Towards Persistent, Distributed, and Replicated User Interfaces in the Octopus

Francisco J. Ballesteros, Enrique Soriano, Gorka Guardiola
nemo@lsub.org, esoriano@lsub.org, paurea@lsub.org
10/4/2007

ABSTRACT

In the Octopus each user has a single PC in the network that is used to run applications. Any machine in the Internet can be used as a user's terminal to access the Octopus, which means that the latency between the user interface devices and the central system may be utterly high. Multiple machines are used as terminals, simultaneously. We have developed a unique user interface file system for the Octopus, `o/mero`, along with a user interface viewer, `o/live`. Together, they can transparently deploy and distribute user interfaces, and any of their components, onto one or more terminals connected to the PC by the Internet. Individual UI components can be replicated in a transparent way, as a convenience for the user. The UIMS is actually a file server that maintains the UI elements as files, and is used by independent viewers that interact with the user.

1. Introduction

Following the ideas of Plan 9 [5] and Plan B [1], we are currently implementing the Octopus, a system whose goal is to provide users with a homogeneous and easily extensible distributed computing environment. Octopus is implemented on Inferno [4] and thus can run either on the bare hardware, or hosted on Plan B, Plan 9, Windows, Linux, and other flavors of UNIX. Furthermore, a plug-in for the Firefox web browser is work in progress.

Unlike many other systems, the main idea behind Octopus is to simplify the distribution of the system by *centralizing everything*. To achieve this, each user designates a particular computer in the network as his *personal computer*, referred to as the "PC", and runs on it all software; both system and applications. Other computers are merely used to provide resources (data, special devices or other services) to the PC. Such machines, known as *terminals*, run the Octopus software in order to export all or some of their resources to the PC.

Terminals may provide any device to the PC, but most notably, they provide graphical user interfaces, audio, and voice support. While in the office, an Octopus user combines multiple terminals to gain more screen space. While on the move, a user may employ any machine available in the environment to run an application program (available for all the platforms mentioned above) and convert the machine on a terminal.

Developing a UI service for the Octopus has been challenging, because of several reasons:

1 With round trip times (RTTs) in the Internet of 100 or 200 milliseconds, deploying user interfaces on remote machines is a problem. User interaction must be decoupled from the user interface, perhaps surprisingly, because of latency.

2. Having multiple machines, users want to move components of UIs among machines, freely. This should be done without disturbing applications. Otherwise complexity increases.
3. Because of the multiplicity of machines, for some applications, it is desirable to be able to replicate some of their controls (UIs) on several different machines to have them available at different places. For example, to have the room's music player controls at all machines in the room, or to have a copy of the mail interface at distant machines.
4. As terminals are supposed to be volatile, we want to keep the state and layout of user interfaces in the central computer, so that attaching a new viewer suffices to continue working as it was left off. In fact, a user might switch off a terminal and swich one or more ones later (perhaps at a different location) to continue working.
5. We want to keep applications simple, regarding UI programming, and unaware of these problems.
6. Last but not least, we want the user interface to be available for programmatic inspection and use from outside its application, to be able to write tools to adapt user interfaces to the circumstances derived from the context of the user, like done in Omero [2].

We have redesigned and reimplemented the Plan B user interface service, Omero [2], for the Octopus [3], to address these issues. After discussing some antecedents of our work, the rest of the paper describes it and our experience building it. Before proceeding, we have to say that the current implementation is still work in progress, because we still keep changing the user interface and because we still try to make it less sensible to latency in the network

2. Problems with Omero

Omero [2] is the Plan B window system. In Omero, the window system is a file system that uses a file hierarchy to represent a hierarchy of panels (sharing a screen). An omero application creates a file hierarchy to create a hierarchy of panels or widgets. As an example, figure 1 depicts a menu and the corresponding file tree.

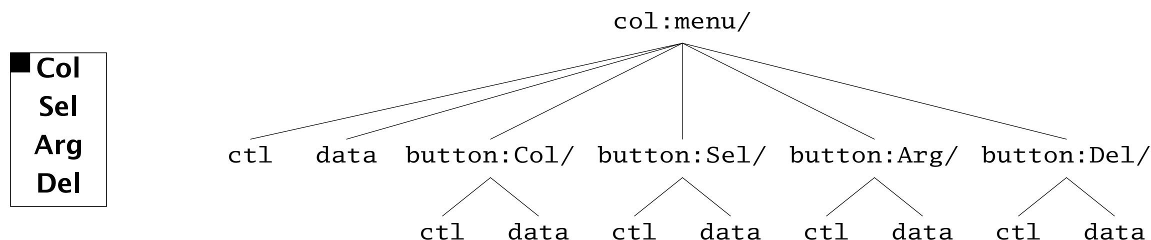

Figure 1: *Files used to represent panels in Omero (and O/mero).*

The application performs I/O on the files to operate on the panels. Events from the user interface to the application are similar to those of Acme and are delivered through network links established from Omero to the applications. Figure 2 shows the processes involved. Omero runs at the terminal machine, close to the screen. Ox (the shell for Omero) and other applications may run at the terminal or at any machine with the Omero file system mounted.

This works fine within a local area network but is not well suited for the Octopus because of the problems we describe next.

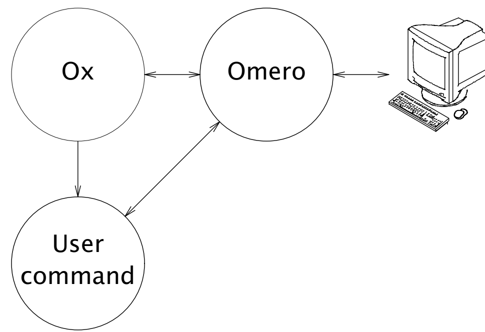

Figure 2: *Processes involved in Plan B's Omero.*

2.1. Latency

Applications perform most of the operations by creating files, removing files, and writing small strings to them just to update the user interface. The result is that bandwidth is not usually a problem unless big images are displayed. Even then, the images are kept within the window system and there is no need to refresh them.

The actual problem is latency. With network round trip times of 100 milliseconds (and above) Omero becomes unbearable to use. Each file operation (performed via 9P) requires multiple RPCs. For example, to create a button panel we might execute:

```
; mkdir /n/omero/row:wins/text:notice
```

Doing this from the shell requires 108 RPCs using Inferno. Using the system call, it requires 4 RPCs. That would be 200ms if done over a connection with 50ms of RTT.

2.2. Network connections

The peer-to-peer nature of Omero makes it necessary for any omero to be able to dial any application. This is necessary to replicate panels. By copying the control file of a panel (to the new copy) the Omero holding the replica would dial the application. From that point the library used by the application would keep the copies updated. In any case, the system holding the applications must still dial and mount each Omero being used.

In the Octopus it is common to have terminals behind NAT boxes, and this makes it unfeasible (in practice) to let anyone dial a terminal. Only the PC is assumed to be able to listen for and to accept network connections.

2.3. Race conditions

There are races among Omeros sharing panels that can be avoided or minimized when latencies are not so bad. In the Octopus, using the technique employed by Omero would either block other terminals for a long time or increase the race time window enough to break the window system (depending on how we address this issue).

This means that it becomes undesirable to handle coherency between the different replicas directly between the terminals (as Omero would do).

2.4. Convenience

When terminals are used to resume sessions, it is convenient to be able to keep the applications running and resume just the sessions. There is no easy way with Omero of doing this.

This problem is more serious in the Octopus than it would be in Plan B, because it

is easier to start an Octopus terminal than it is to start a Plan B one (just run a customized Inferno on whatever system might be at hand). As a result, not being able to resume a session is now considered a burden for the user.

It is also desirable to be able to operate on any text or panel no matter the screen hosting it. In Omero we use shell scripts that operate on the file interface. Now that terminals may be far away, it becomes more desirable to have a command language (similar to Sam), for example, to replicate particular panels at the local terminal (and also perform edit operations).

2.5. Portability

Omero is implemented in C. It would run on Plan B or Plan 9 (provided its fonts are present) but not on any other system. For the Octopus it is not reasonable to ask a user to boot a native system just to access the central PC.

3. O/mero and O/live

Trying to address the problems mentioned above we built a new window system for the Octopus that is implemented by two programs: O/mero and O/live. The former maintains the state for the window system as a file system (and handles panel replication); the latter is a viewer that maps the file system to a graphical user interface on a particular terminal.

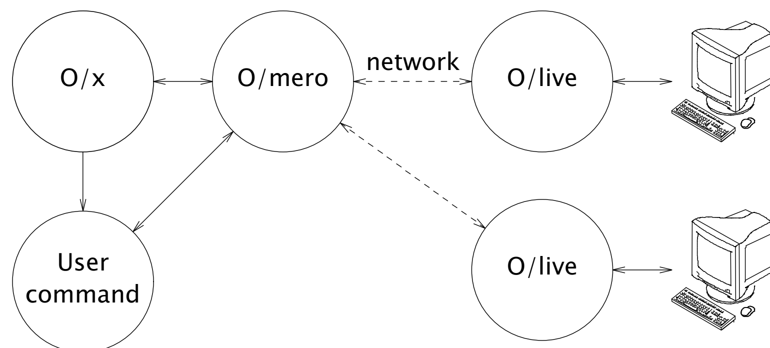

Figure 3: *Structure of o/mero and o/live.*

Figure 3 shows the resulting design. O/x (the new shell) and other applications create their interfaces by using the file system provided by O/mero. Unlike before, O/mero does not draw and does not process mouse or keyboard input. O/live, a new program, builds a graphical user interface corresponding to the file system kept by O/mero and updates it according to user I/O.

Applications can be now kept close to O/mero, within the single PC, and there is no problem regarding latency or bandwidth as far as a the applications are concerned. The viewer, O/live, is decoupled from the application even more than it was in Plan B. Several benefits arise: (1) the viewer can be changed without modifying omero; (2) the viewer can be shut down and restarted without the application noticing; and (3) it is feasible to implement different viewers with a different look and feel each.

O/mero provides a file tree for the applications and distinct, separate, file trees for the viewers. Applications create panels at `/appl` (relative to the O/mero root). The paths for the panels in the `/appl` tree remain fixed for their entire life, which simplifies locating a panel. Note that in Plan B's Omero the path for a panel was modified as the panel location in the screen changed.

Directories created at the root of the O/mero tree, other than `/appl`, represent screens. The viewer, O/live, receives as an argument the path to the screen to be viewed

in the screen. Creation of a screen directory automatically creates an initial panel hierarchy with several rows and columns, as a convenient initial layout for an empty screen.

A panel created at `/appl` is not shown anywhere (unless it is created within a panel already shown). To show a panel at a particular screen, a replica of the panel must be established at the screen (and particular location) of interest. From that point in time, the directory representing the panel can be seen both under `/appl` and at the location of the replica in the file tree for the screen. If the panel is moved in the screen, the location of the replicated file would change to reflect the new layout. Of course, to view a panel in two or more different screens, two or more replicas of the panel may be created. Moving a panel from one location to another is also feasible but only replicas may be moved (not so the original panel).

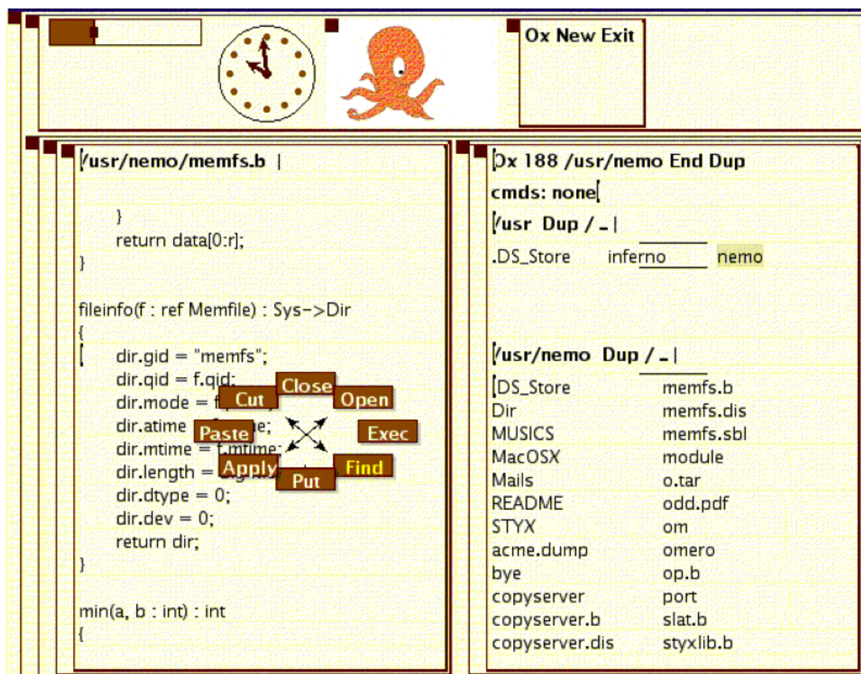

Figure 4: *A terminal running O/live from a mounted O/mero.*

As an example, the following commands create a text panel, copy `/NOTICE` into it, create a screen, replicate the panel at the new screen, and move it to the second column of the screen.

```
% cd /mnt/ui
% mkdir appl/text:xample
% cp /NOTICE appl/text:xample/data
% mkdir screen
% echo copyto /screen/row:wins/col:1 > appl/text:xample/ctl
% echo moveto /screen/row:wins/col:2 > screen/row:wins/col:1
```

Note how the move operation is performed on the replica to be moved, and not on the original file at `/appl`. Being all the involved files under the control of a single program, a move or a copy operation is performed atomically by a single control operation, without affecting the application.

To decouple the viewer from the file system, most edition is meant to happen locally in the viewer. Nevertheless, O/live tries hard to keep the file system up to date to let the user kill the viewer at any instant.

O/live shows a screen layout corresponding to the file tree it is viewing (like Omero did). Figure 4 shows an example screen. To retrieve the contents of a panel the viewer may read its `data` file. That file may be updated later by the viewer to reflect user changes. Moving panels and copying them with the mouse results in control operations issued to the file system, like the ones in the example above. Once O/mero rearranges the file tree in response to such control requests, the viewer is notified of the changes and it updates accordingly.

If the application updates the contents of a panel O/mero would notify all the viewers involved, to let them update their replicas. In the same way, when a viewer updates a panel other replicas are notified.

The contents of the control file for a panel (like in Omero) contain panel attributes. Updating the control file for the panel at `/appl` would update all the control files in the replicas. However, updates for the control file of a replica may or may not propagate to others. For example, writing `hide` to a control file hides a panel. Writing this under `/appl` makes all the replicas hide. Writing this under `/screen` would hide the replica responsible for the control file used. Other control operations (and the corresponding attributes) are global; e.g., `font`.

3.1. Events

A separate program (not mentioned before) provides event delivery to O/mero, O/live, and the application. This program permits clients to create files representing event reception ports, and to write on them a regular expression to select the events of interest. An event is posted by writing its text to the `send` file provided by this program.

O/mero notifies the application of abstract events related to its panels. Typical events are *look*, *exec*, *close*, and *dirty* (look for something in the file system, execute a program, the last replica for a panel is gone, and the user has edited the panel contents). Table 1 shows the complete set of events delivered to the application along with their meaning.

Event	Meaning
Look	look for something in the file system
Exec	execute a command
Apply	apply a command to the selection
Close	last replica for the panel is closed
Click	mouse event (only sent when requested)
Keys	keyboard event (only sent when requested)
Interrupt	user wants to interrupt the application
Clean	the panel is clean (no edits by the user)
Dirty	the panel is dirty (user made edits)

Table 1: *Events sent from O/mero to the application.*

O/mero also posts events reporting changes in the file trees for the viewers, as shown in table 2.

All events have the same format, no matter their purpose or destination, as shown in this example:

```
o/mero: /appl/xample:text 4 exec pwd
```

The event includes the name of the program posting it, the path for the panel (replica), a panel identifier (as set by the application), the event type, and any optional argument for that event. Viewers may register to receive events for panels with paths that have as a prefix the path of the screen viewed. Applications, on the other hand, register to receive

Event	Meaning
Update	update the view for a subtree
Top	view the subtree rooted here
Insert	text was inserted in a text panel
Delete	Text was deleted from a text panel

Table 2: *Events sent from O/mero to O/live.*

events for the panels they created under the `/appl` tree.

An important event sent from O/mero to O/live is *update*. It carries as an argument the path for the subtree that has changed. Changes to multiple files may be reported by a single event referring to a common ancestor in the file hierarchy. Upon receiving an *update* a viewer should check the file system for changes and update accordingly.

Text panels are interesting in that viewers deliver individual *insert* and *delete* events via control operations on the panel (instead of updating the entire file). O/mero reacts by posting *insert* and *delete* events to other replicas involved (to save them the burden of re-reading the panels). This also helps to keep the file system synchronized with the viewer.

O/live synchronizes its state with the file system whenever there is a mouse movement, and also before writing *exec* or *look* control requests (which make O/mero post the corresponding events to the application involved). The selections on the panels are kept as attributes in their control files, and the most recently used panel is named in the file `/dev/sel`, available for all programs to see. This is important for what we describe next.

3.2. Command language

For O/live we wanted to be able to use the Sam command language. We have adapted the code from Inferno's Acme to our purpose. Before describing our changes, we have to say that this piece of the implementation is still crashing, and can not really be used yet.

To let the edition language interact heavily with the text being edited, we placed the implementation inside O/x, que shell for O/mero running close to it. Since O/live synchronizes with the file system before triggering the execution of any command, O/x has all the needed information in the O/mero file system.

An interesting change to the command language is the introduction of new commands to select panels at any screen, delete them, copy them, and move them. With such changes the user may write commands to move certain panels to the viewer being used, or to do similar tasks.

4. Implementation

By the time of writing this, the window system described here can be used but not yet in production. The editor command language is not fully debugged and tested and others parts of the system (although debugged, tested, and being used) are not exercised enough to be reliable. That is to say that this is work in progress.

The implementation for O/mero is 2787 lines of Limbo code, where 706 lines are used to implement particular panel types. The program maintains an array of panels, each of which keeps an array of replicas. The replica number zero corresponds to that under `/appl` and others refer to actual replicas. The panel keeps the data for the panel as an array of bytes (without any structure). Replicas keep the data for the panel attributes, so that each replica may have different attributes as explained above.

Attributes are kept as a list of name/value pairs.

In general, O/mero does not interpret the data it keeps. However, it is important to be able to report errors to the user as a result of bad control requests, attempts to create panels of unknown types, etc. O/mero does not implement any panel on its own but loads panel implementations early when starting. All modules in files matching `/dis/o/omp*.dis` are considered to implement one or more particular panel types. An array of panel implementations is maintained to interrogate each one for known type names and for error checking functions for both data and attributes.

Apart from this, and what has been said in previous sections, O/mero works similar to Omero [2].

O/live is more complex because of the details of layout handling and its attempts to mask latency problems. It has 6730 lines (also in Limbo), where 3725 lines are there to implement particular panel types. Similar to O/mero, O/live does not implement any panel on its own. It would load all modules in files matching `/dis/o/owp*.dis`, which know how to handle the mouse, the keyboard, how to draw, how to process data and ctl updates, and how to handle events for particular panel types.

To try to mask latency problems, O/live spawns multiple processes to read file trees concurrently upon receiving *update* events posted from O/mero. The design is depicted in figure 5.

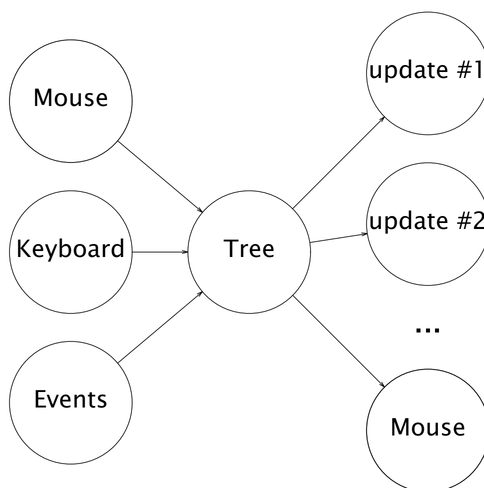

Figure 5: *Processes in O/live*

Like in Rio or Acme, there are processes attending the mouse, keyboard, and events from the underlying system (including those of O/mero in this case). All of these processes determine which operations should be done on the tree of panels and forward requests to a central tree process to execute the appropriate action. For those requests that can be performed immediately, the tree performs them by itself and replies to the caller. Those that either take time are delegated to helper processes (cached using a scheme similar to Acme's Xfids [6]).

Locking between conflicting actions (like updating a subtree and handling mouse commands on its panels) is handled by the tree itself. While an operation is in progress (delegated to a helper process) the *path* for the top-level panel involved in the operation is kept in a locked paths lists. Operations involving paths that are a prefix of a locked path, or where a locked path is a prefix of the operation path, are held. But for the tree, the rest of the implementation can be kept unaware of locking.

Regarding writes to O/mero, O/live updates the tree asynchronously whenever feasible trying to block only when necessary. A subtle issue is when to synchronize to

O/mero those changes resulting from user edits. Synchronizing seldom would result in editions being lost if the viewer crashes or is killed by the user. Doing it too often would result in unacceptable delays for the user. The current implementation collects edits between mouse movements and combines them. When the mouse moves, a single write to a panel control file reports the insertion and deletion of text to O/mero.

5. Application interface

We have implemented a convenience library for using O/mero from Limbo programs. It is described in *panels*(2) in the Octopus manual. The library provides a `Panel` data type to represent a panel. It keeps the path for the panel in the O/mero file tree. The library is intended mostly to aid in the parsing of events. Otherwise, users operate directly on the files provided by O/mero to operate on the panels.

As an example, the following code initializes and creates a text panel filled with the text in `/NOTICE`.

```
panels->init();
ui := Panel.init("xample");
text := ui.new("text:xample", 1);
sfd := open("/NOTICE", OREAD);
dfd := open(text.path+"/data", OWRITE|OTRUNC);
panels->copy(dfd, sfd);
```

Both `Panel.init` and Panel.new use *create*(2) to create the panels. The former uses the program name and process pid to create a unique directory under the `/appl` file tree. The latter creates a panel within the one used to make the call. The `copy` function is a helper used to copy entire file contents from one place to another (which is common when using O/mero).

To show the previous panel in the first column of the first screen the application may execute this:

```
scr := hd panels->screens();
col := hd panels->cols(scr);
text.ctl(sprint("copyto %s0, col);
```

The functions `screens` and `cols` return a list of screen names and column names and are implemented by reading the respective directories.

Event processing is also simple. The library provides a conventional channel based interface for receiving events, as can be seen in the example below.

```
c := ui.eventc();
for(;;){
        ev := <-evc;
        if (ev == nil)
                break;
        print("path %s id %s ev %s0, hd ev, hd tl ev, hd tl tl ev);
}
```

As an aid for the application, O/mero permits setting a per-panel attribute holding an integer panel identifier. Also, it permits setting a per-panel attribute keeping the PID of the process associated to a particular panel. Both attributes are inherited when not explicitly set. The panel identifier is handy to quickly locate the panel for a particular event. The process identifier permits O/mero to notify the application of interrupt requests from the user.

6. Lessons learned and future work

It was very hard to use *Libframe* right. This is the Plan 9 library ported to Inferno for Acme. In the end, we had to write our own mostly because we were incapable of using it right. Our current implementation redraws more than strictly necessary, but is easy to use and easy to debug.

At some point we considered implementing a command language to define compound widgets. This was future work planned for Omero, with the aim of defining simpler interfaces requiring less communication between the application and the file system. The new architecture (with O/mero close to the application) makes this unnecessary. The programming language is indeed used as the command language used to build compound widgets. The file system interface provides for convenient parsing of UI elements and data, supplying most of the user needs.

Regarding latency, we have tried to use O/mero and O/live in several different scenarios that we describe next:

- Mounting O/mero using 9P, with a latency of 100ms. In this case, delays are so big that it is not feasible in practice to use O/live.
- Mounting O/mero using Op, with a latency of 100ms. In this case, O/live feels quite slow but it can be used. This means that there can be a delay of one or two seconds between a user request to edit a new file and the moment when the new layout for O/live (with the new file) has been loaded.
- Mounting O/mero using Op, with a latency of 50ms. In this case, O/live is still slow, but not much more than in the next case.
- Mounting O/mero within the local machine. In this case O/live feels slower than, for example, Acme. This is probably a result of the inefficiency of our frame library and also a consequence splitting the service among different programs (O/mero, O/live, O/x, and O/ports).

In the future we will profile and modify this set of programs to make them faster. One particular optimization we are considering is bundling the changes for the O/mero file tree along with *update* events sent to O/live. This avoids the need to issue multiple RPCs to read the file tree and bring the viewer up to date.

References

1. F. J. Ballesteros, E. Soriano, K. L. Algara and G. Guardiola, Plan B: An Operating System for Ubiquitous Computing Environments, *IEEE PerCom. Also http://lsub.org*, 2006.
2. F. J. Ballesteros, G. Guardiola, K. L. Algara and E. Soriano, Omero: Ubiquitous User Interfaces in the Plan B Operating System, *IEEE PerCom*, 2006.
3. F. J. Ballesteros, P. Heras, E. Soriano and S. Lalis, The Octopus: Towards building distributed smart spaces by centralizing everything., *UCAMI*, 2007.
4. S. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey and P. Winterbottom, The Inferno Operating System, *Bell Labs Technical Journal 2*, 1 (1997), .
5. R. Pike, D. Presotto, K. Thompson and H. Trickey, Plan 9 from Bell Labs, *EUUG Newsletter 10*, 3 (Autumn 1990), 2–11.
6. R. Pike, Acme: A User Interface for Programmers, *Proceedings for the Winter USENIX Conference*, 1994, 223–234. San Francisco, CA..