Automatic load balancing and transparent process migration

Roberto Innocente
rinnocente@hotmail.com
November 24,2000

Download postscript from : mosix.ps
or gzipped postscript from: mosix.ps.gz

Introduction

The purpose of this presentation is an introduction to the most important features of MOSIX:
- automatic load balancing
- transparent process migration

giving an overview of related projects and of the different possible implementations of these mechanisms.

Scheme of the presentation

1. Overview of some Distributed Operating Systems
2. Load balancing mechanisms
3. Process migration mechanisms
4. focus on Mosix

Overview of some distributed OS

Distributed O.S.

- Sprite
- Charlotte
- Accent
- Amoeba
- V System
- MOSIX
- Condor (this is just a batch system, but offering job migration)

Sprite (Douglis,Ousterhout* 1987)

Experimental Network OS developed at UCB.
Was running on Suns and Decstations.
The process migration in this system introduced the simplifying concept of Home Node of a process (the originating node where it should still appear to run).
After then this concept has been utilized by many other implementations.

*of Tcl/Tk fame

Charlotte (Artsy, Finkel 1989)

It's a Distributed Operating System developed at the UoWM based on the message passing paradigm.
Provides a well-developed process migration mechanism that leaves no **residual dependencies** (the migrated process can survive to the death of the originating node), and therefore is suitable also for fault tolerant systems.

Accent (Zayas 1987)

- developed at CMU as precursor of Mach
- a kernel not Unix compatible with IPCs based on a "port" abstraction
- process migration has been added by Zayas(1987)
- no load balancing

Amoeba (Tanenbaum et al 1990)

- Tanenbaum,van Renesse,van Staveren, Sharp, Mullender, Jansen, van Rossum**(Experiences with the Amoeba Distributed Operating System 1990)*
- Zhu,Steketee*(An experimental study of Load Balancing on Amoeba, 1994)*
- doesn't support virtual memory
- was based on the *processor-pool model*, that assumes the # of processors is more than the # of processes

* of python fame

V system (Theimer 1987)

- developed at Stanford, microkernel based, runs on a set of diskless wks
- V has support for both remote execution and process migration, the facilities were added having in mind to utilize wks during idle periods

MOSIX (Barak 1990,1995)

its roots are back in the mid ’80s :

- MOS multicomputer OS (Barak, HUJI 1985)
- set of enhancements to BSD/OS: MOSIX ’90
- port to LINUX (Mosix 7th edition) around ’95: the system call based mgmt has been transformed to a /proc filesystem interface
- enhancements ’98 (memory ushering algorithm,...)

Condor (Litzkow,Livny 1988)

Condor is not a Distributed OS.
It is a batch system whose purpose is the utilization of the idle time of WKS and PCs.
It can migrate jobs from machine to machine, but imposes restrictions on the jobs that can be migrated and the programs have to be linked with a special checkpoint library.
(More details on http://hpc.sissa.it/batch)

Load balancing mechanisms

Load balancing methods taxonomy

- job initiation only (aka job placement, aka remote execution)
  - static
  - dynamic (almost all batch systems can do this: NQS and derivatives)
- migration (Mosix, some Distributed OS)
- job initiation + migration

Load balancing algorithms

Two main categories:

- Non pre-emptive: *they are used only when a new job has to be started, they decide where to run a new job (aka job placement, many batch systems)*
- Pre-emptive: *jobs can be migrated during execution, therefore load balancing can be dynamically applied during job execution (Distributed OS,Condor)*

Non pre-emptive load balancing (aka job placement)

- **centralized initiation:** there is only 1 load balancer in the system that is in charge of assigning new jobs to computers according to the info it collects
- **distributed initiation**: there is a load balancer on each computer that xfers its load to the others anytime it detects a load change, the local balancer decides according to the load info received where to start new jobs
- **random initiation**: when a new job arrives the local load balancer selects randomly to which computer to assign new jobs

Pre-emptive load balancing

- **random**: *when a computer becomes overloaded it select at random another computer to which to transfer a process*
- **central:** *one load balancer regularly polls the other machine to obtain load info. If it detects a load imbalance it select a computer with a low load to migrate a process from an overloaded computer*
- **neighbour**: (Bryant,Finkel *1981) each computer sends a request to its neighbours, when the request is accepted, a pair is constituted.If they detect a load imbalance between them they migrate some processes from one to another after that the pair is broken*
- **broadcast**: *server based*
- **distributed:** *each node receives a measure of the load of other nodes, if it discovers a load imbalance it would try to migrate some of its processes to a less loaded node*

Processor load measures

The metric used to evaluate the load is extremely important in a load balancing system. More than 20 different metrics are cited in only 1 paper(Mehra & Wah 1993):

- std Unix : # of tasks in the run queue
- size of the free available memory
- rate of CPU context switches
- rate of system calls
- 1 minute load avg
- amount of CPU free time

but the conclusion of many studies indicates that the simple rqueue is a better measure than complex load criteria in most situations.

Process migration mechanisms

User/kernel level migration

- Kernel level process migration (Mosix):
  - no special requirements, no need to recompile
  - transparent migration
- User level process migration (Condor,...):
  - compilation with special libraries (libckpt.a)
  - a snapshot of the running process is taken on signal
  - the snapshot is restarted on another machine
  - more details at http://hpc.sissa.it/batch

Process migration factors:

- **transparency**:how much a process needs to be aware of the migration
- **residual dependency**: how much a process is dependent on the source node after migration
- **performance**:how much the migration affects performance
- **complexity**: how much complex the migration is

Transparent:

- Location transparent
- Access transparent (or network)
- Migration transparent
- Failure transparent

System call interposing

In many cases transparency is obtained interposing some code at the system call level. There are different ways to do it:

- Very easy on message based OSs in which a system call is just a message to the kernel
- A modified kernel: in this case it can be completely transparent to the process (MOSIX), reasonably efficient for some kind of processes
- A modified system library: in this case the programs need to be linked with a special library(Condor), reasonably transparent

Process migration parts:

- execution state migration
- memory space migration
- communication state migration

Execution state migration

It’s not that difficult. For the most part the code can be the same as that used for swapping. In that case the destination is a disk while for migration the destination is a different node.

Communication state migration

It is a difficult task and usually it is avoided forcing the migrated process to redirect networking operations to the Home Node where they are performed.

On Amoeba each thread of a process has its own communication state which remembers the stage of the ongoing RPCs, these states are kept in the kernel

- during migration any message received for a suspended process is rejected and a “process migrating” response is returned
- sending machine will rexmit after some delay
- sending machine using a special protocol(FLIP) will locate the migrated process
- the same technique is used for reply msgs to a migrating client

Migratable sockets

Unfortunately TCP/IP has no direct support for this.
Research is ongoing on different solutions.
A proposed user-level implementation requires:

- substitution of the socket library
- relinking of programs with the new library

(Bubak, Zbik et al. 1999)
In this proposal communication is done between virtual addresses (there are servers that keep a correspondence table between virtual and real addresses). When an endpoint migrates, a stub remains on the original node to respond with a "process migrating" msg. The other endpoint then should consult the server for the new real address to contact.

Memory space migration

- **total copy**(Charlotte,LOCUS): process is frozen and the entire memory space is moved, the process is then restarted on remote (this can require many seconds and further it can transfer memory that will not be used)
- **pre-copying** (V System:Theimer): allows the process to continue the execution, while its memory space is being transferred to the target.Then V freezes the process on the source and re-copies the pages modified during the transfer time
- **lazy-copying or demand page**(Accent:Zayas): pages are left on the source machine and are transferred only at page faults. This mechanism can leave dirty pages on all the nodes the process goes through. A small variation used by MOSIX is to freeze the process during the transfer of all the dirty pages and then to transfer from the backing store the other pages when referred
- **shared file server**(Sprite): Sprite uses the network file system as a backing store for virtual memory. Dirty pages are flushed on the network file system and on the target the process is restarted (it's just aswap-out,swap-in)

focus on Mosix

MOSIX important points

- Probabilistic info dissemination
- PPM (Pre-emptiveProcessMigration)
- Dynamic load balancing
- Memory ushering
- Decentralized control
- Efficient kernel communications

Probabilistic info dissemination

- each node at regular intervals sends info on its resources at a random subset of nodes
- each node at the same time keeps a small buffer with the most recently info received

MOSIX load balancing algorithm

Mosix uses a pre-emptive and distributed load balancing algorithm that is essentially based on a Unix load avg metric. This means that each node will try to find a less loaded node to which it would try to migrate one of its processes.

When a node is low on memory a different algorithm prevails: memory ushering

Mosix info variables

These are the variables exchanged between Mosix nodes to feed the distributed load balancing algorithm:

- Load (based on a unit of a P II-400Mhz)
- Number of processors
- Speed of processor
- Memory used
- Raw Memory Used
- Total Memory

Look at http://hpc.sissa.it/walrus/mjm for a java applet showing these values over a Mosix cluster.

Memory ushering algorithm

Normally only the load balancing algorithm is active.
When memory is under a threshold (paging), the memory ushering algorithm is started and prevails the load balancing algorithm.
This algorithm would try to migrate the smallest running process to the node with more free memory.
Eventually processes are migrated between remote nodes to create sufficient space on a single node.

MOSIX network protocols

These protocols are used to exchange load info among the nodes and to negotiate and carry out process migrations between nodes. Mosix uses:

- UDP to txfer load info
- TCP to migrate processes

These communications are not encrypted/ authenticated/ protected therefore it's better to use a private network for the MNP.

(Unfortunately the sockets used by Mosix are not reported by standard tools: e.g. netstat)

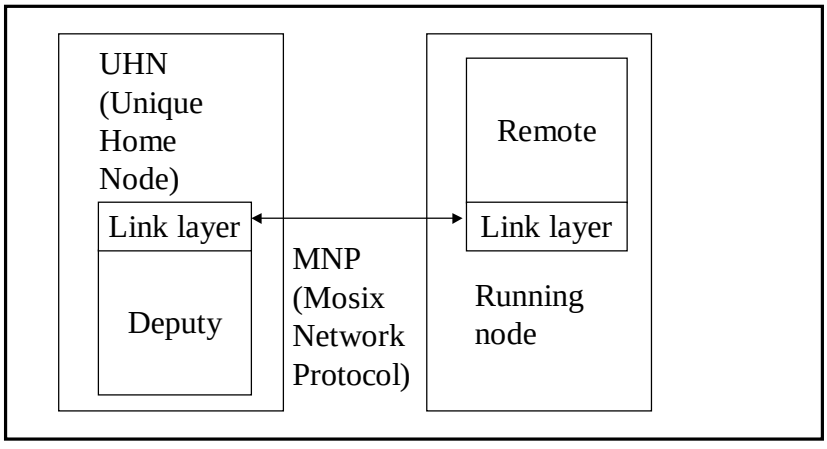
MOSIX process migration

- It's a **kernel level migration** (it's done without the necessity of any modification to the programs, because Mosix changes the linux kernel)
- **fully transparent** : the process involved does'nt have any need to know about the migration
- **strong residual dependency**:the migrated process is completely dependent on the deputy for system calls(and particularly I/O and networking)

Mosix migration transparency

- Mosix achieves complete transparency at the price of some efficiency interposing to system calls a link layer that redirects their execution to the originating node of the process(**Unique Home Node**) .
- The skeleton of the process that remains on the originating node to execute the system calls, receive signals, perform I/O is called in Mosix the **deputy** (what Condor calls shadow)
- The migrated process is called the **remote**.

MOSIX deputy/remote mechanism



Mosix performance/1

- While a CPU bound process obtains clear advantages from Mosix being very little penalized by process migration, I/O bound and network bound processes suffer big drawbacks and are usually not good candidates for migration.
- A first solution addressing only part of the problems has been the development of a daemon to support remote initiation (MExec/MPmake) of processes. With the help of this daemon processes that can run in parallel are dispatched during the exec call (you need to change the source and recode the exec as mexec) thus avoiding the creation of a bottle-neck single Home Node where all the I/O would be performed. **Gmake** is a tool already prepared to be recompiled to exploit its inherent parallelism (follow the indication in the MExec tarball).

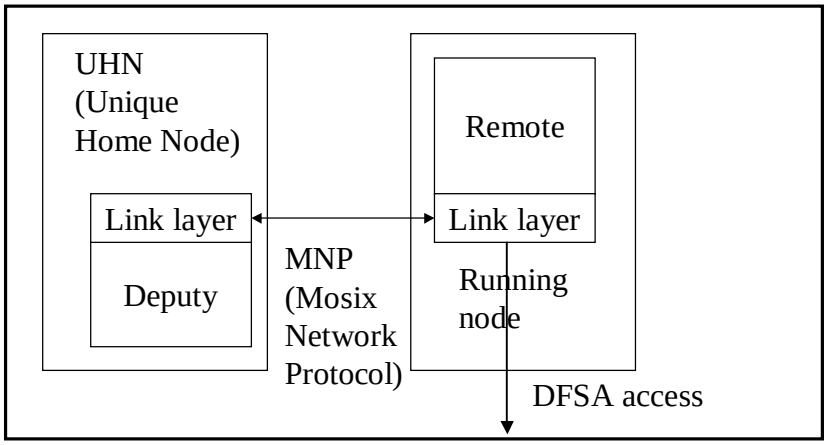
Mosix performance/2

A more general solution to the I/O bound processes performance problem under migration is anyway envisioned to be the use of a **cache coherent global file system**:

- **cache coherent**: clients keep their caches synchronized, or only 1 cache at the server node is kept
- **global**: every file can be seen from every node with the same name

If such a file system can be used then I/O operations can be executed directly from the running node without resorting to redirect them to the Unique Home Node. Mosix recently has incorporated the possibility to declare that a file system is a Direct File System Access.
For such file systems Mosix will perform I/O operation directly on the running node.

MOSIX dfsa (direct file system access)



Cache coherent global distributed file systems

To be a MOSIX DFSA it's not sufficient to be a cache-coherent global file system. In fact some additional functions (not available in std Linux super-blocks) must be available at the super-block level:

- identify,compare,reconstruct

A prototype DFSA that is just a small software layer over the std Linux file system is **MFS (Mosix File System)** provided directly by the MOSIX team. Other possibile candidate DFSA are:

- **GFS** (GlobalFileSystem)
- **xFS** (Serverless File System)

It seems now stable the integration of Mosix with GFS.