Execution Context Migration with ZOS

Roberto F. Ligeiro Marques and Felipe M. G. França
Universidade Federal do Rio de Janeiro/COPPE
{*ligeiro,felipe*}*@cos.ufrj.br*
Vítor Santos Costa
University of Wisconsin-Madison and Universidade Federal do Rio de Janeiro/COPPE
vitor@biostat.wisc.edu

Abstract

It is nowadays common to find users that have to use different machines at work, home, and travel. Such users often spend significant amounts of time sychronising and restarting their work environments. Often, they eventually have to cope with inconsistent data at different locations. ZOS (Zombie Operating System) proposes that users should have a main execution context (or *anima*) containing not only the user's data, but also application images. Ideally, the *anima* should reside in a small server, the *master*. In ZOS, masters take over other machines, the *zombies*, and then take advantage of their resources, such as better CPU, better interfaces, more disk, or extra connectivity. We describe our first implementation of ZOS, which, we believe, demonstrates that the idea is practical and worthwhile.

1 Introduction

It is nowadays common to find users with a personal computer at home, a workstation in the workplace, a laptop to travel, and a palmtop for light work. Such users often find synchronising data between different computers in very different working environments to be difficult and error-prone. Moreover, quite often they will spend significant amounts of time just restarting their work environment, either because they are using a new machine, or just because they may have to restart several applications every time they start up.

Arguably, the major problem is data synchronisation. One problem about current tools [12, 10, 35, 8, 36, 25] is that, unfortunately, they are not entirely transparent. One issue is just the delay in synchronising through the network: hard-pressed users will just go ahead and use their applications, thus making synchronisation much harder. A second issue is that such tools can be used to break protection boundaries (often unwittingly). An alternative which is gaining popularity is to always rely on a central Internet connected server [34, 26, 4, 30]. Note that such platforms rely on excellent, always-on, connectivity. Moreover, they do raise questions on privacy and data-ownership issues.

Progress on component miniaturization, namely the ever-shrinking RAM and Hard-Disk, suggests a third alternative. Instead of synchronising with huge servers or between peers, why not make each user responsible for hers or his own data? Users would have their own servers, ideally small devices [37, 38], and these devices should be able to transport the user's whole *execution context*. By execution context, we mean not just the user's data, but also application state, so that users could quickly recover their environment, instead of having to always go through the painful application bootup cycle.

The ZOS project aims at reaching that goal. ZOS (Zombie Operating System) is based on the idea that a machine contains an user's execution context. We call such a machine the *master*. Masters take over other machines, the *zombies*. Execution on a zombie is *dominated* by the master: graphical applications will be a copy of what is going on the master, whereas batch jobs may temporarily migrate to a brawnier slave. Zombifying a machine should be fully transparent. To achieve these goals, we need to design a communication protocol, update applications and or operating systems, while considering performance and security issues.

Our goals are more ambitious than process migration tools, such as Condor [16], Nomad [5],

Libckpt [23] or Zap [19]. Migration is a part of ZOS, as we shall discuss, but we fully want to duplicate the working environment from the main server to the host. Last, ZOS is not a remote environment control system, such as VNC [27]. We do not just want to peek over a remote machine, we want to take advantage of the zombie's abilities, as much as we possibly can. On the other hand, process migration and remote environment control are a part of ZOS, and we shall try to use currently available technology as much as possible.

Next, we describe ZOS in some detail. We first introduce the major components of ZOS design. Next, we describe our first prototype. We then report some performance data, compare with related work, and present our conclusions.

2 The Design of ZOS

It should be by now clear that ZOS sees the world as divided into two sorts of machines:

- *Masters* – machines that contain the user's environment, and that will take over other machines. A master contains an *anima* with the user's execution context.
- *Zombies* (or *slaves* – hosts that the master needs to take over, either because they have more computing power; or because they have a faster connection; or because they can access other data (e.g., work data); or just because they have a better screen and keyboard.

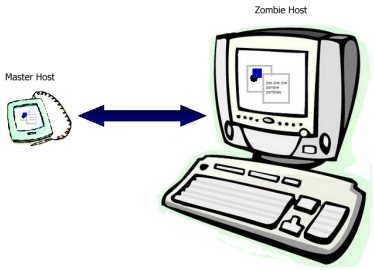

Figure 1: Zombie Operating System

In a nutshell, the key idea of ZOS is that a Master spends some of its time scanning for machines that it can *dominate*, that is, that it can export its *anima* to. We call such a process *discovery*. During discovery, the master will extract information from the possible zombie and decide on the type of domination. We envisage two forms of domination. Domination may be *complete*, where ZOS would fully take over the slave machine, exporting an operating system and restarting the hardware; or *partial*, where ZOS relies on the slave's operating environment.

At finding a zombie, ZOS initiates a two step process:

1. The master sends to the zombie the relevant components of its working environment. It will share some of its storage, and it will move some of its applications (tasks), including their graphical interface.
2. The Master and the Zombie work together in a step-by-step fashion. That means that zombies operating on an application will ripple through to the master (and vice-versa). Such effects should not be immediate: the user should see the same context or *anima*, but now available from different machines.

   Clearly, security is a fundamental issue here. At least, secure authentication must be used. Cryptographic techniques would be required in an unreliable world. Trust is an issue: can we believe a zombie is really a zombie?

Last, notice that the roles of master and zombie need not be hardwired, and that a master may eventually have several available zombies.

The major questions in ZOS should by now be clear:

1. What is and how to share the *anima*.
2. How to keep the user's work safe.
3. How to guarantee acceptable performance, as otherwise the system will be useless.

First, we focus on defining what is the *anima*. Abstractly, a *anima* is static information, that is, a partial mapping over a set of files containing data, plus dynamic information, a set of tasks or processes. Each task or process may be at a specific state which will include handles to the static information, plus an interaction context, usually represented as a set of windows over some windowing system.

Both tasks and file systems issues are fundamental to ZOS. Tasks are particularly important because they describe what we expect from such a system. We have different expections for different kind of tasks, say, we just want performance

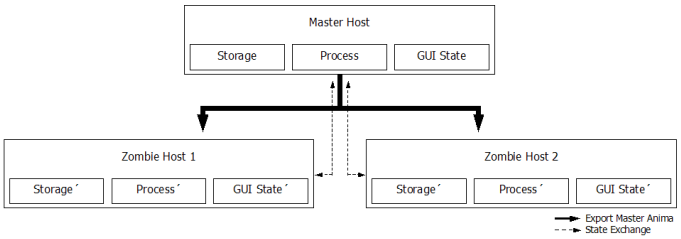

Figure 2: ZOS Anima

from a batch task, but we want consistency and interactiveness from a GUI application. We thus believe that a one size fits all is not adequate to represent all tasks in ZOS. Instead, we classify tasks as being either:

- Shared – these are tasks that logically should be executing both at the master and the zombie(s). Examples include interactive applications, such as a terminal window, a web browser, and office-like applications.
- Migrant – these are tasks whose home is at the master, but who can temporarily migrate to a zombie. A typical example would be a background process which requires heavy computational power. Such a process can perform better at a zombie, but will go back to the master when domination ends.
- Decoupled – these are processes that may execute detached from the master, but that communicate with the master and/or the slaves if it is connected. One example would be processes running on a grid.
- Kernel – tasks that manage ZOS or that perform critical user operations, and which should only run at the master.

Second, we discuss the major security problems in ZOS. Four issues are crucial:

- File Systems – one must check and control how files and directories are exported and imported by master and by zombies. The user should be able to define which files/directories he want to export and in what context.
- Communication Protocols – Data traveling through the network should be protected.
- Trust – Is someone snooping our data? Are we leaving something behind? Is a zombie really a zombie? How much can we trust a zombie, and if we cannot absolutely trust, how to cope with it?

In general, we would start by assuming a context where we can mostly trust. Ultimately, for ZOS to be useful we must be able to work in situations where both the communication medium and the slave system will not be trustworthy.

Last, it comes without saying that acceptable performance must be guaranteed, otherwise ZOS would be worthless. As we shall see, shared tasks are critical in this case, as they required most communication.

A full implementation of ZOS thus touches many research areas in Operating System technology, such as migrating processes [22] [19] [23], distributed computation [16] [1], trusted computing [20, 9]. In this work, we report on an initial prototype. We believe that the initial design demonstrate the usefulness and practicality of the key concepts. Throughout, we tried to use available technology when possible. We believe that the paper's contributions are a proposal for a new approach to managing user's data and tasks, and the demonstration of the proposal feasibility through the implementation and evaluation of different approaches in the prototype.

3 ZOS Architecture

Next, we discuss the major issues on the first ZOS prototype. We focus on system organisation, actual implementation issues are presented in Section 4.

ZOS assume several computing systems connected through an underlying network. Systems are classified as either being:

- Free, that is, they are neither masters nor zombies.
- Connected - they are either masters or zombies.

A free system may become a master and/or a zombie depending on how the user have been the host configured. Indeed, it is possible for the same system to be both a zombie and a slave, which has at least been useful for debugging purposes.

Domination is the process where a system becomes a zombie to some master. Dominations assumes that systems are pre-configured to work as masters and or zombies. As we have explained, domination may be complete, or may only be partial. This largely depends on the access permissions that are activated in the zombie host, and on the trust we have between machines.

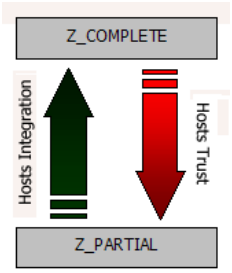

Figure 3: Control Transparency

An implementation of ZOS may thus be divided into two subsystems: one supports master mode execution, and the other supports zombie mode. Both subsystems have a similar structure, as they must work together. They are organised into four components, managing:

- Slave Discovery and Setup.
- File System.
- Process.
- Graphical Interface.

We shall assume that in all implementations of ZOS communication between master and slave subsystems must be encrypted. This is a minimal security requirement. Table 1 shows these modules graphically.

Table 1: ZOS Modules

Security Modules and Functions			
Slave Discovery	FS Manager	Zombie Listening	FS Manager
Process Manager	GUI Manager	Process Manager	GUI Manager
Master Subsystem		*Zombie Subsystem*	

Zombie Management Slave Discovery and Setup is performed by the *SlaveDiscovery* master module and by the *ZombieListen* zombie module. *SlaveDiscovery* searches the current local network for possible target-zombies and performs domination. The *ZombieListen* module runs on systems that are predisposed to become zombies. It receives domination requests from a master, and authenticates them against a local database. The two modules work together while the session is active, and they are also the ones that actually close down a domination.

Users eventually ask for termination. The request is sent to the *SlaveDiscovery*, which in turn notifies the *ZombieListen*. Each *ZombieListen* carries shutdown activities at its node. These activities follow three steps: closing down of each other module, e.g., access to network file systems should be disconnected, cleaning up, and session detachment. Cleanup should guarantee that no master data will remain in the zombie. In the simplest case this would require removal of temporary files and of or consistency checks. Secure environments will require local file system checking, memory checking, and a machine reboot.

File System Management We use distributed file systems so that users can export some points in their file system to slaves. Users should also be able to use the zombie's file system, so that the master can take advantage of extra disk space or of better connectivity. The File System Manager is responsible to start sharing at domination, to disable sharing at disconnection, and to control the actual amount of sharing that goes on.

We would like the file system manager to be very fine-grained. An user may have directories which should not be exportable to some zombies, but which may be exportable to others. Some user directories might never be exportable. It may also be undesirable to allow some information to get in the master, e.g., confidential work data should be removed from the system at leaving the work environment.

The file system manager also serves ZOS by providing shared storage for system state.

Process Management Every ZOS aware process must have a type, which indicates how to perform during domination. As previously discussed, ZOS processes will be shared, migrant, decoupled or kernel. In this prototype we shall not discuss decoupled processes, as they are not

a core issue in ZOS. Kernel processes include all ZOS tasks that run in the master.

Shared processes must be copied into the zombie and must be synchronised during domination. Migrant processes must be transferred to zombie and return at disconnection. Decoupled processes must be informed that they can communicate with the zombie, and must be allowed to upload to the slave.

GUI Management Maybe one of the hardest issues is how to keep the graphical interface of master and zombies synchronised. Input devices at the slave (namely mouse and keyboard) must act as if placed at the master. Shared processes must display consistently between master and slave. This is the major issue we discuss in our prototype.

4 The First ZOS Prototype

A first prototype for ZOS has been built based on GNU/Linux Red Hat9, kernel 2.4.20. A set of processes/threads controls each ZOS modules presented in the previous section. Figures 4 and 5 illustrates the basic structure of the prototype.

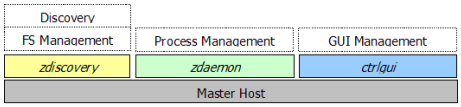

Figure 4: Master Modules

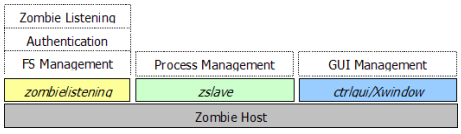

Figure 5: Zombie Modules

Slave discovery, domain management and file system management is carried out by the `zdiscovery` application. The `zombielisten` daemon deals with the identification of zombie hosts, user authentication and the management of the slave file system. Process management is performed by `zdaemon` in the master hosts, and by `zslave` in the zombies. The master also requires `ctrlgui` to perform GUI control. Otherwise, the current prototype relies on the X windowing system [28].

4.1 Remote Sessions

The main objective of ZOS is to allow one to reopen an execution context in a remote host machine which would is not in use by another user. This way, our first goal is to be to able to open an execution context, e.g., an ongoing desktop, over a previously established working environment. To our knowledge, the best support for functionality close to our goals is available from the *Gnome* toolkit. More specifically, we rely on Gnome's `gdmflexiserver` [21]. This application allows one to open an user session as a new window over an already opened session. `gdmflexiserver` itself relies on the X windows `Xnest` service [14].

We modified `gdmflexiserver` to support the ZOS' authentication protocol. Once a user gets authenticated by ZOS (how this is performed will be shown in detail in the next section), `gdmflexiserver` receives a request from the corresponding `zombielisten` daemon and immediately opens a session. Furthermore, once the graphical session has been opened, `gdmflexiserver` informs the `zslave` daemon to request migration for the execution context.

4.2 Zombie possession protocol - ZAP and ZIP

Migration of a execution environment is carried out in two distinct phases performed by *ZAP – zombie attraction protocol* – and by *ZIP – zombie install protocol*. The first phase is actually the default state for a master. In this phase, a master is on the lookout for target devices/machines until an initial contact with such devices is done. The second phase consists of remote session opening and transfer of the execution context.

The ZAP Protocol Migration requires finding target slave hosts throughout the network. To do so we trust the `zdiscovery` application. This application currently sends broadcast packets through the local network. Each packet contains user information, device information (hostname), and also information about the intended level of control, whether *zcomplete* or *zpartial*. It is up to the user to request the level of control wanted. These configurations are kept in a text file. Application `zdiscovery` provides a graphical user interface to change it but the user can also edit it using a generic text editor.

Machines configured as possible slaves snoop the network looking for *zdetect* packets. This is performed by running `zombielisten` as a daemon. On receiving a *zdetect* packet, a machine starts the authentication process based on the `<user,hostname,control>` tuple. The current prototype uses Unix style authentification. Then, in case user/password gets authenticated, one checks if such user is enabled for the level of control intended for that host. File *zpasswd* contains such information, which is organized as `<user:hostname:control>`. If a valid user/host asks for complete control and has not that status in that particular host, partial control is ensured as default.

In order to save time due to the preemptive nature of the authentication process, module `zombielisten` caches a list of `<user,hostname,control>` tuples from previous uses, so that repeated tuples do not get through the full authentication process twice. In order to allow that new configurations of the *passwd* and *zpasswd* files update tuples already authenticated, the `zombielisten` module periodically refreshes the tuple list.

After user authentication, `zombielisten` sends a *slave_detected* packet to the master host containing the slave description, i.e., machine user, hardware description, and hostname. The `zdiscovery` interface then presents users with a list of available slaves. Right now, it is up to users to decide on what set (one or more) of hosts to dominate. Once a host is chosen for domination, `zdiscovery` sends a *znotify* packet to the target host, triggering the machine submission. In case the target slave machine is still available, a *znotify_ok* packet is returned back to the master and the domination process starts.

File System The file system mounting is the first step of the domination process. To do so, `zdiscovery` sends a *nfs_mount* packet to `zombielisten`. This packet informs which points of the master's file system should be mounted in the slave host. The export/mounting of the file system is realized by the *zdiscovery/zombielisten* applications through the use of primitives offered by the NFS file system [31]. We currently rely on ZAP for host and user authentification. The users also have the choice of encrypting NFS traffic by using SSH tunnels (IPSec will be available in the future).

Independently of the control level established (complete or partial) at a given domination process, at least one point of the zombie's (master) file system – by default `/mnt/zos/` – should be exported and mounted in the slave host, where the user files in use are stored. The user can specify other points of the file system to be shared through `zdiscovery`.

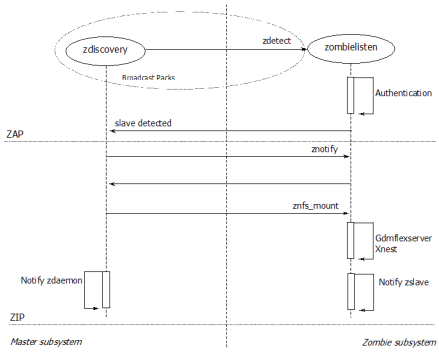

Figure 6: ZIP and ZAP Protocols

Once the file systems are mounted, `zombielisten` starts the `Xnest` application authenticated by the user sent by `zdiscovery`. Next, `zdiscovery` notifies `zdaemon` about the hostname of the slave machine. Then, `zdaemon` and `zslave` start copying the user's working environment in the target slave host, in such a way that the machines involved can guarantee consistency.

4.3 Process Management

Process control is arguably the most critical area of ZOS. As we explained above, we focus on shared and migrating processes. Process management is carried out by both `zdaemon` at the master, and by `zslave`, at the zombie slave.

Shared and migrating processes originally execute in the master host. At the beginning of zombie possession they should be replicated and/or migrated to the zombie host.

Every shared or migrating process must be registered with `zdaemon` first. To do so, we link such processes to a process loader, the `preloader`. In order to initiate a process, `preloader` first registers the process with `zdaemon`: given the process' name and type the `preloader` initiates it with the appropriate *role* and notifies `zdaemon` about the new process. The `zdaemon` daemon thus keeps a list of processes executing in the master, including processes' type – shared or migrating – and state – exported or not exported.

Upon receiving a message about the start of zombie possession, `zdaemon` establishes communication with the `zslave` daemon at zombie. The first step is to send a list of participating processes, the `ZDAEMONLIST`. All shared processes are exported to the slave host. Migrating processes are just exported once, currently to the first dominated slave. The slave's `zslave`, upon receiving `ZDAEMONLIST`, dispatches each process according to their respective role.

Shared processes initiated during a zombie possession are immediately informed to `zslave` by `zdaemon` (`ZDAEMONAPPENDLIST`), after registration of the corresponding application. By the end of a domination, `zslave` takes charge of redirect migrating processes which could have been sent to the slave host (and that are still running) back to the zombie host.

4.4 Shared Processes

Shared Processes are available both at at the master and at the zombie: updates to the state of a shared process in either machine are recognised in both processes. A typical example would be a shared text editor. Typing at the slave should be immediately visible at the master, and vice-versa. Domination thus must guarantee that the editor is open at both hosts and that all events recognised by each process must be recognised by the other (e.g., keyboard, mouse, and so on).

Next we report on our experiments with the execution of shared processes. We followed two different approaches:

1. Port the application to be ZOS-aware: that is, actually run two processes and make them keep in synch.

2. Execute the application in a single machine but under a single GUI manager, so that it appears as if executing on the slave.

The first approach is closest to our goal in ZOS. Namely, it allows users to take advantage of the extra capabilities of the zombie slave, say more memory for larger spreadsheet calculations or access to private networks. On the other hand, it requires considerable effort to adapt the application to our needs. The major issues are application startup and synchronisation. Application startup is hard because we need to start the application on the zombie with a very accurate simulation of the master. Synchronisation is surprisingly simpler, as we can always manage the streams for both slave and master so that they receive much the same input.

The second approach can take an application as is. Essentially, we always run the application on the master, but we allow its interface to be exported to the zombie slave. We have used the well-known VNC system [27] to implement this approach. In this case we do provide some of the advantages of ZOS, such as quick setup, but we cannot take advantage of all resources available at the master. We experimented the first approach with *gnome-terminal*. This is a non-trivial application with a limited (but existing) graphical interface. We experimented the second approach using VNC.

4.4.1 Truly Shared Processes: Gnome-terminal

An important case of shared processes are terminals. These processes receive text input and output the result of system commands. They thus reflect the state of the machine they are executing on. In this case, it makes sense that commands typed at the zombie will execute at the master. Thus a shared process will give you the status of the master, and never the slave (of course, users are still allowed to have local processes at the slave).

The gnome-terminal application is the major terminal manager for the well-known Gnome toolkit. Although a text-based application, it does support *xmouse* events and it does manage a graphical context, albeit limited. We define three roles for this application: the *master*, the *slave*, and the *inslave*. The *master* role corresponds to a shared process launched at the master. Given the flag `--szos master`, gnome-terminal will take this role. In this case, the first step is to contact the `zdaemon` telling him that there is a new gnome-terminal around, and that this new process must be migrated at domination. Gnome-terminal does not require pre-loading: the application talks directly with `zdaemon`.

`zdaemon` next registers the new process and sends extra parameters for execution. The parameters are: a place to store data to be exported at domination, a place to store the terminal's *diff*, that is, all input received after domination started but before the slave is actually executing, and pipes to control the application streams. In the case of gnome-terminal the data to export at domination is basically a buffer with the terminal's history (such buffers may in fact grow to be quite

large).

At domination, `zdaemon` gives the zombie's `zslave` a list with shared processes. The list contains the parameters required to execute the slave's gnome terminal, that is `--szos slave`, plus handles to the remote terminals connected to the zombie's terminal via the history, diff and pipes. We currently use NFS for this purpose.

The interaction during domination follows the following algorithm:

- The slave opens the history and sets up a matching display
- A *thread* in gnome-terminal will periodically check the diff, send it out to its display, and clean it up. Master and slaves synchronise through a lock.
- A second *thread* at the slave periodically reads an input buffer and sends it out to the pipe. A *thread* in the master gnome-terminal in turn reads and empties the remote buffer, executing corresponding commands.

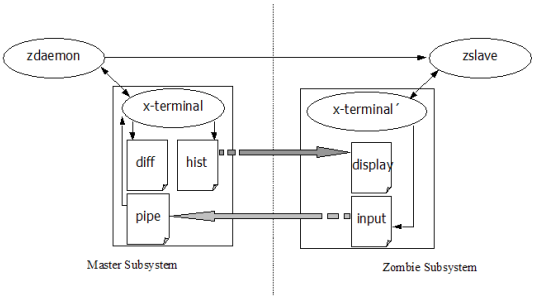

Figure 7: Shared Processes: Gnome-terminal

New ZOS gnome-terminals may be created at any time, even during dominion. They are immediately exported to the slave.

The last role that gnome-terminals can play is *inslave*. The *inslave* option is intended to cover the alternative of opening a gnome-terminal in the slave during a domination. In this case, a request for opening a new gnome-terminal must be sent to `zslave` which, by its turn, will redirect the request to `zdaemon`. From this point on, `zdaemon` will start the installation of the new terminal as usual.

4.4.2 Virtually Shared Processes

Our second approach requires much less effort: we mainly guarantee that an application at the master can interact with different devices at different hosts. This particular problem has been the subject of significant effort, and we shall take advantage of previous work for the VNC system [27], designed to allow remote control of a host. We briefly discuss the key ideas in VNC, and then we explain how to use them in ZOS.

The key idea is to use an hidden display for VNC. Virtually shared applications work with this display, which is then distributed by the VNC client, *vncviewer*, to other machines.

In ZOS, applications are configured as virtually shared through the GUI interface, or through directly editing a configuration file. Virtually shared applications must use the services of the `preloader`. This application will set up a VNC file saying that the application must be started by a new VNC server, and that it will be shared by several VNC clients. When the process starts, the preloader starts a new VNC server for this specific application and notifies ZOS about process startup and the server's display. Note that each application will have its own VNC server, and thus its own virtual display.

To make the application visible in the master host, `zdaemon` just has to start up a VNC client using the server given by the preloader. The application is now available to the server's display. The process is repeated for every zombie. At domination `zdaemon` sends out a list of available VNC servers, and `zslave` starts up a VNC client for each application. Each process thus has one local *vncviewer* client at the master, and n remote clients for each zombie, where n takes the number of slave hosts.

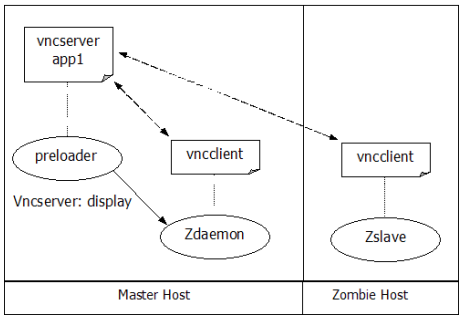

Figure 8: Shared Processes: VNC

4.5 Migrating Processes

Migratory processes are those that move to the zombie at domination. When domination ends

they would return back to the host. Example of such processes include not only computationally intensive applications, but also applications that may benefit from a zombie's larger bandwidth. In our specific case a motivation would be machine learning tasks that would run slow on a laptop, but that could take advantage of larger machines, even if only temporarily.

Several approaches to migrating processes are available in the literature. EPCKPT [22] does not require recompilation of applications, but it does require changes to the operating system kernel. Alternatively, Condor [16], libckpt [23], and the Dynamite chekpointer [11] do not require operating systems updates, but force recompilation against a library. Zap [19] is a particularly interesting approach in that it does not require kernel or application modification.

In our current prototype we use the Dynamite checkpointer. We use Dynamite because it is well supported and because it works with our Linux setup. Dynaminate uses the signal `USR1` to inform processes that they must be checkpointed. At receiving the signals, processes will die storing an executable image in disk. To continue the process it is sufficient to execute the image on (as long as one uses the same OS and Kernel version).

We use this mechanism as follows: at domination, `zdaemon` sends each migrating processes receive a `USR1` signal, and the processes dump their image on a temporary directory. The images are then collected and executed by `zslave`. The process is reversed when domination ends. `zslave` will send a `USR1` to the processes, and `zdaemon` collects their images

4.6 GUI Control

Managing the Graphical User Interface is fundamental in ZOS. The `ctrlgui` daemon performs this task. `ctrlgui` is activated from the `zdiscovery` manager, and it is responsible for guaranteeing that dominated hosts have their Input/Output synchronised. The key idea is that all events at the master host are directed to the zombie host. Therefore, all input to ZOS-aware applications is at the slave: the user may keep on using the master as a workstation. This avoids input synchronisation issues between different machines. Note that all events directed from a master host to a zombie host are restricted to the `Xnest` session opened by ZOS. This limits the actions of a ZOS user over a dominated session to his own environment.

We rely on Xwindows functionality to perform this task. Namely we use the libraries Xlib e Xtest [33, 29, 7, 17]. These libraries take advantage of the fact that all communication with the Xwindows server is through streams, so it is relatively straightforward to direct events elsewhere, and to receive events from other sources.

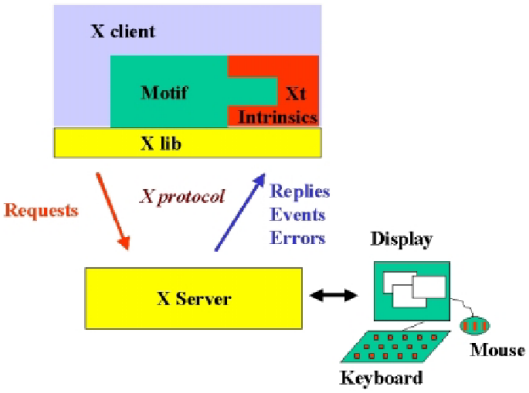

Figure 9: X Window System

5 Results

To evaluate whether ZOS would be feasible, we measured ZOS performance under sever difference performance loads. Results were obtained in a the Laboratório de Arquiteturas e Microeletrônica (LAM). We used as master a notebook AMD Athlon XP 2.6 GHz with 512 MB main memory. The zombie was a desktop Pentium 4 2.4 GHz with 512 MB main memory. The network is a relatively slow switched 10 Mbps Ethernet.

We first measure the amount of time it takes to migrate an environment from master to zombie. The total time can be divided into three components: (i) zip/zap time, zt consists of user authentification and setting up the file systems; (ii) graphical setup time, xt, consists of the time required to set up the zombie's graphical interface, in this implementation using `Xnest`, and last; (iii) we consider the amount of time pt we need to migrate processes in the dominion. The total time tm given for setting up a zombie is thus:

$$tm = zt + xt + \sum_{k=0}^{n} pt$$

Where n is the total number of processes in the domain. This number thus ranges over all shared

and migrant processes. We would expect zt to be independent of n, depending instead on network load and on the total number of file system points to export. We also observe that in the current prototype processes may only migrate after the graphical environment is setup.

We estimate xt, the total time spent setting up the zombie's graphical environment to be constant. To obtain an actual value we instrumented the application `gdmflexserver` to report on how much time it takes to create a graphical environment. Our results indicate total time to be close to 10 seconds. In real life, xt varies with different machines, and with different environments.

Our first results are thus obtained by experimenting with the migration of shared processes. The number of shared processes ranges from 1 to 16. Figure 10 presents the results.

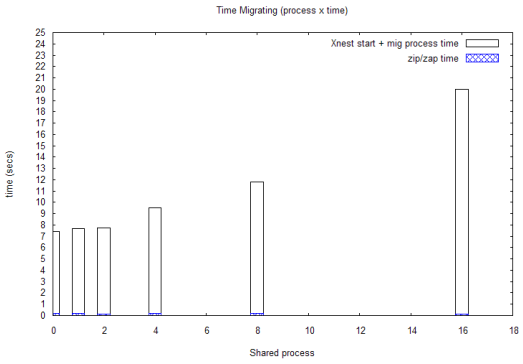

Figure 10: Migration Time (time x no. processes)

Figure 10 shows migration times around 20 seconds, which is observable but still much less than what would be required to manually setup an environment. Unsurprisingly, xt is the major contributor to system setup time. In contrast, zip/zapping time is almost negligible. The system copes well with a significant increase in shared processes: going from 1 to 16 processes results in a two second slowdown, on a relatively slow network.

The second experiment consists of migratory processes. Again we experimented with up to 16 processes. Figure 11 shows the actual results.

The results are similar to the first experiment. Times are still dominated by xt. We also remark that the actual overhead of sending migratory grows a bit faster, as we must send the full process.

The last experiment shows performance with an equal mix of shared and migratory processes,

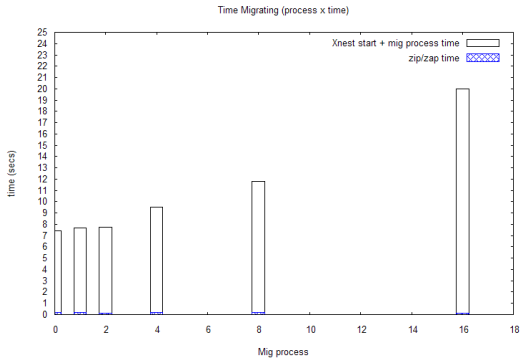

Figure 11: Migration Time (time x no. processes)

ranging from 2 up to 16 processes. Figure 12 shows the results. Actual overhead is dominated by xt, and growth is dominated by the migratory processes.

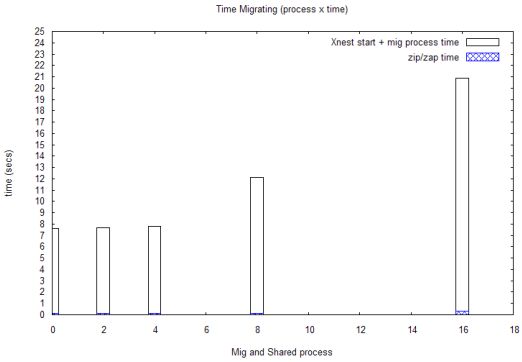

Figure 12: Migration Time (time x no. processes)

The time to shut-down a slave is similar to the time to create/dominate a new one. As expected, process migration overhead is similar in both directions. However, a small difference in time, favouring master-to-slave process migration times, could be observed due to the way the file system mounting is done, i.e., in the master-to-slave migration case the copying of a process image is performed locally and then exported to the slave. On the other hand, in the case of processes returning from slaves to the master, the copying of a process image is done in a remote machine, i.e., in the master file system.

6 Conclusions and Future Work

We propose a system that can distribute an user's computing environment to other machines. Our motivation was the authors' own problems in trying to use several computing platforms with dif-

ferent characteristics and have the same environment throughout. What happens when one started work on the laptop and the plane is landing? What happens when we are writing a project, and the bell rings to fetch the kids from school? The key idea of ZOS is that there is an user environment which we should be able to move around. Ultimately, one should be able to take advantage of progress in the miniaturization of storage to contain such storage in a very small device.

To our knowledge, Intel's personal server project [37, 38] is closest to our work. We believe our goals are more ambitious: we intend to move the whole environment to the personal server, whereas their project considers other machines as mainly Input/Output devices. A different approach to our problem is to have all storage in the Internet, say as in the Oceanstore project [26, 4, 30]. We do not feel comfortable with such approaches for personal data.

The whole ZOS project is rather ambicious. Our approach was to first implement really essential functionality, that is, process management, throughout using available tools. Our hope is that such a prototype can be useful by itself and to motivate future work. This has led us to focus on shared processes, namely interactive applications. Interactive applications are a hard problem, but our experiments with two different approaches show that our ideas are practical, and relevant to users.

In the continuation, we will focus on three major issues: security, improved file systems, and improved sharing operations. Next, we discuss each one briefly, starting with security. Ultimately, taking over a machine requires some amount of trust from both sides. Security issues cut through all this work, but in the case of the master, three issues are paramount: communication should not be tapped; data should not remain on the machine; and, the slave should be what it says it is. Encrypting communication is a well known problem, and we can take advantage of well-known technology ranging from tunneling to IPSec. We would like to require for data not to remain in the zombie slave, or if it does to be unreadable. Progress in this component will be based on the huge amount of work on encrypting file systems [39, 6]. More recently, there is interesting work on encrypting memory itself [24].

Regarding file system issues, our main goal will be to allow users to fine tune how much and what they want to export. This feature will require a grammar and a graphical interface. Implementation should not be too difficult: we can use annotations over standard file systems such as EXT2 [2], or we can try to develop a kind of translucent file system.

Improving process migration would include not just to experiment other alternatives to checkpointing, but how to solve some questions concerning when and which processes should be migrated, including the exchange of previously migrated processes among slaves. This problem relates to the issues discussed in Agile [18], which proposes a dynamic approach to the adaptation of applications (in terms of resource usage) running in mobile devices. Regarding process syncrhonisation, our current work is to try to use an extra layer over Input/Output so that applications can actually run on the slave with little change. This work will expand *ctrgui*. We have so far avoided changing OS functionality. Our more ambicious goals are to be able to copy masters to machines where one may not trust the available Operating System. One such approach would be to simply copy the whole master's OS, as long as one can trust the underlying hardware [15]. A middle-of-the road alternative would be to extend work on the migration of virtual machines [32], although this would also require trusting on the Virtual Machine Manager [13, 3].

7 Availability

Complete information about ZOS, including the sources for the first ZOS prototype is available from the ZOS' home page:

`http://www.cos.ufrj.br/~ligeiro/zos`

Acknowledgments

The authors would like to thank Edil Fernandes for kindly allowing access to the LAM's computational resources. Roberto F. Ligeiro Marques was supported by a CNPq grant. We also want to thank Anderson Borges for some discussions in the early design of ZOS.

References

[1] Amon Barak and Oren LaZadan. The mosix multicomputer operating system for high performance computing. *Journal of Future Generation of Computer Systems*, 3(1), March 1998.

[2] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. http://e2fsprogs.sourceforge.net/ext2intro.html.

[3] Grzegorz Czajkowski, Laurent Daynès, and Ben Titzer. A multi-user virtual machine. In *Proceedings of USENIX Annual Technical Conference*, June 2003.

[4] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the eighteenth ACM Sysmposium on Operating Systems Principles*, 2001.

[5] Eduardo Souza de Albuquerque Pinheiro. Nomad: Um sistema operacional eficiente para clusters de uni e multiprocessadores. Master's thesis, Federal University of Rio de Janeiro - UFRJ/COPPE, 1999.

[6] Roland C. Dowdeswell and John Loannidis. The cryptographic disk driver. In *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX Track*, pages 179–186, June 2003.

[7] Kieron Drake. *XTest: Extention Protocol*. X Consortium Standard.

[8] Dane Dwyer and Vaduvur Bharghavan. A mobility-aware file system for partially connected operation. *Operating System Review*, 31(1), January 1997.

[9] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Sysmposium on Operating Systems Principles*, pages 193–206, October 2003.

[10] P. Honeyman and L.B. Huston. Comunications and consistency in mobile file system. Technical Report CITI 95-11, Center for Information Technology Integration - University of Michigan, 1995.

[11] K. A. Iskra, G. D. van Albada, and P. M. A. Sloot. The implementation of dynamite - an enviroment fot migrating pvm tasks. Technical report, Informatics Institute, Universiteit van Amsterdam.

[12] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: A toolkit for mobile information acess. In *Proceedings of the Fifteenth ACM Sysposium on Operating Systems Principles*, 1995.

[13] Samuel T. King, George W. Dunlap, and Peter-Peter M. Chen. Operating system support for virtual machines. In *Proceedings of USENIX Annual Technical Conference*, June 2003.

[14] George Lebl. *Using and Managing GDM*.

[15] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the 19th ACM Sysmposium on Operating Systems Principles*, pages 178–192, October 2003.

[16] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the unix kernel. In *Proceedings of USENIX Winter conference*, 1992.

[17] George Sachs Mark Patrick. *X Input Device Extention Library*.

[18] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of 16th ACM Sysmposium on Operating Systems Principles*, October 1997.

[19] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: A system for migrating computing environments. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*. USENIX, December 2002.

[20] David S. Peterson, Matt Bishop, and Raju Padey. A flexible containment for execution of untrusted code. In *Proceedings of 11th USENIX Security Symposium*, 2002.

[21] Martin K. Peterson. *Gnome Display Manager Reference Manual*.

[22] Eduardo Pinheiro. Truly-transparent checkpointg of parallel applications. Technical report, Federal University of Rio de Janeiro - UFRJ/COPPE, 1999.

[23] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent chekpointing under unix. In *Proceedings of USENIX Winter conference*, 1995.

[24] Niels Provos. Encrypting virtual memory. In *Proceedings of 9th USENIX Security Symposium*, 2000.

[25] David Rasch, Randal Burns, and Johns. In-place rsync: File synchronization for mobile and wireless devices. In *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX Track*, pages 91–100, June 2003.

[26] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weathrspoon, Ben Zhao, and John Kubiatowicz. Pond: the oceanstore prototype. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2003.

[27] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2, January/February 1998.

[28] James Gettys Robert W. Sheifler. *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD.* X Consortium Standard.

[29] James Gettys Robert W. Sheifler. *Xlib: C Language X interface*. X Consortium Standard.

[30] A. Rowstron and P. Druschel. Past: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of HotOS VIII*, November 2001.

[31] Russel, Sandberg David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the sun network filesystem. In *Proceedings of USENIX Conference*, 1985.

[32] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica, S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*. USENIX, December 2002.

[33] Robert W. Sheifler. *X Window System Protocol*. X Consortium Standard.

[34] Edward Swierk, Emre Kiciman, Vince Laviano, and Mary Baker. The roma personal metadata service. In *Proceedings of the Third IEEE Workshop on Mobile Computing Systems*, 2000.

[35] Carl Tait, Hui Lei, Swarup Acharya, and Henry Chang. Intelligent file hoarding for mobile computers. In *ACM Conference on Mobile Computing and Networking (Mobicom'95)*, 1995.

[36] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, Department of Computer Science, June 1996. (http://rsync.samba.org).

[37] Roy Want, Trevor Pering, Gunner Danneels, Muthu Kumar, Murali Sundar, and John Light. The personal server: Changing the way we think about ubiquitous computing. In *Proceedings of Ubicomp 2002: 4th International Conference on Ubiquitous Computing*, volume 2498 of *LNCS*, pages 194–209, Gotebog, Sweden, September 2003. Springer-Verlag.

[38] Roy Want, Trevor Pering, James Kardach, and Graham Kirby. The personal server: The center of your ubiquitous world. Technical report, 2003.

[39] Charles P. Wright, Michael C. Martino, and Erez Zadok. Ncryptfs: A secure and convenient cryptographic file system. In *Proceedings of USENIX Annual Technical Conference*. in proceedings of USENIX Annual Technical Conference, June 2003.