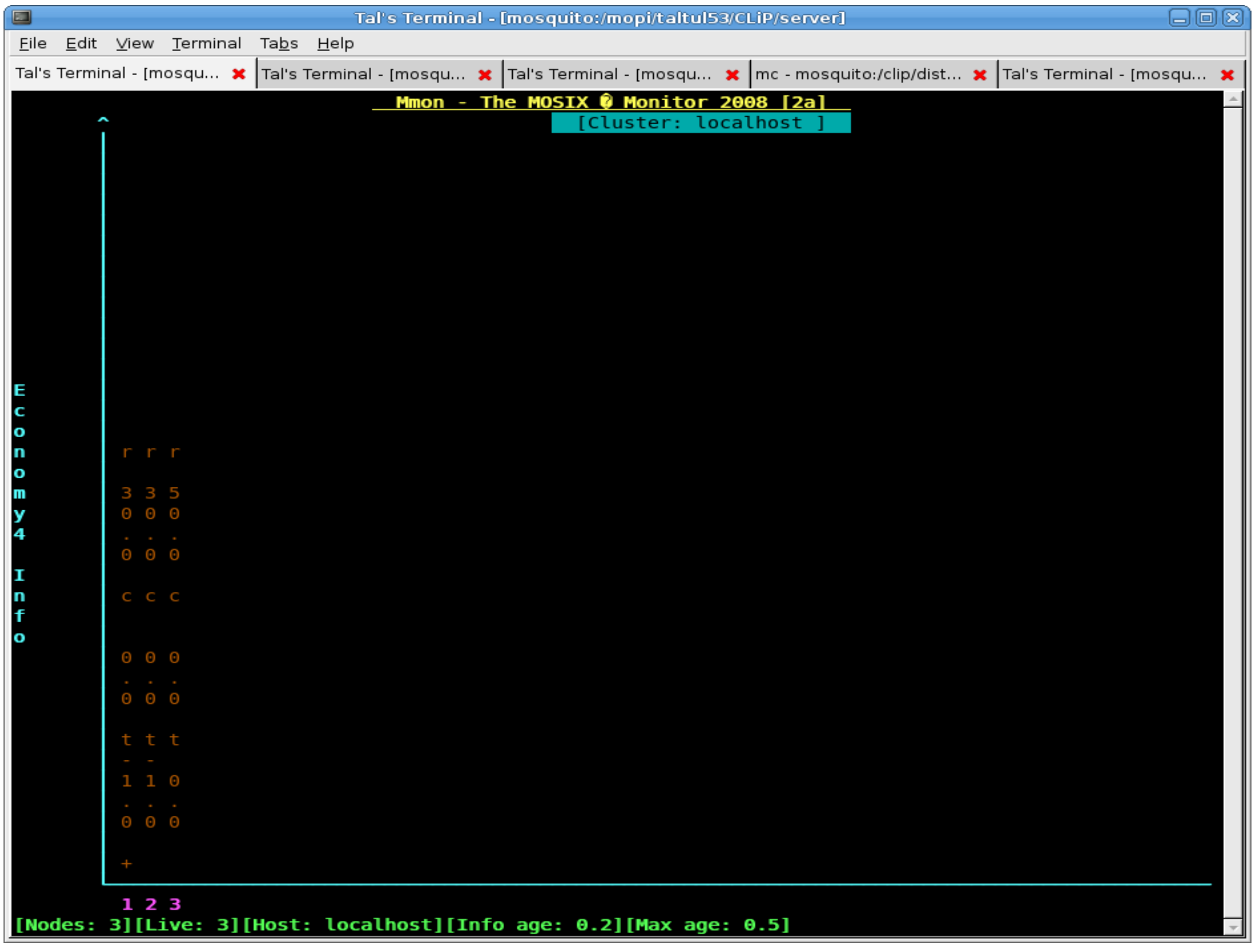
The MOSIX-SORMA system uses the information subsystem for several purposes. First, the market component uses this subsystem to detect active provider nodes. Each provider node adds economic information to the regular information disseminated, including the node's market status. Thus, when the market status indicates it is willing to be a part of the market, the market manager takes this into account and will send jobs to this provider when submitted,

The disseminated information is also accessible for other providers that may use this info to change their own economic parameters (such as price) according to the current status of the market. Clients can also access the provider's information to obtain a current view of the market status, allowing them to determine their own economic parameters.

The information dissemination system was enhanced to disseminate the economic parameters of the providers. A specific interface was created to allow the provider daemon to update the economic parameters shared with other nodes (including the market). This way, the market daemon (as well as other nodes) can obtain the economically based information directly from the information system.

The monitoring tool of the information system (i.e. mmon) was modified to include screens showing the current economic status of the system nodes. Those screens enable users of the system (and developers) to get an online view of the current status of the cluster. For example, Figure 2-2 shows a screenshot of the mmon program. In the figure we can see that three providers are detected (pink labels of 1, 2 and 3) and for each such provider we can see its reservation price. (30, 30 and 50 accordingly). The mmon has other screens showing the current price (if the provider is currently being used) and more.

[1] Lior Amar, Amnon Barak, Zvi Drezner and Michael Okun, Randomized Gossip Algorithms for Maintaining a Distributed Bulletin Board with Guaranteed Age Properties, Accepted for publication in journal of Cuncurrancy and Computation Practice and Experience


Tal's Terminal - [mosquito:/mopi/taltul53/CLiP/server]
File Edit View Terminal Tabs Help
Tal's Terminal - [mosqu...
Tal's Terminal - [mosqu...
Tal's Terminal - [mosqu...
mc - mosquito:/clip/dist...
Tal's Terminal - [mosqu...

```
                    Mmon - The MOSIX Monitor 2008 [2a]
                                 [Cluster: localhost ]
      ^
      |
      |
E     |
c     |
o     |
n     |  r r r
o     |
m     |  3 3 5
y     |  0 0 0
4     |  . . .
      |  0 0 0
I     |
n     |  c c c
f     |
o     |
      |  0 0 0
      |  . . .
      |  0 0 0
      |
      |  t t t
      |  - -
      |  1 1 0
      |  . . .
      |  0 0 0
      |
      |  +
      +------------------------------------------------------------
         1 2 3
[Nodes: 3][Live: 3][Host: localhost][Info age: 0.2][Max age: 0.5]
```

Figure 2-2 Mmon showing economical information

2.1.2. Preemptive Process Migration and Freezing Support

MOSIX supports cluster and Grid-wide (preemptive) process migration [2]. Process migration can be done either automatically or manually. The migration itself amounts to copying the memory image of the process and setting its run-time environment. In MOSIX, the node where the process was initially started is referred to as the process' ***home-node***. After a process is migrated out of its home-node, all the system-calls of that process are relayed to and processed at the home-node. The resulting effect is that there is no need to copy files or libraries to the remote nodes and a migrated process can be controlled from its home-node as if it was running there.

The preemptive process migration is a ***key feature*** for our preemptive spot market. Using this feature the market algorithm can reallocate jobs to better (either cheaper or faster) providers once such providers are detected. In addition, using this feature the MOSIX-SORMA spot market can trade all resource and not only idle resource, by migrating a lower valued job out of a provider in favor of a higher valued one. During the development of the MOSIX-SORM system we performed extensive research on the benefits of migration for online Grid markets[3].

In a dynamic Grid environment, the availability of resources may change over time, e.g., when clusters are connected or disconnected from the Grid. In MOSIX, guest processes running on nodes

[2] Barak A., Shiloh A. and Amar L., An Organizational Grid of Federated MOSIX Clusters. Proc. 5-th IEEE International Symposium on Cluster Computing and the Grid (CCGrid05), Cardiff, May 2005
[3] Lior Amar, Ahuva Mu'alem and Jochen Stosser, On the Importance of Migration for Fairness in Online Grid Markets, Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid 2008)

that are about to be disconnected are migrated to other nodes or back to their home-node. In the latter case, there may not be enough available memory at the home-node to receive the returning processes.

To prevent the loss of running processes, MOSIX has a freezing mechanism that can take any running MOSIX process, suspend it, and store its memory image in a regular file (on any accessible file-system). This mechanism can be activated automatically when high load is detected, or manually by request. The freezing mechanism ensures that a large number of processes can be handled without exhausting CPU and memory resources.

The MOSIX-SORMA system relies on the manual freezing mechanism of MOSIX to freeze already running jobs which the market mechanism could not allocate to any provider. Without this feature the MOSIX-SORMA could not trade already used resources.

2.1.2.1. Modifications of the Process Migration and Freezing Subsystem

In order to enable the client component to obtain important information about currently running jobs, a file called "/var/.mosinfo/{pid} is created for each MOSIX (mosrun) process. This file contains the following newly exported information about the process' status:

Field Name	Description
SonPID	Process ID of the actual job
Frozen at	The location at which the mosrun process is frozen (if it is, indeed, frozen)
Frozen Pages	Size of the frozen job.
Where	The location at which the job is currently running
Migration	Number of migrations performed by the process
Failed Migrations	Number of failed migrations attempts

For example, the field "Frozen Pages" was added to the system during our pilot study following a case where several large memory jobs were being unfrozen at the same time. In such a case, the local memory of the node is not sufficient to accumulate all the unfrozen jobs, and it is necessary to perform the unfreezing according to the currently available free memory size.

A new freezing state was added to the MOSIX system stating that a job is frozen by an external subsystem. This is an addition to the previously existing freeze states: *manual*, *automatic* and *evict*.

2.2. Client Component Detailed Description and Implementation

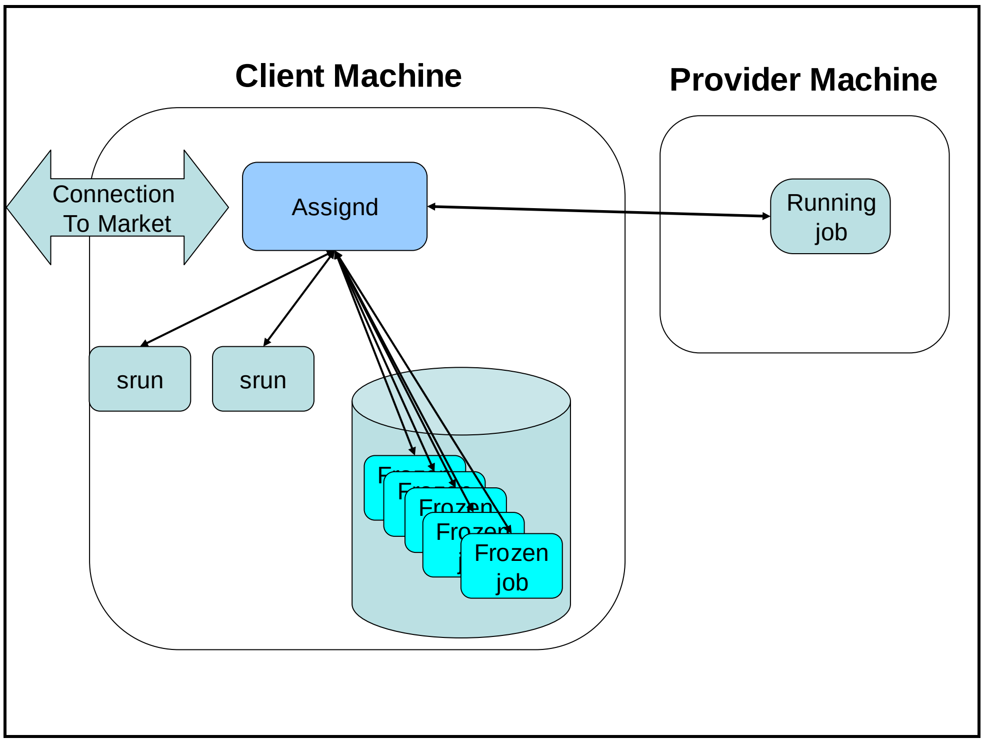


Figure 2-3 Client component schematic view

This section describes the client component and its implementation. The client component is responsible for job submission and for enforcing the allocation of jobs to providers. This component is currently composed of 2 programs: the *srun* program used to submit jobs to the system, and the *assignd* daemon. A schematic view of this component is depicted in Figure 2-3.

2.2.1. The *srun* Program

srun submits a migratable job to be run under the supervision of a MOSIX-SORMA market component. The resulting job will be started where and when the market component decides, and may occasionally be migrated or suspended according to the MOSIX-SORMA economic considerations.

Once an *srun* program is launched, it connects to a daemon, called *assignd*, on the local host (see Figure 2-3 and the section about the *assign* daemon below) and waits for instructions. When the *assignd* daemon allows *srun* to continue (and specifies a location for it to run at) *srun* launches a mosrun job with the appropriate parameters and exits. From that point forward, the *assignd* daemon will no longer need the *srun* process, and will monitor the mosrun processes directly.

The following economic parameters are passed to SORMA, of which only '–p' (or --maxprice) is mandatory:

-p {$$} --maxprice={$$}	The maximum amount the user is willing to pay per hour (currency is not specified and must be agreed with SORMA).
-b {$$} --budget={$$}	The maximum total amount the user is willing to pay for running the job (currency is not specified and must be agreed with SORMA).

-f {hours} --finish={hours}	The maximum time, in hours, allowed for this job to complete (possibly including a decimal fraction).
-m {mb} --memory={mb}	The amount of memory in Megabytes the job is expected to use.
-s {strategy_name} --strategy={strategy_name}	An economic strategy to apply to this job. This may be any string of up to 31 characters, but must not include the quote character (").

For example, to submit a job with a maximal price of 40 and a memory requirement of 180MB, the user should run:

```
srun -p40 -m180 myprog myprog-args“
```

srun can also be used to modify the parameters of an already running job. This is done by submitting an *srun* job with the –J <job-id> flag. In this case, the *job-id* parameter specifies the job-id of a currently running job.

2.2.2. The *assignd* Daemon

The *assignd* daemon is responsible for managing all the economy-aware processes in each client node. The *assignd* daemon keeps track of all the *srun* processes and reports to the market manager about newborn *srun* processes and about finished *srun* processes.
The *assignd* daemon is also responsible for receiving instructions regarding the allocation of *srun* processes from the market manager. Once such instructions arrive, the *assignd* daemon uses the underlying MOSIX system to enforce the market decision. For example, if the central market instructs the *assignd* daemon to move a given *srun* processes from one provider to another, the *assignd* daemon uses the manual migration capabilities of MOSIX to send the *srun* process to the new location. If, on the other hand, the *assignd* daemon is ordered to suspend a process then the process is frozen to the local disk using the MOSIX freezing mechanism.

The *assignd* daemon assigns jobs that were generated by *srun* according to instructions received from a market component. When *srun* jobs are launched, they first ask the *assignd* daemon where to start: The *assignd* daemon provides a unique job-ID to each new job, relays its parameters to the market component, waits until the market component decides to start the job, and then tells the *srun* job where to start running. The job then uses the MOSIX submission mechanism to start its execution on the specified node.

The *assignd* daemon also receives instructions from the market component to re-assign, suspend, resume or abort existing *srun* jobs. It interfaces with MOSIX to make sure that the instructions are followed and informs the market component about the outcome. Suspension of jobs uses the freezing capabilities of the underling MOSIX system, which was enhanced to provide extra information about suspended jobs to the *assignd* daemon. The *assignd* daemon also relays job parameter change requests to the market component regarding running jobs.

2.2.3. Protocol with market manager

In this section we specify the protocol between the *assignd* daemon and the MOSIX-SORMA market component. All the messages are XML strings.

Connect message:
The first message sent by the *assignd* daemon is a connection message:

```
<client type="assignd" />
```

New job message:
Each time a new job (*srun*) is submitted, the following message is sent to the market component.

```
<job id="{job-number}" >
        <strategy> "name of strategy" </strategy>
        <mem> {number}MB </mem>
        <finish> {hours} </finish>
        <uid> {user-number} </uid>
        <max-pay> {price-value} </max-pay>
        <max-budget> {price-value} </max-budget>
      </job>
```

- The *job-id* attribute specifies a unique identifier for the job in the current host. This means that the combination of an *assignd* daemon's host IP and an *srun* job ID is a unique combination in the system.
- The *strategy* field is a free text string allowing the client to specify a possible behavior strategy
- The *mem* field specifies the maximum amount of memory this job is going to use. This is important so the job will always be allocated to nodes that have enough free memory.
- The *finish* field specifies a possible deadline/runtime that the job may have. This is useful for a possible integration of the MOSIX-SORMA spot market with an external futures market
- The *uid* field specifies the user ID of the job's submitter. This is important for correctly maintaining the budget and payment information of the job.
- The *max-pay* field specifies the maximum amount the client is willing to pay for running the job in units of currency per CPU hour.
- The *max-budget* field specifies the maximum budget this job has. It is forbidden to spend more currency then the specified budget.

Broken connection:
If, for any reason, the TCP connection between the *assignd* daemon and SORMA is severed or the IP/port configuration in /etc/mosix/assign.conf changes (it is re-checked once every minute), then the *assignd* daemon will attempt to reconnect to SORMA. Once reconnected, it (re)sends to SORMA a report about all existing jobs. The format of the report is per job, as above, for new jobs, except that the reports for jobs that are already running contain an extra line with the current IP address where they run:

```
      <where> {a.b.c.d} </where>
```

Changing job parameters:
The *assignd* daemon reports changes to job-parameters in the same way as new jobs, except that the first line of the report changes into:

```
        <job id="{job-number}" change="1" >
```

Managing job Allocation:
To start, restart or migrate a job, SORMA sends the *assignd* daemon a message in the following format:

```
        <assign-job id="{job-number}">
            <status> run </status>
            <provider> a.b.c.d </provider>
        </assign-job>
```

When all processes of that job have been migrated to (or started/or restarted on) the given IP address "a.b.c.d",the *assignd* daemon replies with:

```
        <job-status id="job-number"> migdone </job-status>
```

To suspend a job, SORMA sends the *assignd* daemon a message in the following format:

```
        <assign-job id="{job-number}">
            <status> suspend </status>
        </assign-job>
```

When all processes of a job have been suspended, the *assignd* daemon replies with:

```
      <job-status if="job-number"> freezedone </job-status>
```

To abort a job, SORMA sends the *assignd* a message in the following format:

```
        <assign-job id="{job-number}">
            <status> abort </status>
        </assign-job>
```

When the job is terminated, the *assignd* daemon replies with:

```
        <job-status id="{job-number}"> finished </job-status>
```

The above message is also sent when a job completes by itself.

2.2.4. Job Suspension

Job suspension is implemented using the MOSIX freezing mechanism. Processes of frozen jobs are shown by mosps(1) as "preempted" (the letter 'P' in the "FRZ" field). When the market component requests a restart of suspended processes, they are not necessarily unfrozen immediately - this depends on the following two flags which can be passed to the *assignd* daemon upon startup:

- When the -u flag is given, processes start to unfreeze only when there is sufficient memory to accommodate them in the main-memory of the local node (which is also the home-node of all *srun* jobs). If several jobs are restarted simultaneously, then only as many as can fit in the main-memory are unfrozen at any given time.

- When the -o flag is given, only one process is unfrozen at a time. Further processes start to unfreeze only when a previously unfrozen process has reached the designated node where it should run. (the –o flag overrides the -u flag).

- When neither -u or -o are given, all un-suspended processes are immediately unfrozen.

2.3. Provider Component Detailed Description and Implementation

This section describes the provider component and its prototypical implementation. This component is represented by a daemon running on each provider called *providerd*. The *providerd* daemon manages a MOSIX node participating in an alternate scheduling paradigm to the default MOSIX load balancing scheduling. It handles the economical properties of a MOSIX provider and handles connections from markets and clients. In addition a control tool named *pvdctl* allows resource owners to modify the node economic parameters during runtime.

2.3.1. The *Providerd* Daemon Structure

The *providerd* daemon is composed of the following subcomponents:

An economic properties manager component – handles connection from clients (such as *pvdctl*) allowing such clients to set/get the economic properties of the provider (e.g. the reservation price).

A time frame component – manages the definitions of time frames during which the provider can participates in the markets. Those time frames allow for the automatic *join-to-market* and *leave-market* events. For example, the owner of the resource can define that the provider will participate in the market every day between 8:00 and 22:00.

A negotiation component – responsible for getting scheduling decisions from the market component, or bid offers from the market in the case of a distributed market protocol. This component allows the provider to participate in different market types and run various market mechanisms. In our central spot market, this component is a very basic and simple one since all allocation decisions are made by the market manager. But in our distributed spot market, this component allows each provider to perform negotiations with potential clients to decide which client will use the provider.

A network manager component – responsible for handling all network based operations and managing the above components when a related communication event occurs. For example, when a connection from a client arrives, the message is forwarded to the economic properties manager.

2.3.2. *pvdctl* – a provider control tool

The provider component also includes a command line tool called *pvdctl***.** *pvdctl* is used to control a running *providerd* daemon. It enables the owner of the provider to modify/query the economical properties of the node without interrupting the *providerd* daemon.

Supported *pvdctl* commands are:

status	Print the status of the provider
set-price {new-price}	Set the reservation price (min-price) of the provider to new-price.
set-memory {memory-size}	Set the amount of memory the provider reports it is willing to sell. This number must be lower than the physical amount of memory.
set-speed {new-speed}	Sets the speed the provider is reporting to have to a different value. This is useful for testing installations with different speeds or to calibrate the speed of the nodes in a different way than the usual MOSIX method.

market-on	Open this provider node to accept jobs from the market.
market-off	Leave the market and expel all guest processes (if any).

2.4. Market Component Detailed Description and Implementation

The market component is responsible for connecting the client component and the provider component together and for managing the market. It receives information about available jobs from the client components and information about the status of each provider from the provider component. This information is used to compute a scheduling. The market component is implemented as a daemon called *economyd* which runs on a single node (or many in the case of a distributed market). Figure 2-4 presents a schematic view of the *economyd* daemon.

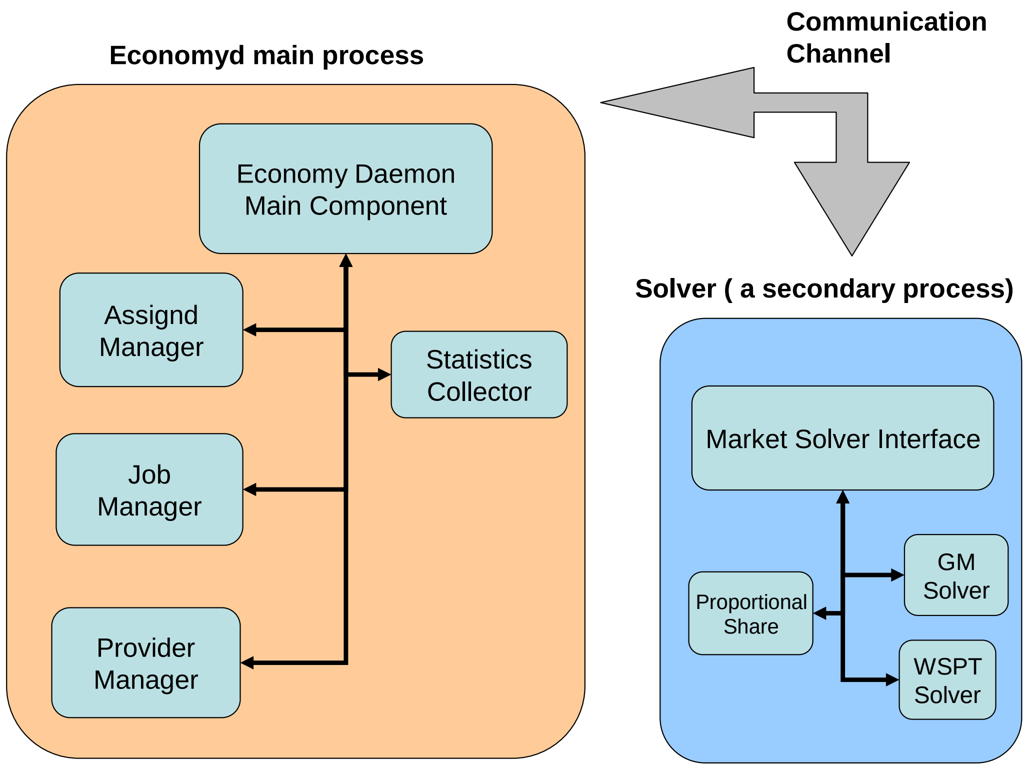


Figure 2-4 Market Component schematic view

Below we give a description of each of the *economyd* daemon's subcomponents:

job-manager – responsible for keeping track of all the active jobs in the market. This component registers new jobs when information arrives from the client, deletes a job when it is done, and sets its status when there is a change due to scheduling decision. This object is connected to the *assignd-manager* so when an allocation decision arrives to the job-manager it can request the appropriate *assignd* daemon to perform that allocation.

assignd-manager – responsible for communicating with all the *assignd* daemons in the system (which represent the clients). It receives information about new jobs, finished jobs and changes in the status of jobs (when migration or freezing is done). The information is communicated to the job-manager which takes care of the changes in a per job fashion. The *assignd-manager* also sends job scheduling instructions (such as run, freeze or migrate) to the *assignd* daemons in the system.

provider-manager – responsible for communicating with all the providers in the system. This component collects information about the current status of the providers: their availability (on/off), the reservation price, free memory and more. This information is kept updated by periodically querying the information system of MOSIX. The *provider-manager* component is also responsible for sending information to the providers in case of a scheduling decision (such as: job id 1234 is about to migrate to provider P1). All changes in the providers' status are quickly passed to the solver component so new allocation decisions can be computed rapidly.

market-manager – is a base object responsible providing an interface for running the chosen market algorithm and computing the allocation of jobs to providers. This component uses the above mentioned *job-manager*, *assignd-manager* and provider components and manages the operation of the market. The *market-manager* component provides a general implementation of a market and allows inheriting markets mechanisms to be easily used.

centralized-market - inherits the *market-manager* object and implements a centralized market place. In this market the allocation decision is performed in the *market-manager* and all participants are willing to cooperate. The *centralized-market* object creates a secondary process (as can be seen in Figure 2-4) which runs the solver framework (see below). The *centralized-market* communicates with the solver framework and passes new information (about jobs and providers) to this process. Once an allocation decision is made by the solver, the *centralized-market* updates the relevant jobs via the *job-manager*.

distributed-market – responsible for the implementation of an experimental distributed market protocol. This component also inherits from the *market-manager* component and adds the ability to negotiate with the providers (using a special distributed bidding protocol). When the market is operated in a distributed mode, each *market-manager* constantly competes against other market managers.

2.4.1. Market Solver Framework

The solver framework allows the market to run the market-based scheduling algorithm using a separate process. This has many advantages, such as being able to utilize a multi core node, and creating a complete separation between the management components and the scheduling components. As can be seen in Figure 2-4, the solver interface object provides a single interface for all possible solvers. This way the *economyd* daemon can easily switch to a different market scheduling algorithm, if necessary. We note, that the same solver framework is also used in the market simulator (see below).

Information about the current status of jobs and providers is constantly sent to the solver by the *centralized-market* component via the communication channel. This information is propagated to the specific market scheduling module used which, in turn, computes an allocation of jobs to providers as well as the payments. Once an allocation is computed, the solver sends it back to the main *economyd* daemon process. The implementation of the solver should be as efficient as possible since most of the time consumed by the market daemon is due to this component.

2.4.1.1. Available market scheduling modules

greedy-migration: This module implements the Greedy-Migration scheduling algorithm as presented in [4]. In this algorithm the jobs are sorted by their values and the machines by their speed. Then, the most valuable job is assigned to the fastest machine. This goes on until no providers are left or all jobs are assigned to providers. In case there are jobs which were not assigned to providers, those jobs are frozen by the MOSIX freezing mechanism.

[4] Lior Amar, Ahuva Mu'alem and Jochen Stosser, On the Importance of Migration for Fairness in Online Grid Markets, Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid 2008).

<u>proportional-share:</u> This module implements a proportional share scheduling algorithm, in which each user receive a share of resources proportional to the sum of values of all its jobs. This is an experimental module and is not the default of the MOSIX-SORMA system.

2.4.1.2. Interface of the solver program

The solver program has a text based interface. Each line represents a command, and spaces are used as delimiters. The following commands are supported by the solver program (for all types of market algorithms):

info	Shows general information about the current status of the market solver
aj id, v, m, c, start, run, (uid)	Add a job to the market. The job has the following parameters. • *id* – A unique integer job id to use • *v* – The value of this job in terms of the maximal amount of currency units per CPU hour it is willing to pay • *m* – The maximal amount of memory this job is about to use • *c* – The number of CPUS (on the same node) this job requires • *start* – Start time of this job in integer units which are seconds since the market started to function • *run* – The run time of the job in seconds. Note that this parameter is not necessarily supplied by all systems. • *uid* – An optional uid value to represent the user this job belongs to. This is necessary for proportional share allocation mechanisms.
dj jobId	Delete a job from the solver. This is how the system signals the solve that the job is done
uj jobId val	Update the job's value parameter. This allows the system to modify jobs' values while running
jmd jobId	Signal the solver that the job finished its migration
jfd jobId	Signal the solver that the job finished its freezing (suspension)
ap id, r, m, c, s	Add a provider to the solver. The provider should have the following parameters: • *id* – A unique integer provider id • *r* – Reservation price of the provider, meaning the minimal price it is willing to accept a job for. • *m* – maximal amount of memory it is offering for jobs • *c* – number of CPUs it is offering for jobs • *s* – Start time (in seconds) this provider should join the solver
dp id	Delete the provider with *id* from the system. This is used to signal the solver framework that a provider is no longer available
up id, r	Update the provider reservation price, allowing providers to dynamically modify their preferences

go	Run the solver. This command asks the solver to calculate an allocation
alloc	Get pending allocations (as a result of a previous *go* command). The allocation is returned in a format of one job per line, where each line contains the id of the job and the id of the provider. A provider id od ‘0’ means that the job should be suspended.
payment	Get pending payments. Each allocation computed also has payments attached to it. Using this command the payments each job should perform are communicated.
time t	Set the solver time to *t* integer units from the beginning. This command allows the system to signal the solver about the amount of time passed
exit	Exit the solver program
lj	List all jobs currently active in the provider. This is used for testing purposes.
lp	List all currently active providers. Used for testing purposes
reset	Reset the market, clearing all jobs and providers. Used for testing
set-solver solver-name	Allows the replacement of the active solver algorithm. This function is for testing only.

2.5. Simulator Environment Detailed Description and Implementation

The market simulator framework was built in order to enable the development and comparison of new market based scheduling algorithms. The market simulator uses the same *market-solver* framework as described in the market component section. Thus, the exact code used by the simulator is also used by the real system. The market simulator also enables the use of real workload traces taken from production system. By doing this we can test our market based scheduling algorithm against complicated scenarios which are very hard to create in a pilot system (due to time and resource limitations).

The market simulator is very important for potential users of the MOSIX-SORMA system. Using the simulator it is easy to replay real data collected from the real system and simulate the behavior of the MOSIX-SORMA system (without really needing to install it). This way, potential users can better understand what the expected results are and can identify potential weaknesses of their local system. Also, if a potential user of the system would like a slightly modified scheduling algorithm, the market simulator is a very important testing tool for such a new algorithm before it is applied in the real system.

Figure 2-5 presents a schematic view of the market simulator framework. The core component of the framework is the simulator engine, which simulates the real system conditions to the solver framework. The simulation engine gets a providers file and a jobs files as its input. Those files can be created manually (e.g. when simulating a simple scenario) or automatically from a workload trace file. Once the input files are loaded, the simulator starts simulating the specified system. It uses the solver framework to access the specific market scheduling algorithm being used and asks this algorithm to compute an allocation of jobs to providers. This allocation is carried on by the simulation engine which simulates the running jobs. Once a job finishes it is removed from the system. When all jobs in the trace are done, the simulations engine creates an output file with statistics about the jobs. The output analyzer tool can then be used to produce relevant graphs from the output file.

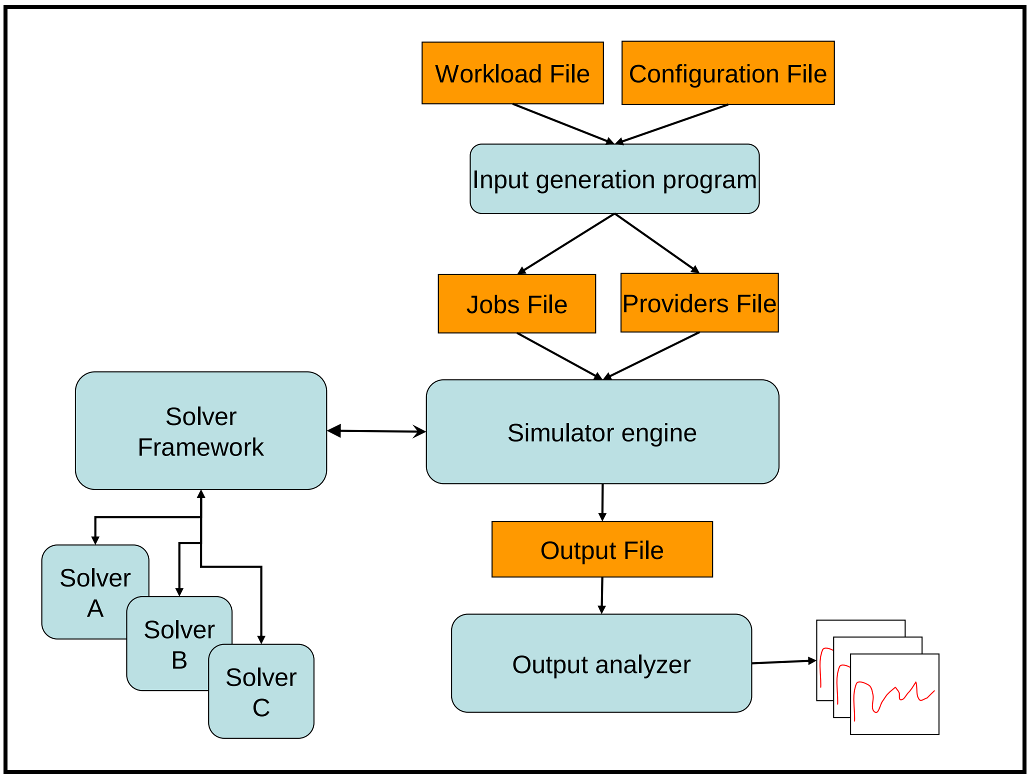


Figure 2-5 Market Simulator schematic view

2.5.1. Simulator Input

The simulator gets its input from 2 files. A provider file, which describes the providers present during the simulation, and a job file, which describes the stream of jobs during the simulation.

The provider file has the following format:

1. Comment lines starts with #
2. Empty lines are ignored.
3. Non empty lines have the following format:

ID	The id of the provider (must be unique for every line)
Start	The time this provider joins the market
End	The time this provider leaves the market
R	Reservation price of this provider
Mem	Memory size of the provider
Cpus	Number of CPUS
Speed	Speed of the provider, where a speed of 100 is the standard speed

Below you can see an example of a provider file containing the definition of 5 identical providers

```
# A test provider file
#Name    Start    End  R      Mem    Cpus    Speed
1              0       -1   10     1000         1        100
2              0       -1   10     1000         1        100
3              0       -1   10     1000         1        100
4              0       -1   10     1000         1        100
5              0       -1   10     1000         1        100
```

The job file has the following format:

1. Comment lines starts with #
2. Empty lines are ignored.
3. Non empty lines have the following format:

start	The time the job is available to the system
run	Time it takes to run the job on a standard (speed=100) provider
value	The value of the job
mem	Memory requirement of the job
cpus	Number of CPUs needed by the job

Below there is an example of a jobs file containing 7 jobs

```
#  start run   value    mem    cpus
1        0    50   24       100       1
2        0    50   23       100       1
3        0    50   22       100       1
4        0    50   21       100       1
5        0    50   20       100       1

# add a more expensive job
6        5    45   25       100       1

# add another more expensive job
7  10     40    26        100        1
```

2.5.1.1. Generating Input From Workload Traces

One of the main purposes of our market simulator is to enable the usage of workload traces taken from real systems. The workload trace files we are using are the one described at http://www.cs.huji.ac.il/labs/parallel/workload .

In order to generate a provider file and a jobs file from a workload trace file we supply the utility *gen-input-sim.pl*. Since the workload trace file does not contain the value of the jobs, this value is added artificially from a definition in a new configuration file. Also, since the characteristics of the providers are not a part of the workload file, the configuration file also contains the providers information.

The format of the configuration file is as follows:

Jobs related configuration parameters:

```

# To use a workload trace file or not
useSWF      = true;

# Name of the workload trace file
SWFFile     = real-logs/WHALE-2005-middle-memfix.swf.gz

# Starting day (in case we would like to take only part of the trace)
#swfStart   = 196d

# Ending day
#swfEnd     = 204d

```

```
# A parameters used to divide stat time of jobs (in order to make the workload more condence)
#swfDivStart = 2

# A parameters used to multiply the runtime of each job (to make runtimes longer)
#swfMulRuntime = 4

# How many jobs to take from the trace (-1 means take all jobs)
jobsNum       = -1

# Which value distribution to use for the value parameter (below is an example of a uniform
# distribution from 10 .. 60
valueDist      = uniform
valueDistArgs  = 10,60

# An example of a bimodal distribution
#valueDist      = bimodal
#valueDistArgs  = (80,30,15):(20,90,15)
```

Providers related configuration parameters:

```
# Number of providers to use
providersNum =  3072

# The speed distribution of providers (in this case fixed speed)
speedDist = fix
speedDistArgs = 100

# Reservation price distribution of providers (in this case a fixed distribution)
reservationDist = fix
reservationDistArgs = 1
```

Below is an example of generating the input files for the simulation from a configuration file:

```
gen-sim-imput –c configuration-file –j jobs.1 –p providers.1
```

The above example will generate a job file called jobs.1 and a provider file called providers.1 which are ready to be used by the simulator.

2.5.2. The Simulator Engine

Once the input files are ready it is time to invoke the simulator engine. This is done by running the *sim* command. The following example shows how this is done:

```
sim -j jobs.1 -p providers.1  -s mig –o output.1
```

In the above example, the *sim* program is activated with two input files, with the solver ‘mig’ and with output file ‘output.1’.

Once the simulator is launched it does the following:

- Read input file
- Create a list of jobs ordered by their arrival time to the system
- Create a list of providers ordered by their arrival time to the system
- Initialize the solver framework and set it to use the market algorithm chosen by the –s option (where the default is the greedy-migration algorithm)
- Start the time counter at t=0

At each cycle of the simulation the simulator does the following:

- Increment the time counter t=t+1
- Check if there are jobs that should be submitted to the simulation. If so, those jobs are moved from the initial job list and are added to an internal list of waiting jobs. The solver framework is notified about the arrival of new jobs
- Check if there are providers that should be registered in the system at that time. If there are such providers the solver framework is notified about the addition of new providers
- Check if there are running jobs that consumed enough CPU to finish. If so the solver framework is notified about the finished jobs
- Check if there are jobs that finished migrating or have become suspended and notify the solver framework
- Ask the solver framework to compute an allocation of jobs to providers, receive the allocation and implement it.

Each time an allocation is computed the simulator does the following:

- If the allocation is an allocation of a job not yet started (waiting) then this job is marked as running immediately with no delay
- If the allocation means a migration of the job from one node to another, the time it should take the migration to occur is calculated (based on the job's memory size and network bandwidth) and the job is kept in a "migrating" status for that time
- If the allocation means a suspension (freezing), then the time it should take this suspension to occur is calculated (based on the job's memory size and the disk speed) and the process is kept in a "being suspended" state until this time passes

For each completed job, the simulator computes statistics of its run and keeps them in a special memory contained buffer. Once all jobs finish running, the statistics are outputted to a single simulation output file.

2.5.3. Simulator Output

The simulator outputs a single output file at the end of the simulation. This output file may be composed of several parts where each part contains a different type of information. We will describe the basic statistics the simulator produces.

2.5.3.1. Simulation Summary

This part of the output file provides a full summary of the simulation. The example below shows the summary part of the output for running the simulations of the 'jobs.1' and 'providers.1' input files. We mark out comments in blue while the original output is in black:

```
Signal:              0                          Signals received during simulations
Solver:              mig                        Name of solver used
Steps:               96 (0 days)                Number of simulation steps performed
Wall clock:          0 sec                      Actual time it took to run the simulation
Providers:           5 (test_input/providers.1) Number of providers and the providers file
Total jobs:          7 (test_input/jobs.1)      Number of jobs and the job file
All jobs:            7
Mig speed:           100.00                     Migration speed used in MB/sec
Freeze speed:        100.00                     Freezing speed used in MB/sec

Results:
Avg runtime:         47.8571                    Average runtime of jobs
Avg slowdown:        1.2543                     Average slowdown computed over all jobs
Avg mig:             0.0000                     Average number of migrations per job
Avg migTime:         0.0000 (0)
Max concurrent mig:  0                          Number of cocurrent migrations occurred
Avg freeze:          0.5714                     Average number of freezing events per job
Avg migFreezeTime:   2.0000 (2)
Max concurrent frz:  2                          The maximal number of cunncurent freezing
WCT:                 9446.0000                  The Weighted Completion Time metric
WStayTime:           9487.0000                  The Weighted Stay Time metric
WBSD:                197.4400                   The Weighted Bounded Slow Down metric
Market counter:      0
Market avg time:     nan (milli sec)            Average time it took to run the market
Market max time:     0.0000 (milli sec)         Maximal time it took to run the market
```

2.5.3.2. Performance vs Value Output Section

Another important section of the simulation output is the performance versus value section. This section provides an in depth view of the performance gained by each group of processes with the same value. This section is basically a table showing performance values obtained by the group of processes with the same value.

The example below shows the performance-vs-value section obtained for the ‘jobs.1’ and ‘providers.1’ input files.

```
#+-+-+- section-start: pv
#This file was automatically generated by the economy simulator
#Slowdown vs Value data
#MIN-SD      Value   SD       BSD      Jobs   BSD-Stddev
0.00         20      1.9400   1.9400   1        0.0000
0.00         21      1.8400   1.8400   1        0.0000
0.00         22      1.0000   1.0000   1        0.0000
0.00         23      1.0000   1.0000   1        0.0000
0.00         24      1.0000   1.0000   1        0.0000
0.00         25      1.0000   1.0000   1        0.0000
0.00         26      1.0000   1.0000   1        0.0000
```

In the above example each non comment line (comment lines starts with #) indicates the performance gained by the processes with the values of 20 through 26. The first column specifies the minimal slowdown which those jobs had. This minimal slowdown is ‘0’ in the example. The performance is specified by the following three metrics:

- Slowdown – which is *runtime/total-time*
- Bounded slowdown which is: *max(runtime,alpha)/max(runtime,alpha)*
- Bounded slowdown standard deviation.

3. System Administrator Manual

This chapter contains the "the system-administrator installation and configuration manual" contained in the MOSIX-SORMA package.

3.1. Introduction

This guide is intended for system administrators wishing to install the MOSIX-SORMA package on top of a MOSIX cluster (for more details on MOSIX visit www.mosix.org). The MOSIX-SORMA package enables economically enhanced scheduling in a MOSIX Grid instead of the default load balancing based scheduling. Using this package, resource owners such as a university department owning a student farm can earn money by allowing remote jobs to run on the student farm cluster when parts of it are not in use.

The main benefit of using MOSIX is the ability to perform transparent preemptive migration of processes. This means that a running process can be migrated (moved) from one node to another without the process being aware of this migration. Thus, in the above "student-farm" scenario, when one of the nodes becomes unavailable to the market (having been reclaimed by a student) the processes running on that node are transparently moved out of the node without having to be killed.

3.1.1. The MOSIX-SORMA package structure

The **MOSIX-SORMA** package is composed of 3 independent components:

- The Market component: responsible for trading resources between resource owners and clients.
- The Provider component: responsible for representing the resource owner.
- The Client component: allowing the user to submit jobs with economical properties

This guide **DOES NOT COVER** the internal structure of the above components. It is only an installation and configuration guide.

3.1.2. Structure of Installation Guide

This installation guide is a step by step guide for installing the MOSIX-SORMA components on an already existing MOSIX installation.

In section 3.2 we cover the installation of the market component. Section 3.3 covers the configuration of the market manager component. In Section 3.4 the configuration of the provider component is covered. Section 3.5 covers the configuration of a client node. In Section 3.6 the installation and configuration of the MOSIX-SORMA web interface is presented. Finally in Section 3.7 we provide an example of how to test that the MOSIX-SORMA installation is working properly.

3.1.3. Installation Example Information

In this installation guide we will use the following simple scenario as a step by step example. We assume the existence of a 3 nodes MOSIX cluster, where the nodes will be referred to as: *sorma-market*, *sorma-provider* and *sorma-client*. We'll keep the example as simple as possible so it will be easy to recreate the example on almost any site. We also assume that the IP addresses of the nodes are as follows:

Node Name	Nodes IP	Description
sorma-market	192.168.1.100	Runs the market component
sorma-provider	192.168.1.101	A provider node
sorma-client	192.168.1.102	A client node

3.1.4. Prerequisite

- A working installation of MOSIX. In order to install MOSIX you need to go to www.mosix.org where you can obtain all the required packages and manuals of MOSIX.
    - Another possibility is to download a pre-configured VMware virtual machine disk with MOSIX already installed. This way you can setup a virtual MOSIX cluster without any need to install MOSIX.
- Perl version 5.8.8 or higher.
- A C language compiler (gcc) for compiling the client component.

3.1.5. Downloading the MOSIX-SORMA package

The MOSIX-SORMA package can be downloaded from http://www.mosix.org/sorma/mosix-sorma-current.tgz

3.2. *Installing the MOSIX-SORMA package*

The instructions below assume that a tar ball named mosix-sorma-1.0.tgz was downloaded (see instructions in the introduction section). Note that this manual uses version 1.0 of the MOSIX-SORMA package though in the general case the version number might be different. We also assume that the installation is done as user *root*. The following steps should be performed on all the nodes:

- Open the MOSIX-SORMA tar ball by running

```
>> tar xvfz mosix-sorma-1.0.tgz
```

- `Enter the newly created directory mosix-sorma-1.0`

```
  cd mosix-sorma-1.0
```

- Run the *mosix-sorma-install* program

```
   ./mosix-sorma-install
```

- You will be requested to specify a root directory for the installation. This is in case you would like to install on an alternative root (e.g. when managing a diskless installation)

```
    Where is the root directory '/' located (default /): [/] :
```

- You will be asked to specify the installation directory (default directory is */opt/mosix-sorma/*):

```
  Where would you like to install MOSIX-SORMA (default is under /): [/opt/mosix-
sorma] :
```

- You will be prompted for acknowledging the creation of new directories:

```
  Directory /opt/mosix-sorma/lib does not exits
  Would you like to create it? [Y/n]? [Y] :
  Installing lib/ecologger libdir in /opt/mosix-sorma/lib/lib/ecologger
  Installing lib/economyd libdir in /opt/mosix-sorma/lib/lib/economyd
  Installing lib/providerd libdir in /opt/mosix-sorma/lib/lib/providerd
  Installing lib/Util libdir in /opt/mosix-sorma/lib/lib/Util
  Installing lib/tld libdir in /opt/mosix-sorma/lib/lib/tld
  ...
  ...
```

- Eventually the installation process will finish:

```
 *************************************************************
 * Installation of MOSIX-SORMA completed successfully
 *
 * You can run /opt/mosix-sorma/bin/mosix-sorma-conf to configure this node
 * as a market, provider or client.
 *
 ************************************************************
```

- At the end of the installation process, all the MOSIX-SORMA binaries will be placed (by default) under */opt/mosix-sorma*. This location will be assumed in the next parts of this guide.

- In addition, the installation process installs the file */etc/init.d/mei* which is an init script responsible for starting and stopping the MOSIX-SORMA components when the node boots-up or shuts-down. Make sure that the appropriate symbolic links were created from */etc/rc{3,4,5}.d* (or the equivalent directories depending on your Linux distribution) to */etc/init.d/mei*. If not, you should install the *mei* service manually by using you Linux distribution service enablement tools, such as *chkconfig*.

3.3. Configuring a Market Manager

The market manager component is responsible for running the market mechanism for allocating (and reallocating) jobs to providers. In this section we will show how to configure this component. It is assumed that the installation of the MOSIX-SORMA package finished successfully.

Before you configure the market manager node, make sure that you choose a node which has enough computing power and memory to run the market manager component. Although the memory and CPU requirements are relatively low, it is highly unrecommended to configure the market-manager component on a node which is also a provider node. The market-manager might disturb the execution of the hosted jobs and this will result in a lower performance of the provider then expected.

Another issue to consider is stability. It is recommended to choose a stable node which is not expected to reboot frequently. Otherwise the market environment will experience no-service periods which will cause the market to be unstable.

3.3.1. Automatic Configuration

The recommended configuration method of the market-manager component is to use the *mosix-sorma-conf* program. This program is located under the *bin* directory of the MOSIX-SORMA installation (by default, the location will be */opt/mosix-sorma/bin/mosix-sorma-conf*).

To configure the local node as a market-manager simply run (as root):

```
  ./mosix-sorma-conf market
```

Thats all. If everything goes well, the *economyd* daemon will start running. You can verify that his daemon is running by typing:

```
  ps auxww |grep economyd
  root      7803  0.3  3.6  16952 12048 ?        Ss   10:37   0:00 /usr/bin/perl -
w /opt/mosix-sorma/bin/economyd
```

In the above example you can see that the *economyd* daemon started to run (pid=7803)

3.3.2. Manual Configuration

This section covers the manual installation procedure of the *economyd* daemon market-manager daemon. Note, that the above automatic method is much simpler and you should use the manual installation only when your settings are special.

The configuration of the *economyd* daemon is divided into 2 parts. The first is creating a configuration file for the *economyd* daemon and the second is making sure the *economyd* daemon will start/stop when the */etc/init.d/mei* script is used.

The configuration file of the market daemon resides in */etc/mosix/ecod.conf*. This is an XML formatted file which controls the behaviour of the market daemon. The structure of this file is as follows:

- <status-file> tag contains the name of an xml status file the *economyd* daemon is creating every few seconds
- <log-file> tag contains the name of a log file the *economyd* daemon is using to log its activity
- <market-type> tag specifies the type of the market. In our case only the **central** value should be used
- <market-solver> tag specifies the name of the market algorithm to use.
- <solver-prog> tag specifies the location of the market solver program.
- <providers> tag contains the definition of the providers that can participate in this market.

- <cluster> tag contains the definition of a cluster of providers. Each such cluster should contain at least one representative definition.
- <rep> tag contains the host name (or IP address) of a provider representative.

Configuration File Example

Bellow you can see an example of a configuration file for the *economyd* daemon's *market-manager* daemon (as created by the automatic configuration procedure):

```
<conf>
  <status-file>/tmp/.stat</status-file>
  <log-file>/tmp/ed.log</log-file>
  <market-type>central</market-type>
  <market-solver>mig</market-solver>
  <solver-prog>/opt/mosix-sorma/bin/solver</solver-prog>
  <providers>
    <cluster name="sorma-demo">
      <rep>localhost</rep>
    </cluster>
  </providers>
</conf>
```

In the example you can see that the market-manager is using the local host as the cluster representative. In this case it is assumed that the market-manager node is part of a running MOSIX cluster so it can access information about other nodes in the cluster via the *infod* system.

Automatic start/stop of *economyd*

The second part of the manual configuration is to make sure the *economyd* is automatically started/stopped when the node starts or stops. To do so you need to edit the file */etc/default/mei* file. Below, is an example of such a file:

```
# This file was generated by sorma-conf
ECOD_ENABLE=yes
PVD_ENABLE=no
AD_ENABLE=no
```

In order for the *economyd* daemon to automatically start on this node, the value of the variable *ECOD_ENABLE* should be set to **yes**.

3.4. Configuring a Provider Node

In this section we show how to configure a node as a provider by running the *providerd* daemon. A provider node participates in the MOSIX-SORMA market by publishing itself as an active provider via the MOSIX *infod* information system. Once the market-manager detects an active provider it will send jobs to that node (keeping track of the currency this provider earns).

Again, as with the market-manager component, the installation can be done automatically or manually.

3.4.1. Automatic Configuration

To configure the provider node automatically just run (from */opt/mosix-sorma/bin*):

```
./mosix-sorma-conf provider
```

The output will be:

```
Configuring a provider node
Loading configuration from /etc/mosix/mei.conf
Stopping providerd ...
```

You will be prompted to enter a provider nickname:

```
Enter provider nickname: provider1
Nick name (provider1) ok
Loading configuration from /etc/mosix/mei.conf
Starting providerd ...
Found /etc/mosix/mei.conf
Found MEI_LIB_DIR= /opt/mosix-sorma/lib at /etc/mosix/mei.conf
```

As can be seen in the above example. The provider is eventually started on the local node. To verify that the provider is running:

```
>> ps auxww |grep providerd
root      7844  0.0  2.9  14872  9548 ?        Ss   11:14   0:00 /usr/bin/perl -w
/opt/mosix-sorma/bin/providerd
```

3.4.2. Manual Configuration

The configuration of the *providerd* daemon is composed of 2 main parts: generating a configuration file, and insuring that the provider will start/stop whenever the */etc/init.d/mei* service is started or stopped.

The configuration file of the *providerd* daemon is located at */etc/mosix/provider.conf*. This is an XML file with the following structure:

- <status> tag specifies the starting status of the provider (either: on or off)
- <market> tag specifies the market type this provider is participating in. In our case only the *central* value is used/
- <min-price> tag specifies that minimal reservation price of the provider.
- <time-frame> tag specifies the time frames in which the provider is active. This tag allows provider node to join the market environment only in predefined times.
- <nickname> tag specifies an optional nickname for the provider.

Configuration File Example

Below is an example of the *provider.conf* configuration file (as created by the automatic installation procedure):

```
<provider-conf>
      <status>on</status>
      <market>central</market>
```

```
       <min-price>30</min-price>
       <time-frame>
               <always>1</always>
       </time-frame>
       <nickname>provider1</nickname>
 </provider-conf>
```

In the example, the provider has a minimal price of 30 currency units per CPU hour, It is always participating in the market and it's *nickname* is *provider1*.

To make sure the *providerd* is correctly managed by the */etc/init.d/mei* service, you should edit the file */etc/default/mei* and make sure the value variable *PVD_ENABLE* is equal to 'yes'.

3.5. Configuring a Client Node

The client component of the MOSIX-SORMA package allows users to submit jobs to the market. The configuration of this component can be done automatically or manually. The client component is represented by a daemon called *assigned*.

3.5.1. Automatic Configuration

To configure the client node automatically do:

```
 mosixdemo:/opt/mosix-sorma/bin# ./mosix-sorma-conf client
```

The output would be something like:

```
 Configuring a client node
 Loading configuration from /etc/mosix/mei.conf
 Stopping Assignd ...
 Please choose a way to determine the market this client node
 will connect to:
 1)  Choose market manager specifically
 2)  Active - this client will try to detect the market
 Enter your choice: 1
 Enter IP address or name of market node:localhost
 Using localhost as the market manager
 Loading configuration from /etc/mosix/mei.conf
 Starting Assignd ...
```

Note, that in the above example the program requested the host-name (or IP address) of the market-manager node. Two methods are possible for specifying the address of the market-manager. The first is "Choose market manager specifically" which is the method that was chosen in the example. The second method is using the automatic detection mechanism of the MOSIX-SORMA package.

Once this stage is done, you should make sure that the *assignd* daemon is running:

```
 >> ps auxww |grep assignd
 >>root      7871  0.0  0.5   1940  1940 ?        SL   11:30   0:00 /opt/mosix-
sorma/bin/assignd
```

3.5.2. Manual Configuration

The manual configuration of the *assignd* daemon is composed of two steps: generating a configuration file, and making sure the */etc/init.d/mei* service will start/stop the *assignd* whenever it is stopped/started.

The configuration file of the *assignd* is located at */etc/mosix/assign.conf*. This is a text file with the following format:

- A line with the format: C {hostname|IP_ADDRESS} # to specify the address of the market-manager node
- A line with the format: P {port_number} # to specify the port number to use (this line is optional)

Once this file is created you should make sure the *assignd* is started/stopped by the */etc/init.d/mei* service. This is done by editing the file */etc/default/mei* so that the value of the variable *AD_ENABLE* is equal to ‘yes’.

3.6. Web Interface

3.7. Testing the Installation

This section contains information about how to test that an installation of the MOSIX-SORMA package is functioning correctly.

3.7.1. Testing MOSIX

First we test that our MOSIX installation is working properly. From the client node we should be able to see the provider node in our MOSIX configuration printout.

```
 >>setpe -R
```

This command should print something similar to...

```
FILL IN THE OUTPUT OF SETPE –R on a demo cluster
```

3.7.2. Testing the *Infod* Information Service

To test the *Infod* information service do:

```
 sorma-market> infod-client
 sorma-provider> infod-client
 sorma-client> infod-client
```

You can also use the utility *mmon* to display an online monitor of the cluster (you can run this from any one of the nodes). The result should be that all 3 nodes are connected together and see each other.

3.7.3. Testing the Market

To test that the market daemon is working properly use the *ecodctl* command as follows:

```
mosixdemo:/opt/mosix-sorma/bin# ./ecodctl status
Hostname:          mosixdemo
Assignd Port:      9004
Ctl Port:          9006
Assingd's          1
Market:            Central
Solver:            Greedy-Migration
Jobs:              Total:0
                   Wait: 0 Run:0 Suspend:0                  Finish: 0
Providers:         1 (in 1 clusters)
Time to next run:  1
Consumers Surplus: 0.000
Providers Surplus: 0.000
Social Welfare:    0.000
```

This should print information about the current status of the market daemon. In the example above, it can be seen that currently there is 1 *assignd* daemon connected to the market, and that 1 provider was detected.

3.7.4. Testing the Provider

To test that a provider node is working properly use the *pvdctl* program to query the status of the provider by doing:

```
mosixdemo:/opt/mosix-sorma/bin# ./pvdctl status
machine: localhost:
-----------------------
Status:        on
Market status: free
Min Price:     30
Curr Price:    0
Next Price:    30
```

In the example above, the *ecodctl* program managed to connect to the *providerd* daemon and obtain status information.

3.7.5. Testing the Client

Finally, we can test a submission of a job to the market. To do so run:

```
>> ./srun -p 100 testload -t 120 &
```

To view the status of the job run the following command on the market node:

```
>> ./ecodctl lj
ID        User      Status      V      T runtime    M runtime    Where     Comment
65620     root      run         100.0  12           12           mosixdemo Running
```

As can be seen in the example above, the job started to run on the provider *mosixdemo*.

Now, we will submit another job with higher value by:

```
>> ./srun -p 200 testload -t 120 &
```

The expected result is that the first job will be suspended while the second process will start to run. To verify this:

```
>> ./ecodctl list-jobs
ID        User      Status      V      T runtime   M runtime   Where     Comment
65620     root      suspend     100.0  120         120                   Suspended
65621     root      run         200.0  2           2           mosixdemo Running
```

As can be seen from the output of *ecodctl list-jobs' the first job was suspended and the second jobs started to run.*

4. Users Manual

NOTE: This chapter contains the MOSIX-SORMA user manual as provided in the MOSIX-SORMA package

4.1. Introduction

This guide is intended for users (both resource buyers and sellers) who would like to use the MOSIX-SORMA package. The information presented here is also available in the manuals provided with the MOSIX-SORMA package. Each of the following programs: *assignd, srun, economyd, ecodctl, providerd, pvdctl* has its manual (which is usually installed under the /opt/mosix-sorma/man directory).

4.1.1. Prerequisite

This guide assumes that the MOSIX-SORMA package was already installed and configured by the administrator of the system.

4.2. Resource Provider

This section describes the operations that can be performed by the owner of a provider node.

4.2.1. Joining the market

To join the market, use the command:

```
   ./pvdctl market-on
```

This command will cause the *providerd* daemon to join the market environment and publish itself as available.

4.2.2. Leaving the market

To temporarily leave the market environment do:

```
./pvdctl market-off
```

This will cause the *provider* to leave the market and publish itself as not available.

4.2.3. Setting the price

To modify the minimal price the provider node is asking for running a job do:

```
 ./pvdctl setprice 50
```

This will set the reservation price of the provider to 50 currency units per CPU hour.

4.2.4. Setting time tables for market participation

In case the provider is not supposed to always participate in the market, it is possible to define time frames in which the provider may join the market (leaving the market when out of the time frame). To define a time frame, you will need to edit the file */etc/mosix/provider.conf* and add a section like:

```
<time-frame>
    <frame>
        <start>8:00</start>
        <end> 12:30</end>
    </frame>
</time-frame>
```

In the above example the provider will participates in the market only between 8:00 and 12:30.

4.3. Buyers of resources

This section presents how users should submit jobs via the MOSIX-SORMA package.

4.3.1. Submitting jobs

The command *srun* is used as the job submission tool of the MOSIX-SORMA package. It is a wrapper program for the MOSIX *mosrun* utility.

To submit a job to the market with a maximal price of 55 currency units per CPU hours do:

```
./srun -p 55 PROGRAM PROGRAM-ARGS
```

In the above example, **PROGRAM** is the program the user wishes to run and **PROGRAM-ARGS** are its arguments.

4.3.2. Monitoring submitted jobs

To view the status of submitted jobs do:

```
>>ecodctl list-jobs
ID        User      Status      V      T runtime   M runtime   Where     Comment
65623     root      run         55.0   0           0           mosixdemo Finished
waiting; now running
```

The output of *list-jobs* is the current status of registered jobs. This shows the user, which submitted the job, the current status, the value of the job, the node on which it’s currently running etc.

5. MOSIX-SORMA Market Simulator User Manual

IMPORTANT NOTE:

This chapter will be provided at the end of the project since the simulator framework is a research tool which is still under development.

We note that the simulator is not an integral part of the MOSIX-SORMA system as specified by the description of work. But since we found it to be such a valuable tool we decided to include it in the final deliverable of the system.

6. The MOSX-SORMA software deliverable

This chapter describes the contents of the MOSIX-SORMA software deliverable. As specified before, this software part of D5.4 contains the following:

- A VMware virtual machine disk image of a pre-installed virtual machine with MOSIX and the MOSIX-SORMA components.
- A tar-ball of the MOSIX-SORMA package, containing all the MOSIX-SORMA components in a ready to install manner.
- A CVS snapshot of the MOSIX-SORMA source tree containing all the MOSIX-SORMA components in a ready to compile state.

6.1. Description of the MOSIX-SORMA VMware disk image

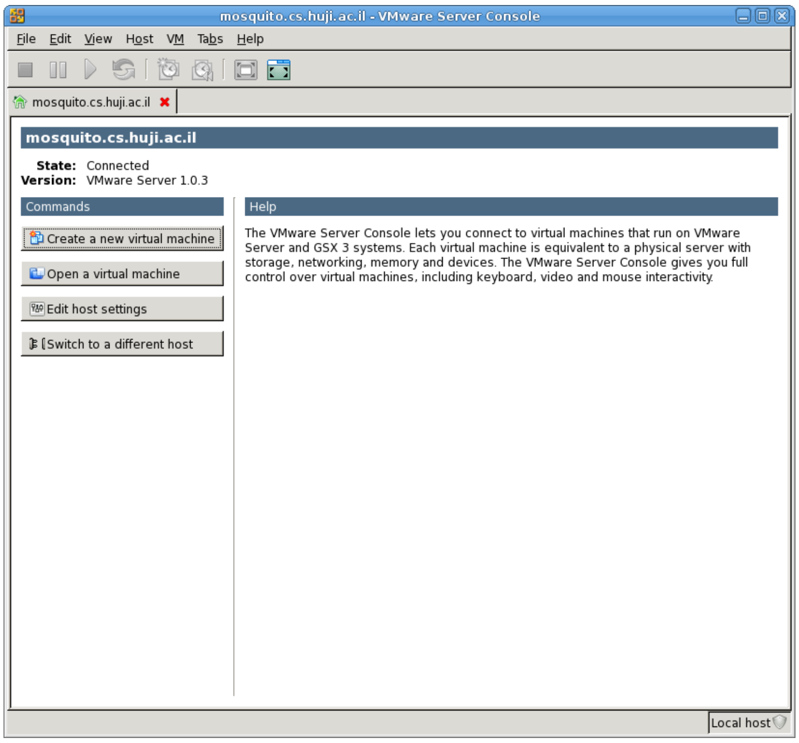
The MOSIX-SORMA VMware disk image is a complete installation of a Linux Debian machine inside a VMware server virtual machine. This virtual machine includes a complete installation of the MOSIX system as well as the MOSIX-SORMA system. This makes it is possible to boot up the VM and instantly play with the MOSIX-SORMA package.

We provide a short guide on how to create and run a virtual machine using the MOSIX-SORM Vmware disk image.

6.1.1. VM Creation Quick-Guide

For this guide we are using "VMware Server 1.0.3", but the procedure is quite the same with any other VMware product.

First we run "VMware Server Console", which should look like this:

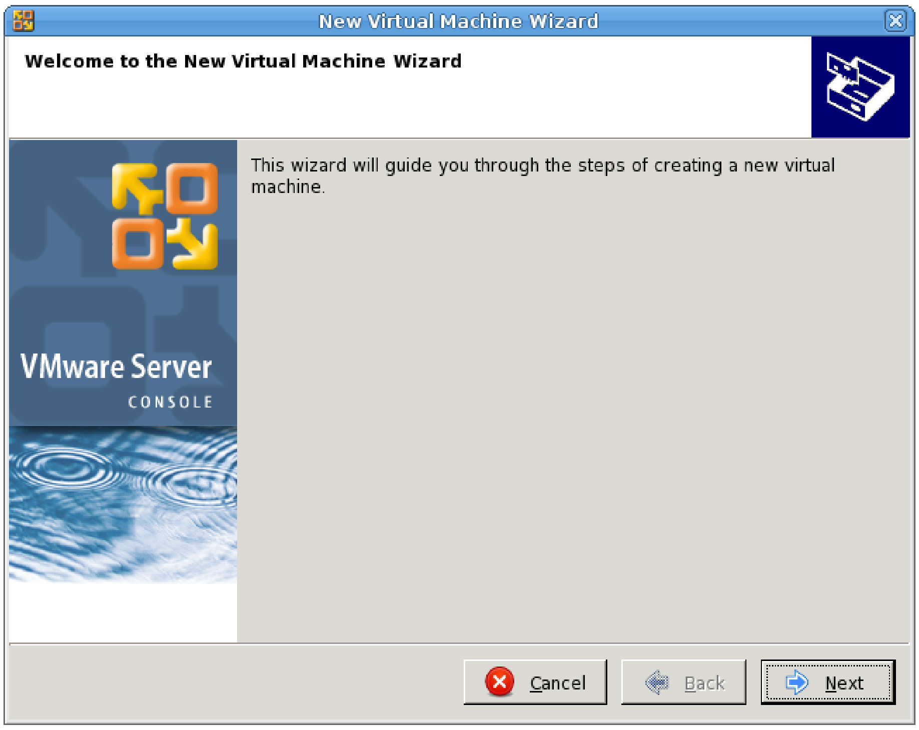

mosquito.cs.huji.ac.il - VMware Server Console
File Edit View Host VM Tabs Help
mosquito.cs.huji.ac.il
Commands
Create a new virtual machine
Open a virtual machine
Edit host settings
Switch to a different host
Help
Local host

mosquito.cs.huji.ac.il

State: Connected
Version: VMware Server 1.0.3

The VMware Server Console lets you connect to virtual machines that run on VMware Server and GSX 3 systems. Each virtual machine is equivalent to a physical server with storage, networking, memory and devices. The VMware Server Console gives you full control over virtual machines, including keyboard, video and mouse interactivity.

Click on "Create a new virtual machine" and you will see the following dialog:

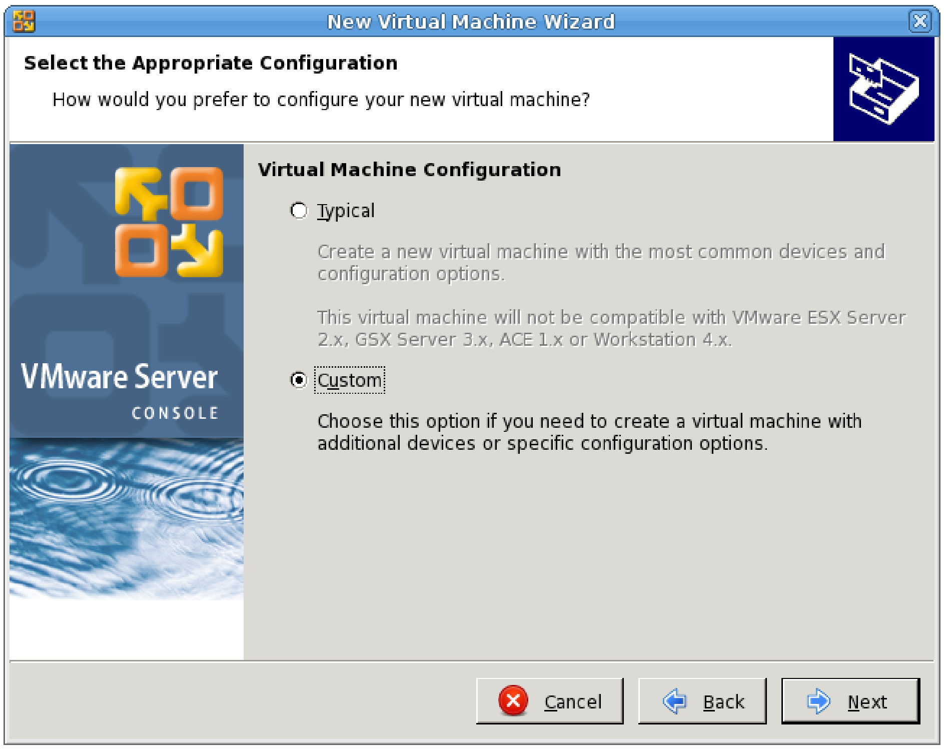


New Virtual Machine Wizard

Welcome to the New Virtual Machine Wizard

This wizard will guide you through the steps of creating a new virtual machine.

Click “Next” and choose “Custom”:



New Virtual Machine Wizard

Select the Appropriate Configuration

How would you prefer to configure your new virtual machine?

Virtual Machine Configuration

○ Typical

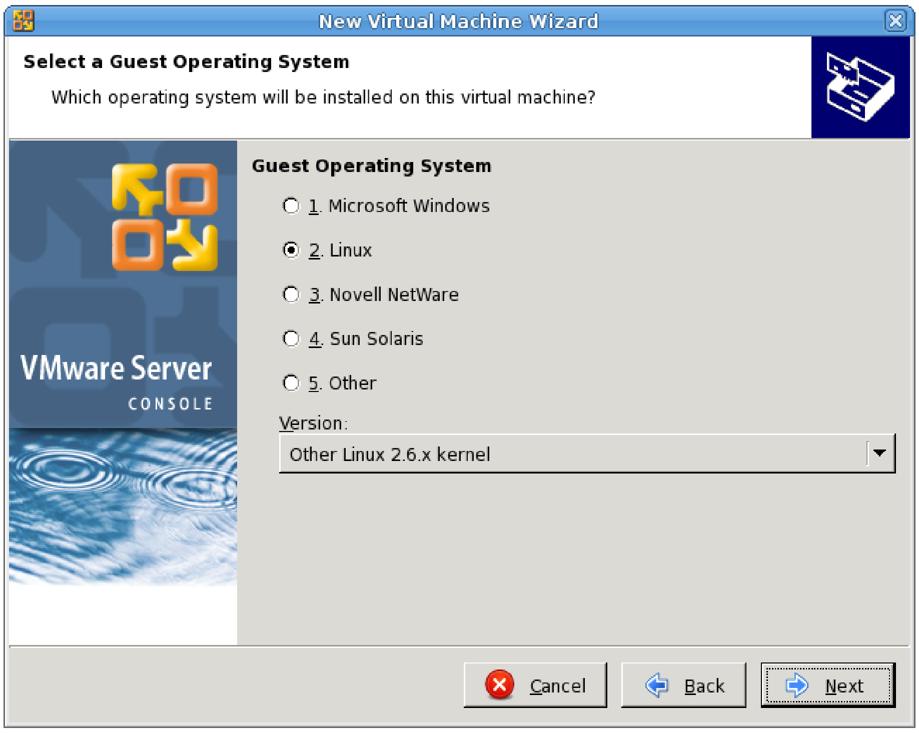
Create a new virtual machine with the most common devices and configuration options.

This virtual machine will not be compatible with VMware ESX Server 2.x, GSX Server 3.x, ACE 1.x or Workstation 4.x.

● Custom

Choose this option if you need to create a virtual machine with additional devices or specific configuration options.

Now choose “Linux” and “Other Linux 2.6.x kernel” from the drop-box and click “Next”:



New Virtual Machine Wizard

Select a Guest Operating System

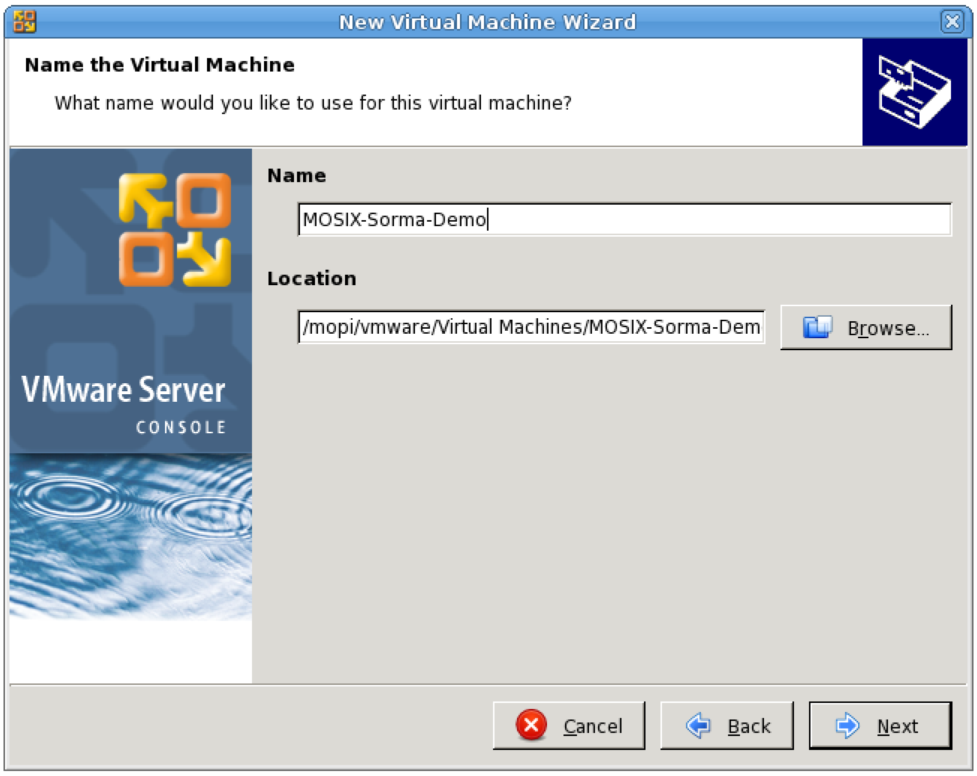
Which operating system will be installed on this virtual machine?

Guest Operating System

- 1. Microsoft Windows
- [✓ 2. Linux]
- 3. Novell NetWare
- 4. Sun Solaris
- 5. Other

Version: Other Linux 2.6.x kernel

Now enter a name for the new VM:



New Virtual Machine Wizard

Name the Virtual Machine

What name would you like to use for this virtual machine?

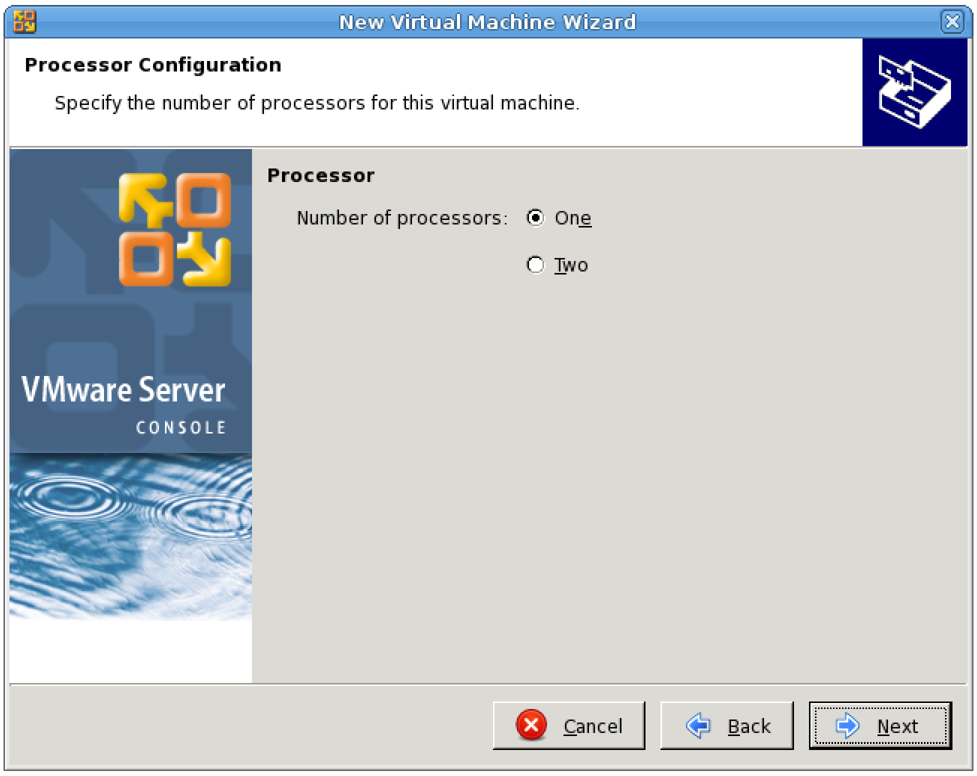
Name

MOSIX-Sorma-Demo

Location

/mopi/vmware/Virtual Machines/MOSIX-Sorma-Dem

Next, choose the number of desired virtual cpus:


New Virtual Machine Wizard

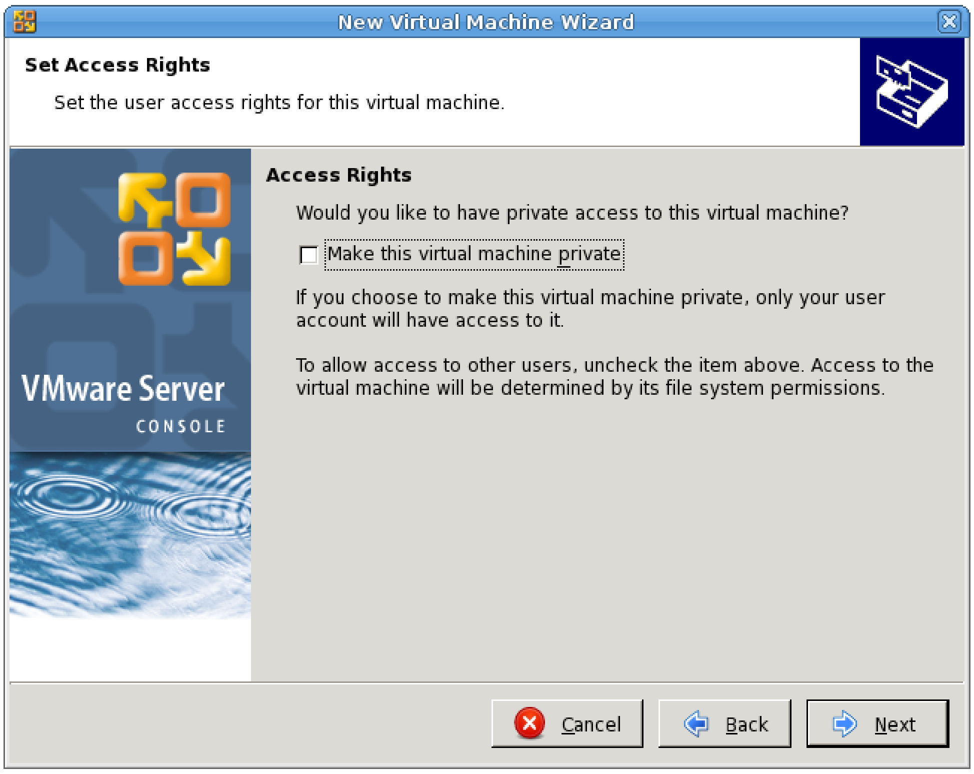
Processor Configuration

Specify the number of processors for this virtual machine.

Processor

Number of processors: [✓ One] Two

It is reasonable to assume that other people will also want to try out the demo, so consider unchecking this checkbox:


New Virtual Machine Wizard

Set Access Rights

Set the user access rights for this virtual machine.

Access Rights

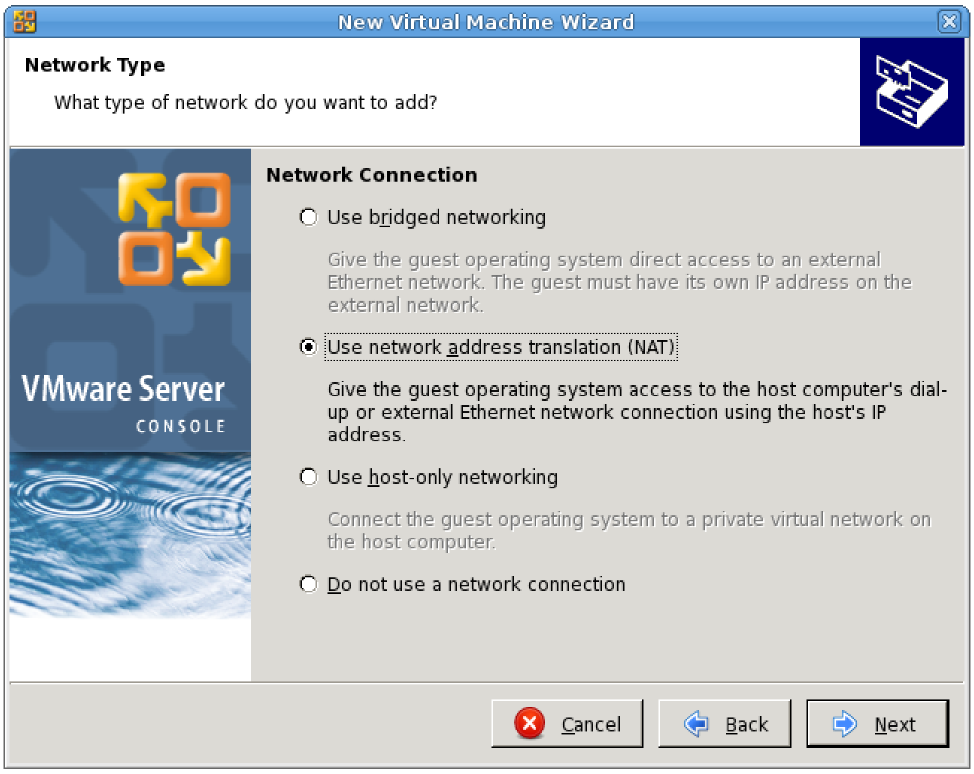
Would you like to have private access to this virtual machine?

☐ Make this virtual machine private

If you choose to make this virtual machine private, only your user account will have access to it.

To allow access to other users, uncheck the item above. Access to the virtual machine will be determined by its file system permissions.

Choosing “NAT” here is the simplest way to configure the network. However, this will require that all the VMs in the demo cluster will run on the same host. If you want to run VMs on several hosts (or physical machines), you should use bridged networking, otherwise MOSIX will not be able to recognize all the machines in the cluster:



New Virtual Machine Wizard

Network Type

What type of network do you want to add?

Network Connection

○ Use bridged networking

Give the guest operating system direct access to an external Ethernet network. The guest must have its own IP address on the external network.

● Use network address translation (NAT)

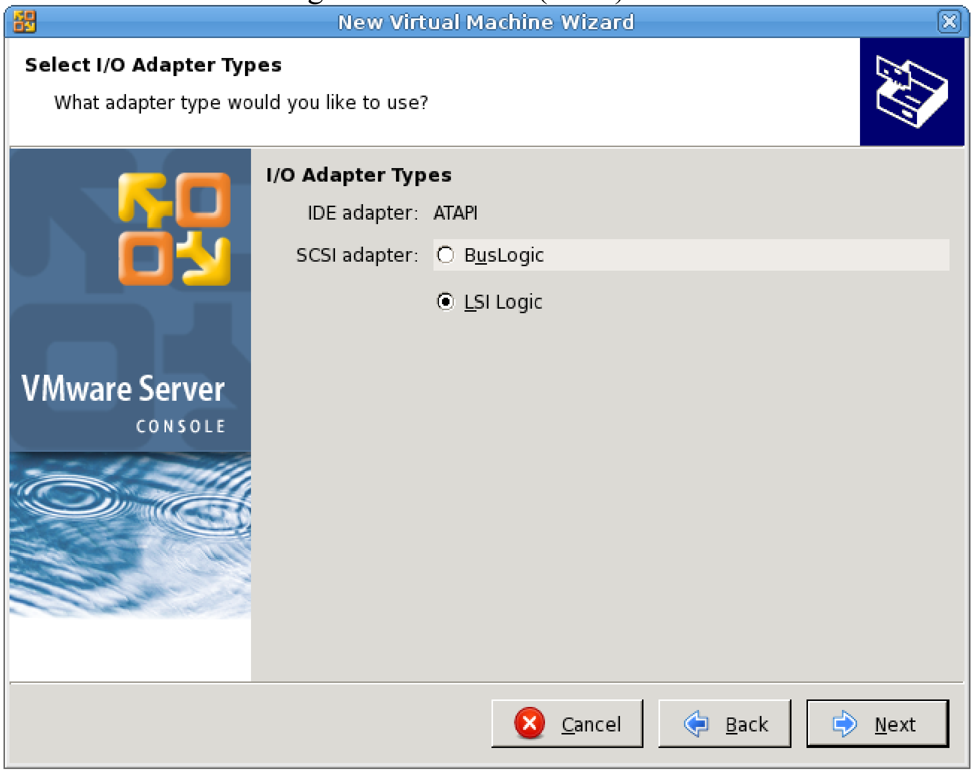
Give the guest operating system access to the host computer's dial-up or external Ethernet network connection using the host's IP address.

○ Use host-only networking

Connect the guest operating system to a private virtual network on the host computer.

○ Do not use a network connection

Out virtual disk is configured as a SATA (SCSI) drive connected to an “LSI Logic” SCSI bus:



New Virtual Machine Wizard

Select I/O Adapter Types

What adapter type would you like to use?

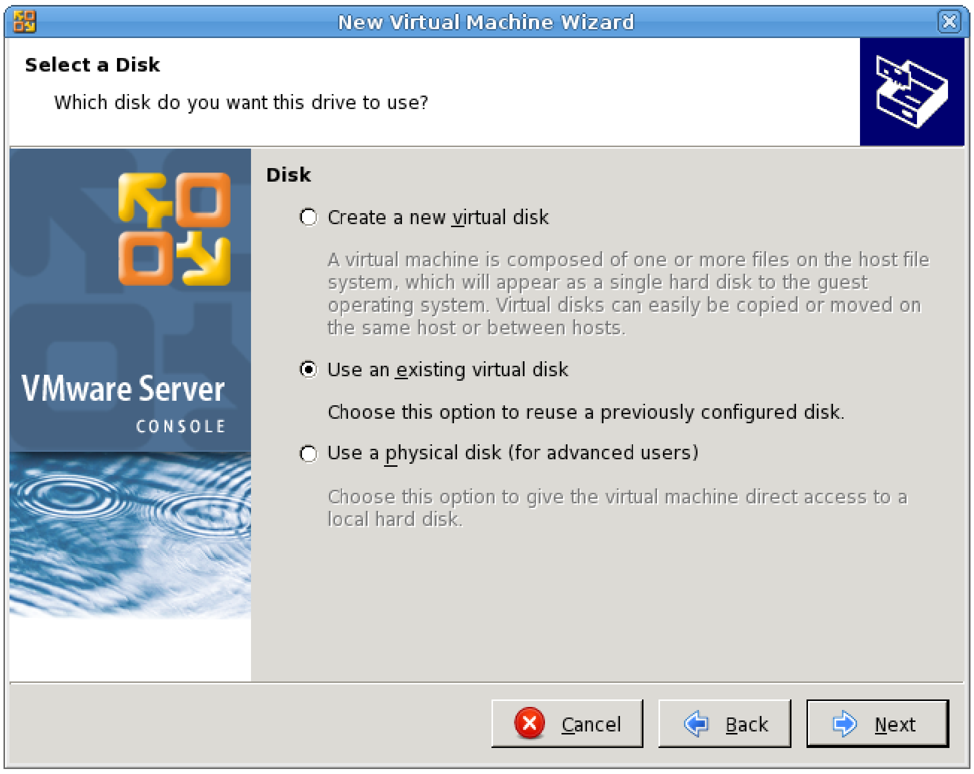
I/O Adapter Types

IDE adapter: ATAPI

SCSI adapter: ○ BusLogic

● LSI Logic

Choose “Use an existing virtual disk”:


New Virtual Machine Wizard

Select a Disk

Which disk do you want this drive to use?

Disk

- ( ) Create a new virtual disk

  A virtual machine is composed of one or more files on the host file system, which will appear as a single hard disk to the guest operating system. Virtual disks can easily be copied or moved on the same host or between hosts.

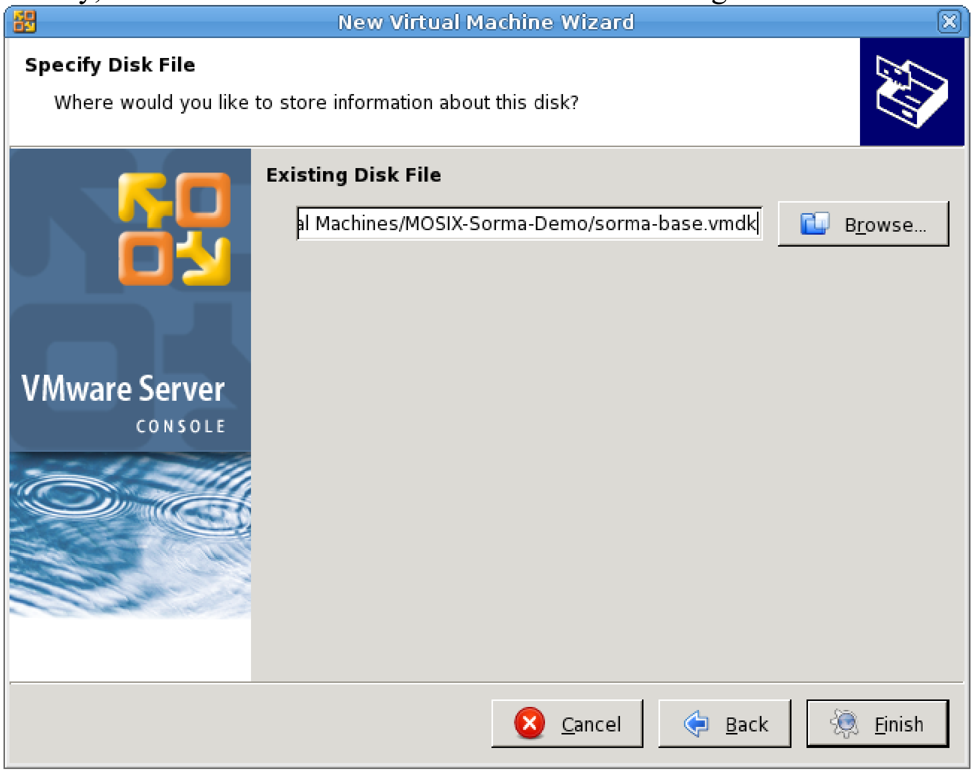
- (●) Use an existing virtual disk

  Choose this option to reuse a previously configured disk.

- ( ) Use a physical disk (for advanced users)

  Choose this option to give the virtual machine direct access to a local hard disk.

Finally, choose the 'sorma-base.vmdk' virtual disk image file and click "Finish":


New Virtual Machine Wizard

Specify Disk File

Where would you like to store information about this disk?

Existing Disk File

al Machines/MOSIX-Sorma-Demo/sorma-base.vmdk

You should now have a fully configured VM ready to be booted with the MOSIX-SORMA demo image.

6.2. Description of the MOSIX-SORMA tar ball

The MOSIX-SORMA tar-ball contains all the necessary components of the MOSIX-SORMA system in a ready to install status. The contents of this tar file are:

- An installer program
- A *bin* directory with all the necessary binaries
- A *lib* directory with all the necessary libraries
- A *doc* directory with all the necessary documentation

6.3. Description of the MOSX-SORMA source tree

The MOSIX-SORMA source tree provide in the MOSIX-SORMA CD is a snapshot of the development tree of the MOSIX-SORMA system. It contains the main following directories

- assignd – all the *assignd* daemon and *srun* utility sources
- economyd – the *market-manager* component
- providerd – the provider component
- ecologger – a facility for obtaining live information from a running market
- Util – Perl language utility function for the Perl based components
- libutil – C language utility functions for the solver and *sim* modules
- sim – the simulator
- solver – the market solver framework
- tld – the testing framework for generating various load scenarios on a real MOSIX-SORMA system
- Web – the web interface of the MOSIX-SORMA system

SORMA Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number IST-FP6-034286.

Barcelona Supercomputing Center

BF/M Bayreuth, Universität Bayreuth

Cardiff University

Correlation Systems Ltd.

FZI Forschungszentrum Informatik

Hebrew University

Institut für Informationswirtschaft und -management (IISM), Universität Karlsruhe (TH)

Sun Microsystems

Swedish Institute of Computer Science

TXT e-Solutions

Universitat Politècnica de Catalunya

University of Reading