Proceedings of the 7th International Workshop on Plan 9

November 14 – 16, 2012

Bell Labs Ireland
Alcatel-Lucent
Blanchardstown Business and Techonology Park
Snugborough Road
Dublin 15
Republic of Ireland

Editor Eric Jul

Organization

Eric Jul, *Bell Labs* (local organizer)

Franck Franck, *Bell Labs* (registration, logistics)

Erik Quanstrom, *IWP9.org* (web site)

Program Committee

Eric Jul, *Bell Labs* (co-chair)

Francisco Ballasteros, *Lsub, URJC* (co-chair)

Sape Mullender, *Bell Labs*

Participants

Balaji Srinivasa	balaji.srinivasa@gmail.com	
Eric Jul	eric.jul@cs.bell-labs.com	Bell Labs Ireland
Jeff Sickel	jas@corpus-callosum.com	Corpus Callosum Corporation
Jonas Amoson	jonas.amoson@hv.se	University West, Trollhättan, Sweden
Thomas Lundqvist	thomas.lundqvist@hv.se	University West, Trollhättan, Sweden
Erik Quanstrom	quanstro@quanstro.net	
Richard Miller	miller@hamnavoe.com	Miller Research Ltd
Franck Franck	franck.franck@alcatel-lucent.com	Bell Labs Ireland
Aram Sadogidis	aram@privatdemail.net	University of Thessaly
Jesús Galán (yiyus)	yiyu.jgl@gmail.com	UGent
Sape Mullender	sape@plan9.bell-labs.com	Bell Labs Belgium
Jan Sacha	jan.sacha@alcatel-lucent.com	Bell Labs Belgium
Dirk Hasselbalch	dirk.hasselbalch@alcatel-lucent.com	Bell Labs Ireland
Tommaso Cucinotta	tommaso.cucinotta@alcatel-lucent.com	Bell Labs Ireland
Nilo Redini	nilo.redini@alcatel-lucent.com	Bell Labs Ireland
Gorka Guardiola Múzquiz	paurea@lsub.org	Universidad Rey Juan Carlos
Andrés Domínguez	andresdju@gmail.com	

Foreword

These are the proceedings of the Seventh International Workshop on Plan 9, IWP9. It took place November 14th – 16th, 2012 at Bell Labs Ireland, at the premises of Alcatel-Lucent in Blanchardstown Business and Technology Park, Dublin 15, Ireland.

The workshop includes a keynote by Sape Mullender on future trends.

The workshop was organized by Bell Labs Ireland, which much deserve thanks for providing support, lunch, coffee, meeting rooms, etc. I would also like to thank Erik Quanstrom for providing support of the iwp9.org web site.

The workshop includes a social program, which, in the spirit of Ireland, includes visits to classic Irish pubs and classic Irish pub grub. Attendance is down from last year, so some thought is required concerning future workshops.

The Tentative Workshop program is included below. The program, the proceedings and other information concerning the workshop can be found at http://iwp9.org.

On behalf of the organizers,

Eric Jul

Table of Contents

Add some Olives to your coffee: A Java-based GUI for the Octopus system 1
Aram Sadogidis, Spyros Lalis

A light-weight non-hierarchical file system navigation extension 11
Jonas Amoson, Thomas Lundqvist

Networking in Osprey 14
Jan Sacha, Sape Mullender

ΠP, a new way to use the Internet 22
Sape Mullender, Jeff Napper, Francisco Ballasteros

A Performance Comparison of Cryptographic Hashes and Ciphers under Plan 9 and Linux 29
Franck Franck

A NIX Terminal 37
Erik Quanstrom

Access Control for the Pepys Internet-wide File-System 42
Tommaso Cucinotta, Nilo Redini

Atomic increments 56
Enrique Soriano-Salvador, Gorka Guardiola Muzquiz

Planned Program

Overall Program Schedule

Nov 14th:

14:00 Registration (and coffee) in Bell Labs Museum Area

16:00 Welcome + keynote by Sape Mullender, in Shannon room

17:30 Prearranged, shared taxi departs for Maldron Hotel/Generator Hostel

18:45 Start walking from Maldron/Generator toward Mulligans, it is 420 meters and should take about 5 minutes

19:00 Intro to Irish pub + gastropub dinner

L Mulligans Grocer, 18 Stoneybatter, Dublin 7

www.lmulligangrocer.com

21:45 Start walking back from Mulligans to Cobblestone, it is about 400 meters and should take about 5 minutes, well perhaps 10-15 minutes, if you discuss more than a few of the 18 beers on tap at Mulligans, while walking to the Cobblestone.

22:00 Live Irish Session music at Cobblestone pub (across from Maldron Hotel)

Cobblestone Pub, 77 North King Street, Dublin 7

23:30 Last call at Cobblestone - music usually continues for at least another hour.

Late: Walk home to Maldron, distance is about 70 meters, and should take less than 20 minutes (depending on how tired you are, some do it in less than one minute...)

Nov 15th

08:45 Prearranged, shared taxi departs from Maldron Hotel.

09:30 talk: Add some Olives to your coffee: A Java-based GUI for the Octopus System, Aram Sadogidis, Spyros Lalis.

10:15 talk: A light-weigh non-hierarchical file system navigation extension, Jonas Amoson, Thomas Lundqvist.

11:00 coffee break

11:15 talk: Networking in Osprey, Jan Sacha, Sape Mullender.

12:00 Prearranged, shared taxi to Lunch at nearby Market (bring coat in case of rain). Buy lunch (bring cash). Prearranged, shared taxi back to Bell Labs.

13:45 demo: Richard Miller, Plan 9 on the Raspberry Pi

14:00 Piepea - a new way to use the Internet, Sape Mullender, Jeff Napper, Francisco Ballasteros.

15:00 coffee break

15:20 talk: A Performance Comparison of Cryptographic Hashes and Ciphers under Plan 9 and Linux, Franck Franck.

16:05 talk: A NIX Terminal, Erik Quanstrom.

17:00 Prearranged, shared taxi to Hotel Maldron/Generator Hostel/Porterhouse, drop you stuff at the hotel and continue immediately to Porterhouse, OR take your time and walk/take a taxi to Porterhouse

17:40 Start walking from Maldron to Porterhouse it is about 1,200 meters and should take about 15 minutes. Directions: exiting the hotel turn right and go down the plaza until you hit the river. Turn left and follow the river. At the THIRD bridge (Capel Street) turn right, cross the river on the bridge. The road turns into Parliament Street, the Porterhouse is on the left, right after you have crossed the bridge. Arrive anytime between 18 and 19.

18:00 Microbrew at Porterhouse Temple Bar, 16-18 Parliament Street, Dublin 2

19:00 Dinner at Porterhouse, www.porterhousebrewco.com, bring cash.

22:00 Those that want to explore the Temple Bar area of Dublin can merrily walk through it--the Porterhouse is at the start of the area, just turn left as you exit the Porterhouse and walk east.

Late: Return by walking or by one of the many taxis that are everywhere. Expect to pay between 6 and 9 Euros for a ride back from the Temple Bar area to the Maldron Hotel. Or just walk, it is about 1.5 - 2 km walk.

Nov 16th

9:30 talk: Access Control for the Pepys Internet-Wide File-System, Tommaso Cucinotta, Nilo Redini.

10:15 demo: Richard Miller, Fantasy on an FPGA

10:30 coffee break

10:45 talk: Atomic increments, Enrique Soriano-Salvador, Gorka Guardiola Muzquiz.

11:30 WIP talk: Dirk Hasselbalch

11:45 concluding remarks

12:00 lunch at Bell Labs

Add some Olives to your coffee: A Java-based GUI for the Octopus system

Aram Sadogidis
Spyros Lalis

University of Thessaly
Volos, Greece

ABSTRACT

In this paper we describe our efforts to enable Java supported terminals to interact with the Octopus pervasive environment. In order to achieve that goal, we developed a Java-based implementation of the Octopus GUI front-end.

1 Introduction

Pretty soon many people will own a large number of smart devices with Internet capability through the wire or over the air. While having many gadgets can be great fun, the user will also be faced with the burden of managing a highly decentralized, uncoordinated, heterogeneous and dynamic device ecosystem. Currently, the problem is typically "solved" by letting the user act as a mediator between these devices. This approach, apart from not being much fun, cannot possibly scale to a large number of devices. As another, relatively recent option, data and applications can be placed in the cloud. However, the flexible exploitation of the hardware and software resources of other devices remains a challenge.

The Octopus system [1, 2] tackles this problem by applying the principle of centralization (as the cloud paradigm) in conjunction with an open, simple and flexible resource sharing architecture. All applications run in a single computer, called ***the computer***, while every other smart device connects to the computer over the network to provide special resources or act as interactive terminals for these applications.

Like Octopus itself, the standard GUI terminal support for the Octopus is implemented on top of Inferno [3]. Even though Inferno has been ported on many platforms (e.g., Linux, Windows, MacOSx, Plan9, Solaris and BSD), in practice only a few people will go through the installation process merely so that they can run an Octopus terminal. Moreover, for all practical purposes, one cannot expect that Inferno will or should be ported on all platforms. However, the Java runtime environment is being ported, aggressively, on all kinds of platforms and enjoys strong support from a large part of the industrial world. In particular, a significant share of the booming smartphone market goes to Android, which comes with native Java support.

With this rationale, we think it is a good idea to extend the arms of the Octopus so that it can reach out for Java terminals. Aiming for that goal, we developed a Java-based implementation of the Octopus GUI front-end, named JOlive (after Olive), through which one can interact with all the applications running on an Octopus computer (which can run anywhere in the net/cloud). In addition, we ported JOlive on Android thereby enabling a smartphone to act as yet another terminal device for the Octopus, while keeping the native look & feel. Besides dealing with issues specific to the Android environment, the smartphone version addresses the problem of limited screen real-estate, by allowing the user to "pull" the UI of just a few or perhaps even a single application in order to enjoy a more focused and efficient interaction.

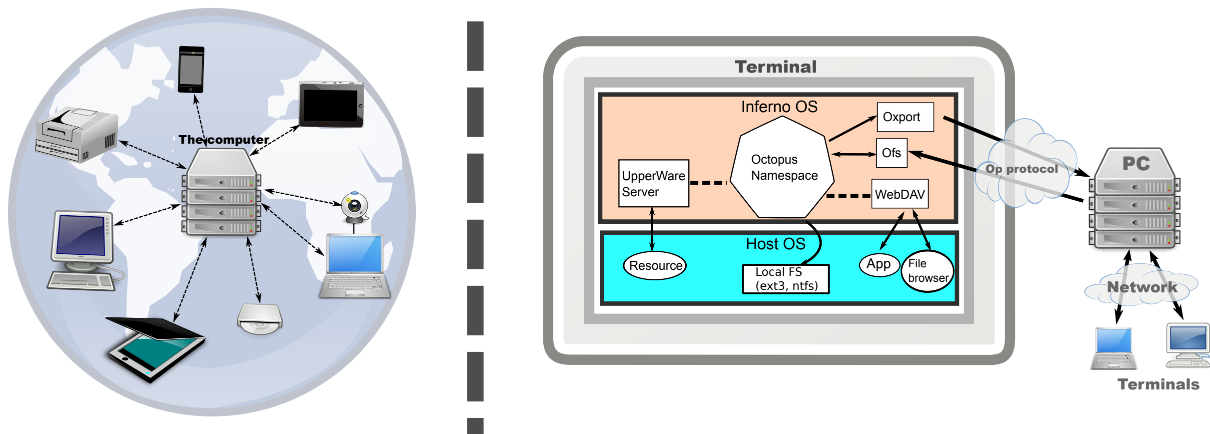


Figure 1: Various terminal resources connect to the central PC and export resources.

In the following we give a brief overview of Octopus and the Octopus UI system. Then, we proceed to describe our Java-based implementation and the respective Android port. We close the paper with some reflections on future work.

2 The Octopus in brief

The Octopus system [1, 2] aims to provide a single and homogeneous (for the application programmer) yet heterogeneous and ubiquitous (for the user) multi-device computing environment. Rather than pretending that all devices are equals, and trying to support a peer-to-peer interaction between them, Octopus distinguishes a single node, called ***the computer***, as the center of the personal computing universe, where everything else is connected to. Figure 1 shows an example. The connections between all other devices and the computer are implemented using the Styx [4] and Op [5, 6] protocols (the latter being designed for long-latency links), which support the abstraction and access of resources in the form of synthetic file systems.

All applications run on the computer. Hence the user is relieved from the frenzy of downloading, installing and managing different applications (including their state and data) on different devices. Moreover, the user can attach, at any point in time, various devices to the computer, in which case their resources automatically become visible to the applications running there. While switching-off devices or bad connectivity will lead to the loss of these resources, such mishaps do not lead to nasty failures or a corruption of crucial application runtime state.

Octopus is built on top of Inferno [3] which in turn runs directly on bare metal or hosted on popular operating systems. Indeed, this large host base is exploited to implement a so-called UpperWare approach [7] for connecting legacy devices to the Octopus computer. More concretely, an Inferno-based software layer, placed on top of the native OS, abstracts and exports various hardware and software resources of the device in the form of a synthetic file system that speaks Styx or Op; of course, local access of these resources occurs though the host OS. This concept is illustrated in Figure 1. It is worthwhile noting that legacy applications can also be wrapped as resources (provided they can be accessed through the host OS interface), making it possible for them to be exploited from within Octopus applications and, conversely, applications (or the user) can exploit Octopus resources through a file system interface.

One example of an UpperWare device exploiting a host application resource is the browserfs [8, 9]. Its purpose is to control the web browser of a terminal in order to replicate its state (bookmarks, history, etc) to other terminals. The UpperWare driver, with the support of host command scripts, stores in regular disk files the state of the browser. These state files are accessible through the Octopus namespace by other terminals, who re-export them to their

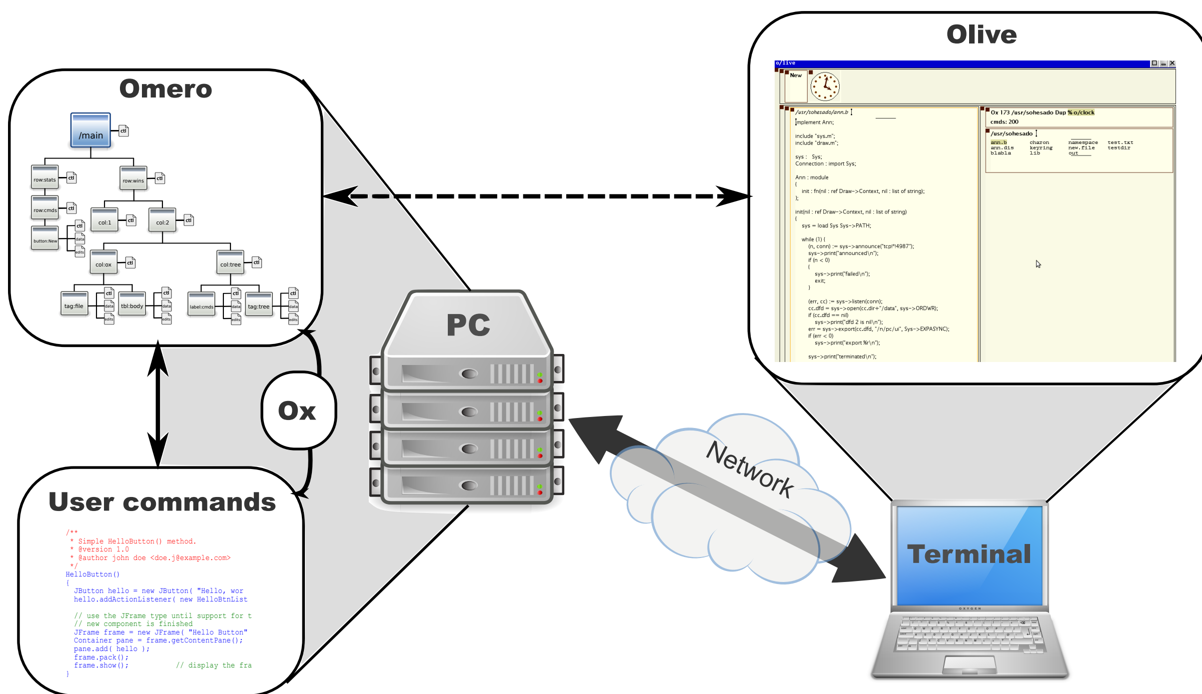


Figure 2: Example of a tree served by Omero and used by Olive on an Octopus terminal.

host systems over WebDAV.

Octopus employs the file-system abstraction for yet another purpose, namely to implement its window system, called Omero [10]. Omero serves a file tree with separate directories representing different virtual screens and a hierarchy of graphical components, known as ***panels***, each being represented in turn by a directory with a control file, and other optional files depending on the type of the panel. A typical Omero file hierarchy is shown in the left part of Figure 2.

Notably, the architecture of Omero enables a number of interesting functionalities, such as: (i) ***migrating*** any part of an application's UI to any terminal simply by moving the corresponding filesystem hierarchy branch to the directory for the screen assigned to this device; (ii) ***replicating*** parts of the UI on different terminals by copying panel directories to multiple screens; (iii) ***reusing*** powerful tools, e.g., ***tar*** to save and restore UIs, or ***find*** to locate windows with a certain title; and last but not least (iv) offering a ***familiar and easy-to-use API*** for writing GUI applications without bothering the programmer with the actual physical placement of the UI on one or more screens.

Omero does not draw and does not interact with the user. This functionality is delegated to the viewer component of the window system, called Olive [10] which runs on terminal devices. Olive is the only program that knows how to draw, how to interact with the mouse and the keyboard, and how to graphically manifest different types of panels. It initializes the graphical window based on the virtual screen to which it is attached to. From that point onwards, it maps the local keyboard and mouse events to the corresponding file operations on Omero's panels, and receives notifications about any UI updates through the Omero event channel file.

Finally, Octopus comes with its own shell, called Ox [11], which can be used to browse the file system, edit, and execute commands. The command language of Ox consists of some built-in commands, Sam commands, host OS commands and Inferno commands. The user can type and execute these commands explicitly, or through the tag and panel operations of the window system viewer.

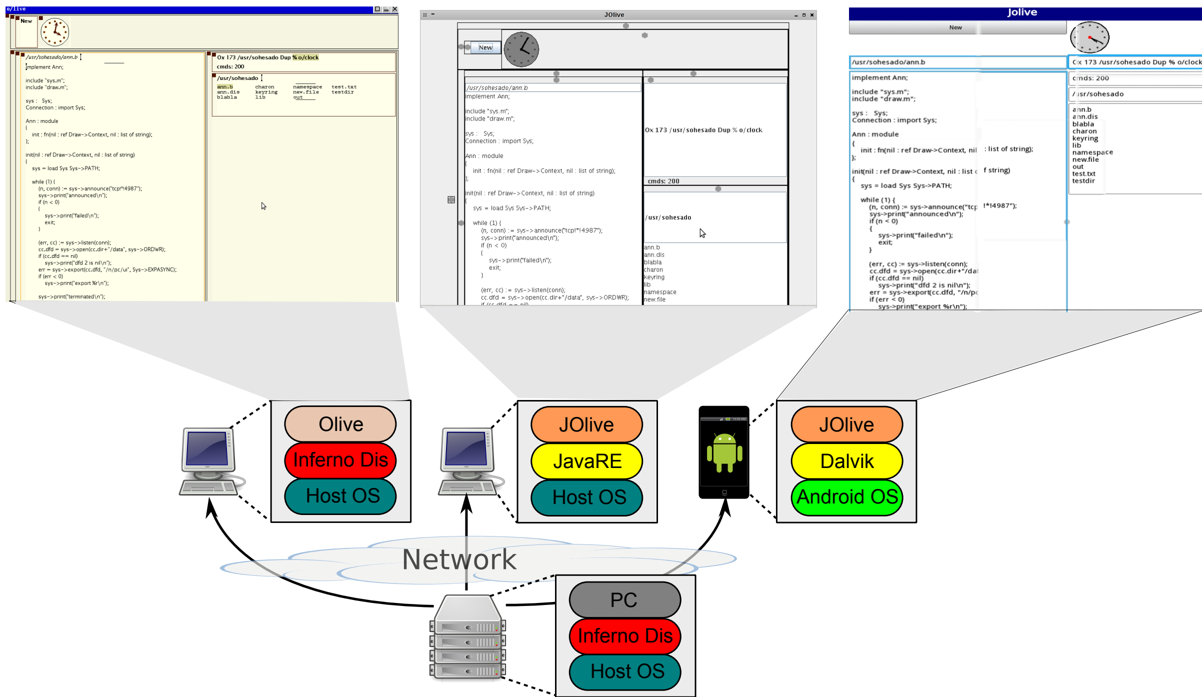


Figure 3: The three terminal variants with their respective UI screenshots.

3 The Java Octopus Terminal

Arguably, the Octopus system's effectiveness increases proportionally to the number of resources integrated to the global namespace. In order to broaden the spectrum of potential resources, we have implemented a Java-based GUI for the Octopus system. By doing so, we enabled Java-supported devices to export display resources (i.e. screen space) to the Octopus environment. The current implementation can be extended in order to expose other resources too (Section 4).

There are quite a few advantages of a Java-based approach compared to the original Inferno-based. A Java developer has the luxury to choose from many GUI toolkit options (AWT, SWT, Swing, Apache Pivot, JavaFX, Qt Jambi) whereas an Inferno developer's options are rather limited. One other technical advantage is the easier installation of Java virtual machine and its applications. Also there are available additional JavaVM-based languages to choose from, like Scala, Jython, Groovy, etc.

It should be noted that by including the Java platform as an alternative underlying software technology, we aim to expand the Octopus development opportunities, not to replace the Inferno-based terminal. Having said that, the number of mobile platforms supporting Java is an "audience" that should not be easily neglected.

JOlive is an Omero tree ***viewer***, implemented in Java. The first version we've developed is based on Java Standard Edition version 6 (JavaRE-1.6 virtual machine) and the Swing graphical toolkit. The second version is focused towards the Android 2.2 platform and therefore it is based on the Dalvik virtual machine and the Android graphical toolkit. Both versions use JStyx[1] to communicate with Omero. The Figure 3 provides an overview of the two implementations compared to ***Olive***.

[1]JStyx's port to Android required some tweaking (Section3.2.3).

OPanel	An object of this class corresponds to a directory that represents a panel. It contains the functionality to invoke ***ctl*** commands, retrieve ***data*** and panel related ***attributes.***
OOlive	This class represents the ***olive*** file served by Omero. It runs on a dedicated thread and it is responsible to receive update messages, demultiplex them and pass them to the corresponding JOPanel object.
Merop MeropCtl MeropUpdate	These three classes encapsulate the information related to the event messages generated by Omero. MeropCtl and MeropUpdate are derived from Merop, effectively unpacking/packing the messages to a ***ctl*** or ***update*** type.
OAttrs OUtils	OAttrs encapsulates the attributes that a panel may have and OUtils contains debugging functions.

Table 1: This table presents the omero package class description for the desktop JOlive.

3.1 JOlive for the desktop

JOlive maintains a bidirectional communication with Omero that implements the action-effect feedback loop. There is the ***user action notification*** which occurs whenever the user interacts with the GUI and the ***UI update notification*** whenever a change occurs to the Omero synthetic file system. The first data flow updates the Omero tree based on the user's actions, and the second updates the terminal's GUI based on events generated by the window system.

The graphical components presented by JOlive, have the corresponding ***action event*** listeners which invoke the corresponding remote commands. So, in an event-oriented fashion the Omero tree is updated as soon as an event is triggered by an action (e.g. a button click).

The ***update notification*** data flow path is more complex. When the filesystem is updated, either by an application or a viewer, Omero generates event messages which mirror the changes, encoded according to the merop protocol[2]. A dedicated component is required to receive the update messages and demultiplex them. The received data are packed ***merop*** messages that are unpacked into ***Merop*** java objects.

The implementation is composed of two packages. The ***omero*** package which implements the synthetic file system communication part and the ***ui*** package that implements the visual part of JOlive. The first is described in table 1 and the second in table 2 and the relationship between the classes in Figure 4.

3.2 JOlive for Android

In order to port JOlive to the Android platform we had to port the JStyx library first. After some hacking, we ported it successfully and consecutively we reused, almost intact, the ***omero*** package which is the JStyx dependent part of the implementation. On the other hand, we had to reimplement the ***ui*** package from scratch since the Swing API is not available on the Android platform. A side benefit of the reimplementation was that the resulting GUI obtained the native Android look & feel.

The Android version follows the same design with the Desktop version. The **additional** classes that have been implemented, are listed table 3.

[2]Omero defines a data packing/unpacking private protocol in order to transmit messages to Olive.

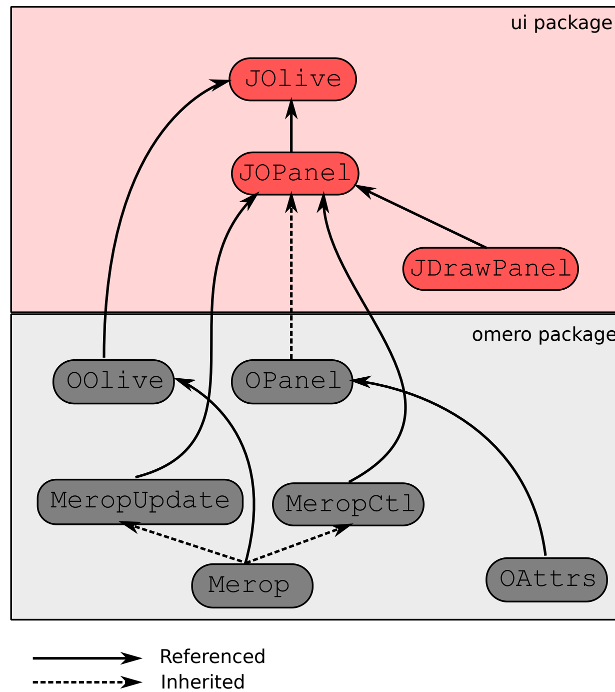


Figure 4: The class dependency of the implementation.

JOPanel	This class is a direct descendant of OPanel and has a reference to the graphical component corresponding to the Panel. Every object of this class is registered to the OOlive's hashtable in order to be notified if a related update message arrives. In essence this class combines the JStyx related stuff with graphical part of the implementation.
JDrawPanel	This is a customized JPanel component, used to represent the Omero's draw panel.
JOlive	The main class that initializes the window and initiates a connection with the Octopus PC.

Table 2: This table presents the ui package class description for the desktop JOlive.

OOliveEventHandler	This is the only additional class in the omero package used to tackle a technical difficulty presented by the Android runtime (Section 3.2.3).
ActionSwipeListener TextLongClickListener	These two classes implement the command invocation swipe widget (Section 3.2.1)
PullAppListener	This class implements the Pull App functionality (Section 3.2.2)
ConfigureLog4j	This is a hack employed while porting JStyx to Android (Section 3.2.3)

Table 3: This table presents the ui package class description for the desktop JOlive.

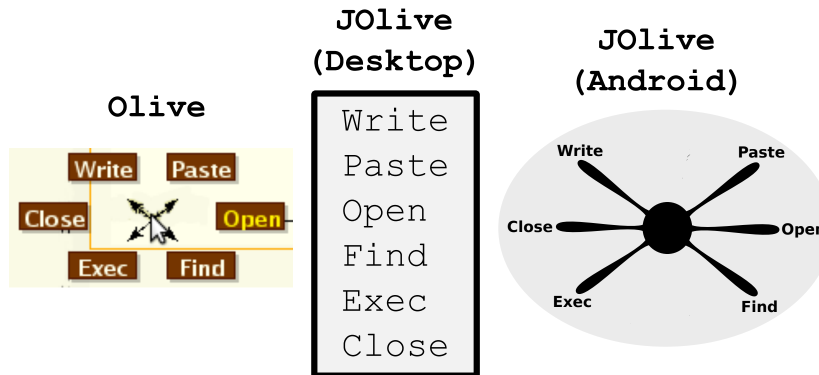


Figure 5: The three variants of menu graphical components.

3.2.1 Command invocation swipe widget

The user may click on a margin or tag and invoke commands over the Octopus panels. Although for the desktop version we've followed the traditional right-click drop down menu, for Android this approach was dismissed as it was rather impractical for touchscreen devices. Instead we adopted the exact same solution employed in Olive, namely the popup menu that shows different options in a circle around the point (refer to Figure 5). To select one you must move the pointer quickly in the direction of the option.

For the Android implementation, in order to invoke a command the user swipes his finger from the center to the direction of the desired command. In essence, the mouse gesture used in Olive, is replaced with a swipe action on the touchscreen. From the usability efficiency point of view, improvements like this make the difference because the menu invocation is the most frequent action that an Octopus terminal user employs. The three menu variants are depicted at the Figure 5.

On the technical side, this widget is implemented by a PopupWindow overlaid with a customized ImageView that tracks finger swipe actions. The widget is generated when a LongClick occurs and destroyed when the chosen command is issued. The swipe gestures can be modified easily by editing the `ActionSwipeListener.java` file in case tweaking is required to accomodate a certain touchscreen's configuration.

3.2.2 Pull function

While the user may have numerous, many tens of open apps, he may wish to use the terminal to interact with just a few or perhaps even just one app at a time, this is especially meaningful when using small screens. From this observation emerges the need of a functionality to easily isolate a small number of application UIs. We've implemented the ***Pull*** option for the Android version which helps the user to select a subset of the system's GUI.

Consider the following motivating scenarios. I want to use my mobile phone to browse and add an appointment in my calendar that is displayed on the main screen, along with my other apps. Instead of scrolling through the numerous running application panels, I ***pull*** the calendar related panels to the mobile phone's screen. As another example, imagine the user makes a presentation with a projector and desires to control the slides remotely. He can pull

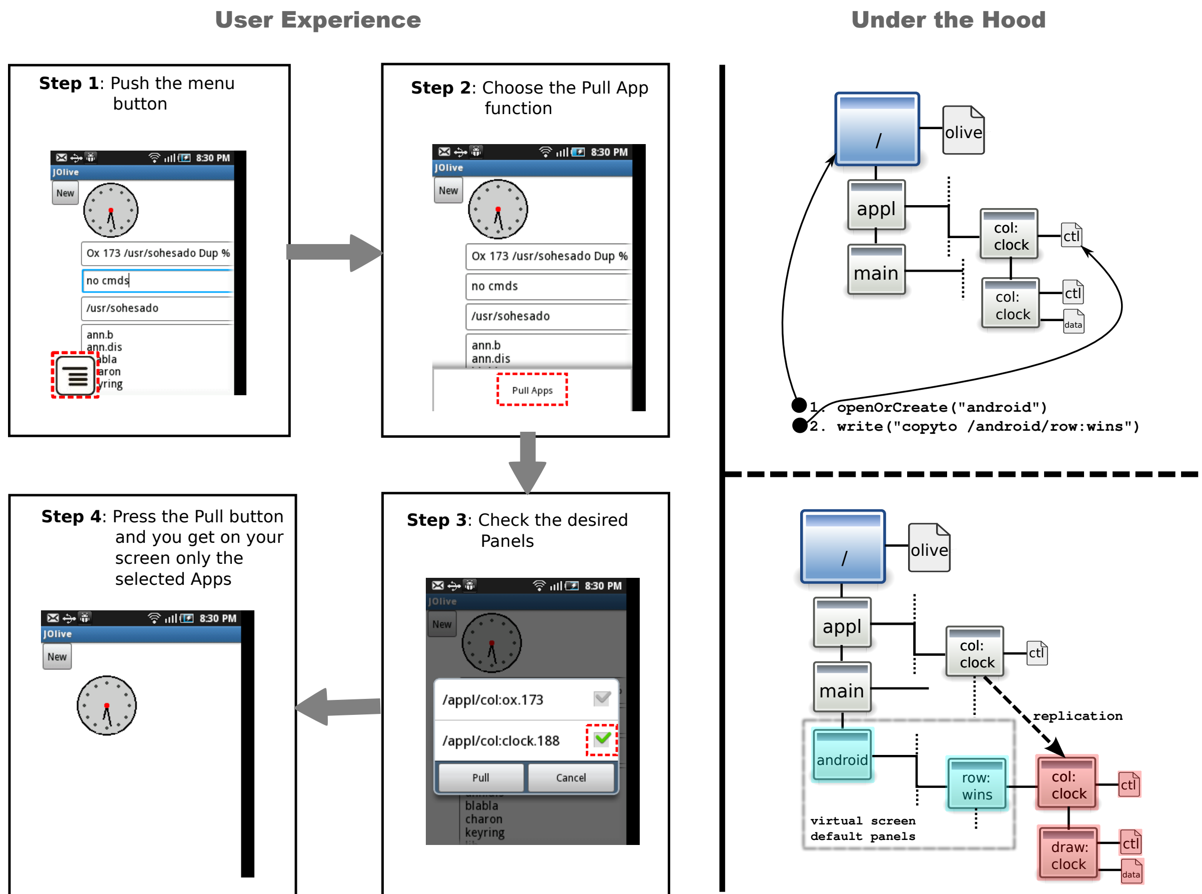


Figure 6: This Figure illustrates the Pull App functionality, at left the user's perspective, at right the underlying process.

the ***next/previous*** buttons to his Android device and use it as remote control. The same thing applies to the case of a multimedia player application or anything that has some sort of graphical control interface and makes sense the remote interaction with that app.

The approach that we adopted in order to implement the pull functionality is based on a popup dialog in order to assist the user to easily select panels. The procedure can be break down to the following steps. (i) Push the menu button. (ii) Select the Pull function. Pops up a checkbox list with the apps running at the PC. (iii) Check the desired apps. (iv) Touch the Pull button and as a result you get on your screen only the panels related to the selected applications. In Figure 6 is illustrated the process of "pulling" the clock application.

Under the Hood, JOlive initializes a new virtual screen by creating a directory at the Omero's root and consecutively issues ***copyto*** [11] commands to the selected panels. Omero copies the file descriptors to the supplied path and generates the corresponding update events. Once JOlive receives that event messages, it presents the newly initialized virtual screen. After the pull is invoked successfully, it will receive updates only for the panels present to that virtual screen, which makes it more efficient.

This effect can be achieved by issuing the appropriate Ox commands. However the extra 30 lines of code or so required to implement this function via a proper dedicated UI element are justifiable in order to avoid the tedious task of typing via touchscreens. With this option, the user with a couple of clicks achieves the result of textual commands that had to be passed manually to Ox.

3.2.3 Trivia

We faced some technical challenges while porting the implementation to the Android environment. While porting JStyx to Android we faced a technical problem related to log4j message logging package because it was incompatible with the Android platform. We had to switch to the compatible alternative that required the android-logging-log4j and slf4j-android along with ConfigureLog4j.java code that configures said packages.

One other unexpected problem caused by the Android runtime environment was the limitation that only the thread that creates a ***View*** object, is allowed to update it. In our original design, the main thread initialized the graphical window (i.e. created the graphical widgets) and a secondary thread received Omero event messages and updated accordingly the respective widgets. This design caused runtime errors so we had to approach the problem in a different angle. We used a ***handler*** object created in the main thread that was asynchronously receiving update messages from the OOlive thread (Table 1) and updating the GUI accordingly.

4 Future perspective

JOlive is an Omero ***viewer*** which is the most important component of a terminal, since it provides the means to interact with an Octopus system. The following subsections describe briefly some interesting resources and other terminal related extensions that can be implemented in order to increase the effectiveness of the Java terminals.

4.1 GPS resources

Modern mobile phones have GPS capabilities. It seems promising, from a pervasive computing environment point of view, to expose the GPS tracking service to the system. The Android terminal could export its GPS coordinates through a passive UpperWare resource driver to the global Octopus namespace, with the intention to enhance the smart space characteristics of the system. Assuming the user carries his phone with him, the system could then automatically infer the location of other mobile computing devices, such as laptops, cameras or other wearable sensors, which in turn could be exploited for context-aware computing purposes.

4.2 JOp

A Java implementation of the Octopus protocol seams interesting. Op is preferred instead of Styx when high latency communication links are encountered. Certainly, it would be interesting to create a Java implementation of Op, at least the client part, which is a useful tool on its own. Indeed, especially the Android version Java terminal could greatly benefit from an Op implementation, given that it will typically communicate with the Octopus computer over WiFi hot-spots or the cellular network (e.g. 3G data transfer network).

4.3 Remote control by voice

The Octopus experiments with the notion of integrating voice support to the system. Smartphones have decent voice recording capabilities. A possible extension of the Android terminal would be to implement the functionality to take advantage of that capability. If we are able to export the voice recording resource of our phone to the Octopus system, assuming there will be a decent audio command server implementation, then the Android terminal would offer many practical applications. In a "smart" space environment, but also when the user is on the move, it can be very practical to issue commands to, but also receive messages from the system, verbally.

4.4 Authentication device

We can exploit the habit of keeping the cell phone within our reach and create an authentication mechanism resource for the Octopus system. We can device a simple and secure mechanism for generating uniquely identifiable tokens that are hard to replicate without possessing a suitable hardware device. One possible way to do this is to combine a user defined touchscreen gesture, e.g., the user's signature, with the hardware ID of the device and generate a digest value that authenticates the user's credentials. The digest value does not give away neither the gesture nor the hardware ID and can be stored to the PC. Even if the secret gesture is "leaked", the token can't be generated without the hardware ID of the user's phone, and vice versa. This resource can be wrapped with a passive UpperWare driver and exploited by other system resources that require a more secure authentication mechanism.

References

[1] Francisco J. Ballesteros, Spyros Lalis, and Enrique Soriano. Building the Octopus. GSyC Tech. Rep, 2006-06.

[2] Francisco J. Ballesteros, Pedro de las Heras, Enrique Soriano, and Spyros Lalis. The Octopus: Towards building distributed smart spaces by centralizing everything. UCAMI, 2007.

[3] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard Trickey, and Phil Winterbottom. The inferno operating system. ***Bell Labs Technical Journal***, pages 5--18, 1997.

[4] Rob Pike and Dennis M. Ritchie. The styx architecture for distributed systems. ***Bell Labs Technical Journal***, 4(2):146--152, April-June 1999.

[5] Francisco J. Ballesteros, Gorka Guardiola, Enrique Soriano, and Spyros Lalis. Op: Styx batching for high latency links. IWP9, 2007.

[6] Francisco J. Ballesteros, Enrique Soriano, Spyros Lalis, and Gorka Guardiola. Improving the performance of styx based services over high latency links. Rosac, Laboratorio de Sistemas, 2 2011.

[7] Francisco Ballesteros, Gorka Guardiola, and Enrique Soriano. Octopus: An upperware based system for building personal pervasive environments. ***Journal of Systems and Software***, 85(7):1637--1649, July 2012.

[8] IEEE Middleware Support for Pervasive Computing Workshop (PerWare). ***Upperware: Bringing Resources Back to the System***, 2010. in proceedings of the PerCom 2010 Workshops.

[9] Gorka Guardiola, Francisco J. Ballesteros, and Enrique Soriano. Upperware: Pushing the applications back into the system. IWP9, 2008.

[10] Francisco J Ballesteros, Enrique Soriano, and Gorka Guardiola. Towards persistent, distributed, and replicated user interfaces in the octopus. IWP9, 2007.

[11] Laboratorio de Sistemas. ***Octopus 2nd. edition User's Manual***, 1 2008. RoSAC.

A light-weight non-hierarchical file system navigation extension

Jonas Amoson
Thomas Lundqvist

University West
Trollhättan, Sweden

{jonas.amoson | thomas.lundqvist}@hv.se

ABSTRACT

Drawbacks in organising and finding files in hierarchies have led researchers to explore non-hierarchical and search-based filesystems, where file identity and belonging is predicated by tagging files to categories. We have implemented a *chdir()* shell extension enabling navigation to a directory using a search expression. Our extension is light-weight and avoids modifying the file system to guarantee backwards compatibility for applications relying on normal hierarchical file namespaces.

1 Introduction

File systems have long been hierarchical, helping both system designers and ordinary users to group their files and thereby hopefully avoiding chaos. Besides this grouping of related files, a *file path* such as `/usr/$user/work/telephone.txt` is also important in that it uniquely identifies a file, directory or other named resource within a given name space, by a sequence of slash-separated strings [3]. While a well structured home directory serves to avoid clutter, it also means that the user will have to remember and type longer paths in order to specify files. This is because the same addressing mechanism is used both for storing files as well as for addressing them later on.

We believe that our user, in the everyday interaction with the computer, could afford some loss of addressing preciseness for the benefit of having to type less. Instead of entering an absolute or relative file path, the file or directory of interest may be specified using a much shorter *search expression*. For example, the command `cd !mydir` will then perform a search for a subdirectory matching `mydir` and make the found directory the new working directory of the process, even if it is located several subfolders deeper down from the current working directory. If the search matches multiple files, the user simply chooses the desired one, or refines the search.

The idea of regarding a path name as a search expression is old. Previous work in semantic file systems or non-hierarchical, tag-based file namespaces [1,2,4] has suggested many ideas and solutions for navigating through files and directories based on search expressions. However, these approaches are rather cumbersome to implement and fail to provide full backwards compatibility with hierarchical file systems.

This paper explores a more light-weight approach of interacting with an existing file system namespace, using the path name as a search string, but keeping backwards compatibility with the existing namespace. We argue that our search mechanism would be difficult to implement in the form of a separately mounted file system since the search capabilities we suggest are really a user interface feature. To explore our ideas, we have made a simple but yet useful prototype implementation in the form of a shell extension that enables powerful new ways of file system navigation.

2 The idea: a file system navigation extension

From a user's perspective, the search feature we propose is a simple extension of the syntax of path names used by the shell. In our implementation, we have picked the character "!" as a prefix to denote a search.

To illustrate the behaviour, consider the following set of directories:

```
/usr/jam/work/lectures/cs101/mydir/
/usr/jam/work/lectures/plan9/
/usr/jam/work/plan9/
```

Let us assume that the current working directory is `/usr/jam/` and that we issue the command `cd !mydir`. This will tell the shell to perform a search for a subdirectory with a name matching `mydir` and change the working directory to the specified directory. Further, the command `cd !cs101` would take the user to `/usr/jam/work/lectures/cs101` and not to its subfolder `cs101/mydir`, keeping the matching directory as shallow as possible.

If there is no unique directory match, the relative paths for all matching directories are listed, and the user could either use one of them, or refine the search by adding substrings separated by slashes to the search expression. The command `cd !plan9` would for instance match two directories, resulting in a failed command and a listing of possible alternatives. Refining the search to `cd !plan9/lect` would succeed and match the folder `work/lectures/plan9`. Only folders "below" the current working directory are considered in the search.

We believe that the possibility of searching directly in a path name is more convenient than the alternative of using existing tools for searching the file system. The command *du(1)* could be used by the user to find usable file paths, but the user would most likely not want to invoke such search commands in the midst of specifying the third file argument of a program to run. The handiness of fuzzy path searches can be compared with filename completion provided by `INS` in *rio(1)* or tab-completion in Unix shells.

A possible problem with our solution could be the use of directories or file names starting with the character "!", thereby needing some way of escaping this character. We are unsure about the magnitude of this problem however.

3 Related work

Since the invention of hierarchical file systems, many researches have pointed out various limitations in organising data storage files in a strict hierarchy. Already in 1991, Gifford et al. [2] presented the idea of a semantic file system where path names can be used as a search string by the user. For example, by writing `cd ext:/c`, you go to a virtual directory containing all files matching the search, in this case having the extension ".c". Their semantic file system idea represents a virtual read-only file system containing symbolic links that point to an underlying regular Unix file system.

Influenced partly by the collaborative *tagging* found in on-line community sites such as *flickr*[1], Stephan Bloehdorn and Max Völkel [1] implemented a file system *TagFS* where files are not organised in directories, but where each file is assigned multiple *tags* by the user. When a file is recalled later, the tags are used as parts of the path similar to an ordinary hierarchical file path. Walking the "directory" structure (`cd` *tag*) yields a search, and the contents (as displayed by `ls`) shows the possible tags that could be used to further narrow down the search. To achieve accessibility from different operating systems and easy integration over the Internet, *TagFS* is implemented as a WebDAV[2]-server using the http-protocol.

Margo Seltzer and Nicholas Murphy [4] also suggest the demise of the hierarchy in file storage, and give the ubiquitous Google-search as a prime example. Seltzer and Murphy have implemented a file system, *hFAD*, similar to *TagFS*, based on FUSE[3] under Linux.

4 Comparison with related work

Compared to these earlier approaches [1,2,4], our shell extension does not modify the underlying file system and thereby maintains full compatibility with an hierarchical namespace.

An important property of our idea is that the `cd` command will, as usual, take you to an existing directory. An incomplete search path will make the `cd` command fail. This means that we always have a well defined current working directory. The latter is important since some

[1]Flickr online photo management (http://www.flickr.com).
[2]Web-based Distributed Authoring and Versioning (http://www.webdav.org).
[3]Filesystem in Userspace (http://fuse.sourceforge.net).

filesystem operations, like creating new files, will be difficult to support otherwise. This can, for example, not be guaranteed for the semantic file system presented in [2] since a virtual folder can be a result of a search returning a union of files from multiple locations. This is true also for the two other approaches where a *directory* can be the result of a union of all files from multiple file system locations.

Another issue with previous approaches is that one file can be identified by many different path names, which could confuse an old-fashioned application that relies on unique file path names.

5 A modified shell implementation

As a first attempt of an implementation we have modified the *cd* (change directory) command in the shell *rc(1)* to accept a search expression prefixed by an exclamation mark.

In *rc*, the path is examined before calling *chdir(2)*. If the path is prefaced with an exclamation mark, the command *du(1)* is invoked to get a list of directories relative to the current working directory. The resulting directories that do not match all search criteria are discarded. Then, the list is further processed to prune sub directories that would overspecify the search. Finally, if only one path remains, *chdir()* is called with the remaining path, otherwise the candidate path names are printed to standard output. The source code for the shell extension and a helper program `dugrep` is available at `http://cumulus.ei.hv.se/~imjam/ref/spath`.

6 Discussion and future work

Our shell extension approach suffers from the nuisance that it is not always possible to narrow down the search, given a set of possible paths. For example, consider the two paths: `work/lectures/` and `lectures/work/`. Navigating using `cd !work/lectures` would match both paths without giving the user any possibility of further refining the search. One solution to the problem would be to force the user to resort to the normal *cd*-command and simply specify the precise path. Another solution would be to extend the search syntax and let the user choose among the alternatives in some way. An interesting observation here is that a set of well-organised directories would typically not trigger this problem.

There are promises in getting beyond the traditional file hierarchy, using a tagging filesystem, especially for certain types of personal files such as lecture notes and media files, but the non-hierarchical tagging ideas might not work as well for directories where the “belonging together” property of files is more important than the uniqueness of individual files, such as in a source code tree.

We believe that the search extension idea, as presented in this paper, would be difficult to implement as a synthetic file server. To do so, it would require some kind of “virtual folders” to represent searches, possibly leading to problems with file creation and unique identification of files.

In the meanwhile, we have already grown accustomed to the handiness and simplicity of the *search-path* extension, and will try to develop it further, with general file and directory search-expansion for all command-line arguments. Another open issue is how the our search capabilities can become integrated with filename completion as provided by `INS` in *rio(1)* or tab-completion.

References

[1] S. Bloehdorn, O. Gorlitz, S. Schenk, and M. Volkel. Tagfs – tag semantics for hierarchical file systems. In *Pro. The 6th International Conference on Knowledge Management (I–KNOW 06).*, 2006.

[2] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O’Toole, Jr. Semantic file systems. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP ’91, pages 16–25, New York, NY, USA, 1991. ACM.

[3] R. Pike and P. Weinberger. The hideous name. In *USENIX Summer 1985 Conference Proceedings*, page 563, Portland Oregon, June 1985.

[4] M. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS’09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

Networking in Osprey

Jan Sacha
Sape Mullender

Alcatel-Lucent Bell Labs
Antwerp, Belgium

Abstract

Network links have more and more bandwidth while processor frequencies do not increase significantly and thus the best way to improve networking performance is to process packets in parallel on multicore machines. This paper describes a networking architecture where incoming network traffic is demultiplexed to user-level network protocol stacks running on different cores using a software packet filter and multiple hardware receive rings. Such architecture allows efficient use of the network interface controller, processor caches, and memory, enabling very efficient and scalable networking. This architecture has been implemented in the Osprey operating system.

1 Introduction

Networking stacks in many operating systems were designed when networks were slow and machines typically had a single CPU with one core. As a consequence, the networking code often has a monolithic, centralized structure where concurrent access is guarded by locks. However, hardware realities are changing. Network links have increasingly high bandwidth and require more and more computing power to process packets. Since processor frequencies do not increase significantly, but the core count per machine grows quickly, the most straight-forward way to improve networking performance is thus to process packets in parallel.

Parallelizing the networking stack is challenging, however, because the networking code can be invoked concurrently from multiple different contexts: User applications pass their requests to the stack using system calls; Packets asynchronously arrive from the network and are typically signaled by interrupts; Finally, protocols such as TCP require timers, which again are delivered asynchronously. Accessing shared networking state from multiple CPU cores is expensive due to the cache coherency semantics. Further, locks prevent parallel execution and might waste a significant fraction of CPU cycles in case of contention.

Hence, in order to achieve high throughput, the networking stack should be organized so that cores share and synchronize as little as possible. A few novel architectures have been proposed recently to reach this goal. In the IsoStack [9] and netmap [6] frameworks, the networking functionality is delegated to a single process which serializes requests to avoid locks and cross-core sharing. In NewtOS [3], cores specialize at performing particular functions, such as filtering packets, running IP, or TCP. In the Affinity Accept design [5], network traffic is divided between symmetrically running cores using hardware support so that each packet is processed on one core only. This latter approach has been shown to allow very high throughput and scalability.

In this paper, we describe an architecture where incoming packets are demultiplexed in software using a packet filter and delivered to a protocol stack running in the application's address space. Such a design does not require hardware support but allows processing packets in parallel with very little inter-core sharing. We describe optimizations in which applications can own private Ethernet buffer rings and use hardware scatter-gather capabilities to avoid memory copying, increasing overall throughput. Our architecture is implemented in the Osprey operating system [7] but we believe it can be ported to other operating systems as well.

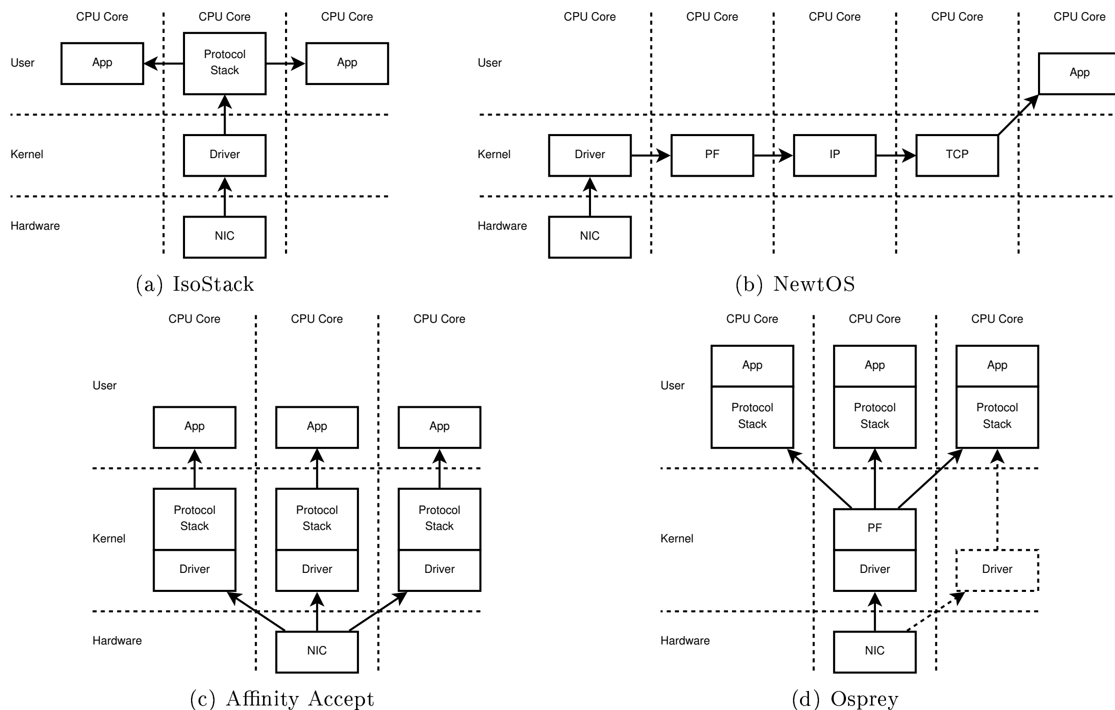


Figure 1: Networking stack organization in sample operating systems.

2 Related Work

The earliest approaches to high-speed networking came from the area of high-performance computing (HPC) where high-bandwidth and low-latency networks, such as MyriNet or InfiniBand, were used to connect powerful compute nodes. Since general-purpose networking stacks in commodity operating systems turned out to be too slow to handle traffic generated in these systems, new networking architectures were developed. HPC applications typically were allowed to interact directly with the network interface controller to bypass kernel abstractions and reduce packet processing overhead. These approaches often provided very little application isolation and security because all hardware was usually owned and controlled by a single HPC application [11, 4, 1].

Today, high-speed networks such as 10G Ethernet become commodity and support for them is added to general-purpose operating systems such as Linux. Figure 1 summarizes a few networking architectures that allow fast packet processing.

In IsoStack [9], the networking stack runs in a single process which serializes events coming from user applications (via asynchronous message queues), the network interface, and the operating system (timers). Such an arrangement allows IsoStack not to use locks and minimizes intercore sharing, allowing efficient use of CPU caches. However, the IsoStack approach does not scale because it can utilize only one core. Similarly to the IsoStack, Osprey runs networking stacks in user processes, but it divides incoming traffic using a packet filter and runs multiple stack instances in parallel to take advantage of multiple cores available in hardware.

NewtOS [3] partitions the networking stack based on functions. Each functional component, such as a device driver, packet filter, IP implementation, and TCP engine, runs on a different core. Network packets are processed by multiple cores in a pipeline. Since cores share very little state and communicate using asynchronous message queues, they can run in parallel. However, this design increases packet processing latency, as every packet traverses multiple cores, and is not able to utilize more cores than the number of networking stack components.

In the Affinity Accept framework [5] all cores perform the same function but network traffic is multiplexed between them using multiple hardware receive queues. Each core maintains a fraction of the networking state using lock-free data structures. Heuristics, such as packet stealing, are developed to match incoming packets with receiving applications and to balance the load between

CPU cores. This approach scales very well and hence enables high throughput. Similar to Affinity Accept, Osprey uses hardware receive queues to multiplex traffic, but it also allows multiplexing in software using a packet filter. Further, Osprey runs protocol stacks in the user space for benefits such as simplified memory allocation, application checkpointing, restarting, and migration.

Netmap [6] is a framework which allows an application to send and receive packets from a network interface very efficiently. It allocates all packet buffers statically at initialization time and maps them into the application's address space to avoid copying. The application receives all network packets from an Ethernet hardware ring and returns empty buffers. The kernel does not multiplex traffic. In Osprey, we use a similar technique to optimize key applications, such as file servers and clients, which can own private Ethernet rings.

Operating systems can improve networking performance by offloading some functions to the network interface controller (NIC). Such offloading can usually be combined with the techniques described above. Modern NICs usually provide checksum calculation and verification, interrupt throttling, header separation, and TCP packet splitting and coalescing. These latter capabilities are known as Large Send Offload (LSO) and Large Receive Offload (LRO) or TCP Segmentation Offload (TSO).

Finally, modern NICs support multiple transmit and receive rings (also called queues) where incoming packets are classified and assigned to the rings based on their source and destination addresses. For example, Intel's Virtual Machine Device queue (VMDq) technology, intended for hypervisors running multiple operating systems, assigns packets to rings based on their destination Ethernet address. Another technology, known as Receive-Side Scaling (RSS), uses a hash from the incoming packet's 5-tuple to select the receive ring. Each ring uses its own buffers and requests interrupts independently, allowing packets to be processed on multiple cores in parallel. Further, packets belonging to the same connection are mapped to the same ring and are hence processed on the same core, improving CPU cache performance.

3 Architecture

Osprey's networking architecture assumes that every network interface has at least one hardware receive (RX) ring and one transmit (TX) ring. Receive ring zero can be shared by multiple applications and is multiplexed by a packet filter. Other receive rings, if available in hardware, can be owned by applications in a way similar to netmap: the application receive all incoming packets with no multiplexing in the kernel, no memory copying, and very little overhead.

Osprey uses only one transmit ring to send all outgoing packets. It does not use multiple transmit rings so that it can decide itself on the order in which packets are enqueued for transmission. If it used multiple hardware transmit rings it would have to let the NIC decide which packets are transferred from the rings to the wire.

In Osprey, every application has its own protocol stack which runs in the user space. Similar to the IsoStack, an Osprey protocol stack can serialize events from the application and kernel, such as packet transmissions, arrivals, and timeouts, in order to work efficiently with no synchronization overhead. The protocol stack runs together with the application—in the same address space and on the same core—which enables zero-copy communication and efficient use of CPU caches. Keeping the networking state in the user space also facilitates application checkpointing, restarting, and migration. From the kernel's view point, the network protocol stack is just part of the application's logic. However, in contrast to IsoStack, every Osprey process can have its own protocol stack, which allows the system to scale better. The only centralized components in the networking architecture are the device drivers and the packet filter.

Osprey inter-process, inter-task (inside the kernel), as well as user-kernel communication is provided by asynchronous message queues. Queue implementation details fall beyond the scope of this paper, however, we mention here that the queues are based on fixed-size shared-memory buffers and fixed-sized cache-aligned messages (64 byte long currently) which are copied on both ends during send and receive operations. Messages typically contain pointers to larger structures such as packet buffers. Asynchronous communication reduces the number of context switches and hence improves performance. In many ways, Osprey's system call API is similar to FlexSC [10].

The asynchronous networking API consists of just a few message types. In order to manage a

```
interrupt_handler()
{
  disable_nic_interrupts();
  send_message(driver, interrupt);
}

transmit(packet)
{
  send_message(driver, packet);
}

driver_main()
{
  loop {
    message = receive_message();
    switch(message.type) {
    case interrupt:
      while(tx_complete())
        send_message(protocol_stack, send_ack);
      ...
      enable_nic_interrupts();
      break;
    case packet:
      tx_ring_append(message);
      break;
      ...
    }
  }
}
```

Figure 2: Driver task pseudocode.

hardware receive ring, an application sends a **ringattach** message containing an interface number and a ring number. Similarly, to use the shared ring zero, the application sends a **netattach** message containing an interface number and a bitmask describing packets that the application needs to receive.

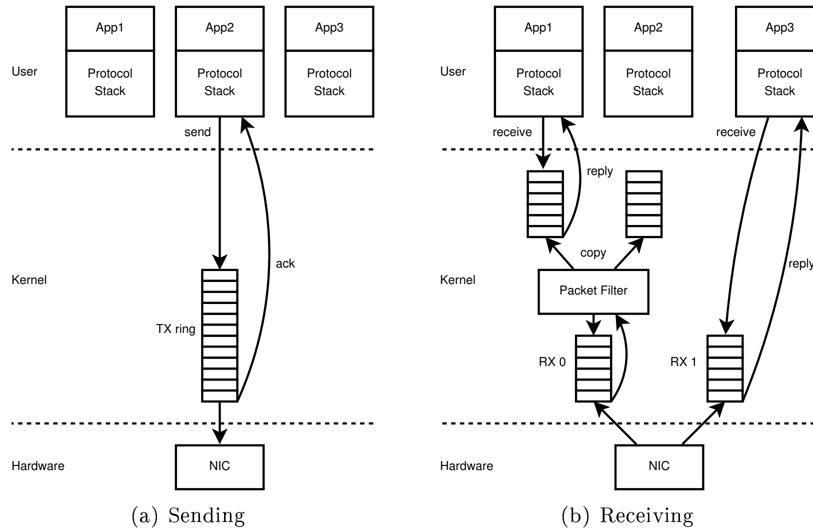
An application sends a packet by issuing a **send** message containing a pointer to the packet buffer and a length. When packet transmission is complete, the kernel sends an acknowledgment message to the application. Similarly, to receive a packet, an application sends a **receive** message to the kernel containing a pointer to an empty buffer and a length. When a packet arrives from the network, the kernel—either the packet filter or the driver managing a private hardware ring—returns a buffer containing packet data. Importantly, the message format is the same for all receive rings, so that applications can easily switch between using shared ring zero and private hardware rings.

Finally, although the networking API is asynchronous, we note that it is very easy to build a synchronous API on top of it. For a synchronous send, an application generates a **send** message and waits for a matching reply, and similarly, for a blocking receive, the application generates a **receive** message and waits for a corresponding reply.

3.1 Device Drivers

Every network interface has its own driver task which communicates with the controller using ports and memory-mapped registers. The driver serializes events coming from hardware and applications to execute efficiently in a lock-free manner. The interrupt handler is trivial. It sends a message to the driver and disables further interrupts from the interface. The driver, upon receiving an interrupt message, inspects the hardware, updates the necessary state, and if a packet transmission or reception is complete, generates a message to the process or kernel task (packet filter) owning the respective ring. When hardware maintenance is finished, the driver enables interrupts again and blocks waiting for a message (see Figure 2).

Similarly, the driver accepts messages coming from user processes and kernel tasks. Upon



(a) Sending (b) Receiving

Figure 3: Osprey's networking architecture overview.

receiving a packet transmission request, the driver appends the packet to a TX ring, and upon receiving a packet receive request the driver appends an empty buffer to an RX ring.

In the current Osprey implementation, every NIC has a single driver task which maintains one TX ring and all available RX rings. However, it is possible to further parallelize the architecture by running a separate driver task per each hardware ring. Since many NICs allow rings to signal interrupts to different cores, drivers tasks could run truly in parallel and service rings independently, improving system scalability on multicore hardware.

3.2 Packet Transmission

Outgoing packets normally are allocated and filled in by the protocol stack in the user space. Kernel tasks are allowed to generate network packets but Osprey rarely uses this possibility. Once a packet is ready to be transmitted, the protocol stack sends a message containing a pointer to the packet body to the driver task managing a chosen network interface (see Figure 3(a)).

The driver does not need to copy the packet into the kernel space but must verify that memory is mapped in and must pin pages to make sure that they are not freed during packet transmission. Pinning pages in Osprey is simple because memory is never swapped out to disk. Further, the kernel verifies that the packet is legal. Since the networking API is asynchronous, the application is allowed to run during packet transmission, and hence could potentially modify the packet after it has been verified by the kernel. To prevent such a possibility, the kernel copies and transmits the header to its own memory.

When packet transmission is complete, the device driver sends a reply message to the process owning the packet. The protocol stack, upon receiving this message, frees the packet buffers and cleans up its internal state. If the user application uses a synchronous networking API, the thread blocked on transmission is woken up.

Older NICs might impose constraints on the packet layout in memory. For example, some NICs require a specific memory alignment, or are able to access only a subset of the address space, or require packets to be contiguous (i.e., do not support scatter-gather). If user-space libraries do not allocate packets in the way expected by these NICs, Osprey falls back to a compatibility mode where the entire packet is copied to the kernel space before transmission.

We are planning to extend the networking architecture to support resource usage management. In the future, the kernel will limit the number of packets and bytes a process can send per time unit. When these constraints are not met, the kernel (e.g., device driver) will delay packet transmission. Further, the kernel will append outgoing packets to the TX ring according to a policy, for example based on process priority or deadline, to allow low-latency network access to privileged processes.

3.3 Packet Reception

In order to receive network packets, a protocol stack needs to provide empty receive buffers to the networking infrastructure (see Figure 3(b)). These buffers are allocated in the user space, and similarly to outgoing packets, must be verified by the kernel. In particular, memory must be mapped in and pages need to be pinned so that buffers are not freed before or during packet reception. Additionally, buffers must be large enough to fit maximum-size packets.

Typically, when a protocol stack is initialized, it allocates a number of receive buffers and sends them to the network interface. Further, each time the protocol stack receives an incoming network packet, it provides a new empty buffer to the interface. By maintaining a fixed number of buffers on the receive ring—either hardware ring managed by the NIC driver or virtual ring managed by the packet filter—the protocol stack reduces the risk of running out of buffers and losing incoming packets.

3.3.1 Shared ring zero

Ring zero is special because it is owned by a kernel task—the packet filter—which multiplexes it between user applications. Such multiplexing is necessary because NICs usually provide a limited number or rings, often only one, and thus may not have enough receive rings for all applications. Further, enabling hardware rings introduces a cost which might be undesired for lightweight or short-lived applications.

Osprey's packet filter classifies packets based on their headers. For each incoming packet, the packet filter extracts key fields from the header, such as protocol types, source and destination addresses, and port numbers, and matches them against bitmasks provided by user applications. The details of the packet classification algorithm and its implementation are described in a separate paper [8].

When the packet filter determines the receiving application, it checks if an user-space buffer is available. If the virtual ring is empty, the packet is dropped. If a user buffer is available, the packet is copied into the user space and a message is sent to the application's protocol stack. The original kernel buffer is always returned to hardware ring zero and reused. This way, the packet filter allocates receive buffers only once—at initialization time.

It is worth noting that currently existing NICs do not have sufficient functionality to implement a packet filter entirely in hardware. In particular, the Intel VMDq technology allows only very simple filtering based on the packet's destination Ethernet (MAC) address, and the RSS technology uses 5-tuple hashes and hence may map packets belonging to different applications to the same ring, requiring further demultiplexing in software. Besides, as already mentioned, most NICs support only a limited number of RX rings.

3.3.2 Private rings

To take advantage of hardware support, Osprey allows applications to use private receive rings. We associate every RX ring with its own MAC address and use the NIC to demultiplex incoming packets. Private rings allow much more efficient networking than the shared ring zero. They do not require software multiplexing, since the application receives all incoming packets, and allow the application to put user-level receive buffers directly on the hardware ring, enabling zero-copy receive.

However, using private receive rings has an impact on the application. First of all, every application owning a private ring must obtain its own IP address—or use a non-IP protocol, such as ATA over Ethernet (AoE). Secondly, Ethernet broadcast packets have a destination address of FF:FF:FF:FF:FF:FF, and hence are all delivered to the same RX ring. In Osprey, we configure the NICs so that broadcast packets are received by ring zero. As a consequence, it is not possible to run protocols such as DHCP or ARP on a private ring because these protocols rely on Ethernet broadcast. To work around this limitation, Osprey kernel handles ARP lookups and runs DHCP on ring zero on behalf of applications that own private RX rings. Given that both ARP and DHCP are critical to network security, we do not view this as a drawback. Most applications should not be allowed to generate ARP and DHCP replies nevertheless.

```
struct Msgpacket {
  void    *addr[4];
  ushort  len[4];
  ushort  flags;
};
```

Figure 4: Message format for incoming and outgoing packets.

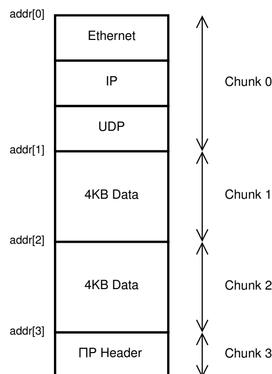


Figure 5: ΠP packet layout.

3.4 Scatter-Gather

So far we have assumed that packet buffers are contiguous in memory and hence can be referenced using a single pointer variable. However, to give more flexibility in packet allocation to the protocol stacks, Osprey supports packets which consist of multiple disjoint chunks of memory. This feature allows for example TCP implementations to combine multiple pieces of data from the user space into a single packet without copying memory.

Messages representing packets have a format shown in Figure 4. A single message can store up to 4 pointers to packet chunks and 4 corresponding lengths. The limit of 4 chunks is imposed by the maximum message size in Osprey, which is currently 64 bytes. The additional `flags` field is used by the device driver to indicate to the protocol stack which checksums have been calculated and verified (e.g., Ethernet CRC, IP header, TCP, or UDP).

Sending and receiving packets consisting of multiple memory blocks requires scatter-gather capabilities in hardware, which most modern NICs already provide. On older hardware, Osprey can always copy fragmented packets into consecutive buffers to overcome NIC limitations.

Modern NICs are also able to parse incoming packets and split them into separate buffers containing the headers and payload. In Osprey, we are planning to use this feature to implement a zero-copy network file protocol called ΠP. In this protocol, a packet consists of network protocol headers, such as Ethernet, IP, and UDP, a ΠP header (known as the operations), and data. Our plan is to align the data buffer on a page boundary so that it can be transferred to an application's address space by updating memory maps. While NICs usually recognize common network protocols, such as IPv4, UDP, and NFS, they regrettably do not support ΠP. However, to make sure that ΠP data is properly aligned, we use a packet wire representation shown in Figure 5. The ΠP header is written at the end of the packet. When the network headers are split off in hardware, data is stored at the beginning of the payload buffer which can be aligned on a page boundary.

4 Status

The networking architecture described in this paper has been mostly implemented in Osprey. We have implemented drivers for a number of Intel NICs including the 82598 10GbE, 82576 1GbE, and older models, as well as the portable kernel components such as the packet filter.

We have also developed a user-space library which provides memory buffer management and a synchronous (blocking or non-blocking) API on top of asynchronous message transactions. We used this low-level library to implement a number of basic network protocols, such as DHCP, ARP, Ethernet, IPv4, and UDP. Finally, we have ported the Lightweight IP library [2] to provide a fully functional user-level TCP/IP stack for Osprey applications.

5 Conclusion

This paper describes a networking architecture where incoming network traffic is demultiplexed using a software packet filter and hardware receive rings to multiple network protocol stacks running in the user space concurrently on different cores. Such architecture allows efficient use of network interface controllers, processor caches, and memory, enabling very efficient and scalable networking. The architecture has been implemented in Osprey. We are planning to run a number of experiments to measure, understand, and validate this architecture's performance.

References

[1] P. Buonadonna, A. Geweke, and D. Culler. An implementation and analysis of the virtual interface architecture. In *ACM/IEEE Supercomputing*, pages 1–15, 1998.

[2] A. Dunkels. Design and implementation of the lwip TCP/IP stack, Feb 2001.

[3] T. Hruby, D. Vogt, H. Bos, and A. S. Tanenbaum. Keep net working - on a dependable and fast networking stack. In *Dependable Systems and Networks*, pages 1–12. IEEE, 2012.

[4] S. Pakin, V. Karamcheti, and A. A. Chien. Fast messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Parallel & Distributed Technology*, 5:60–73, 1997.

[5] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *EuroSys*, pages 337–350. ACM, 2012.

[6] L. Rizzo. Revisiting network I/O APIs: The Netmap Framework. *Communications of the ACM*, 55(3):45–51, Mar. 2012.

[7] J. Sacha, J. Napper, S. Mullender, and J. McKie. Osprey: Operating system for predictable clouds. In *Proceedings of the Dependable Systems and Networks Workshops*, pages 1–6. IEEE, 2012.

[8] J. Sacha, J. Napper, H. Schild, and S. Mullender. Revisiting user-level networking. In *Proceedings of the 6th International Workshop on Plan 9*, pages 17–24, 2011.

[9] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack: highly efficient network processing on dedicated cores. In *USENIX Annual Technical Conference*, pages 5–5, 2010.

[10] L. Soares and M. Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *OSDI*, pages 1–8. USENIX Association, 2010.

[11] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. In *SOSP*, pages 40–53. ACM, 1995.

ΠP
a new way to use the Internet

Sape Mullender

Jeff Napper

Bell Laboratories
2018 Antwerpen

Francisco Ballasteros

Universidad Rey Juan Carlos III
Madrid

1. Introduction

At the beginning of the 20th century, people went to theatres to be entertained and large numbers of entertainers (actors, magicians, comedians) travelled the length and breath of the country to amuse the crowds. In the first half of the 20th century, crowds started going to the movies and actors could stop travelling and settle in Hollywood. In the second half of the 20th century, radio and television created the couch potato. But the entertainment was still scheduled: to watch a particular item, people had to go to a theatre on the right date, or they had to turn on the TV at the correct time. Towards the end of the 20th century, recording devices (DVRs, DVDs, ...) gave people the possibility to be entertained precisely when it suited them in a place of their choice.

And now there is the Internet to make it all even easier and more flexible. It is predicted that, soon, more than ¾ of Internet traffic will be for the purpose of home entertainment. Where a popular show was broadcast just once and watched by millions of people at the same time, popular shows are now being *downloaded* millions of times. Such numbers of downloads cannot be supported by a single server, so the data must be replicated.

Information-Centric Networking is a new research area that sprouted up in response to this massive downloading of media data — also known as *content*. ICN is about identifying *what* data is being transported in an attempt to allow combining downloads of the same data on the same network links. The approaches vary in the degree to which this is done: the Content-Centric Networking approach of Van Jacobson et al. [2009] probably goes furthest in proposing using named information blocks as the primitive building block of a new Internet that routes on *names*. Services like Akamai [Dilley et al. 2002] and Velocix locate large data repositories around the world to which requests for downloads are redirected to shorten the download paths.

All these approaches have in common that the information that is being downloaded is *named* and that a nearby copy of the information requested is sought and, if available, used to satisfy the requester.

We believe that *naming* information is important and that a future Internet can no longer be based entirely on the creation of anonymous end-to-end flows between pairs of IP addresses. This is how it's done today, at least in principle — in practice, there's a lot of cheating going on: DNS mapping tricks, redirection, &c. [Kangasharjua et al., 2001].

Making named-information retrieval efficient requires a combination of communication and storage — consumers of content do not ask for the same data at precisely the same moment. To get the advantage of combining requests for the same data on the same link, that data will have to be buffered somewhere.

Until now, there was a strict separation of concerns. Networks *transport* information from one place to another. File systems *store* information for future readers. And processors, along with the people operating them, *produce*, *transform*, *consume*, and *interpret* the information.

We argue that there is very little difference between transport and storage. A network moves data from *there* to *here*. A file system moves it from *then* to *now*. A *distributed* file system, in fact, moves data from *then and there* to *here and now*.

We believe that the distributed-file-system model is an appropriate start for modern-day information-centric networking. The emphasis is shifting from the network to the information itself. The network, as well as the storage infrastructures around the network are merely the tools we use to get the information we seek where and when we want it.

There are three aspects that are characteristic of (distributed) file systems that barely crop up in discussions of information-centric networking:

1. File systems allow files to *modified* as well as merely read/downloaded. Content-Delivery Networks are typically designed to deliver immutable data. If data is changed at all, it is by mechanisms outside those delivery mechanisms.

2. File systems enforce *access-control* policies. The authentication and access control mechanisms are fundamental to file systems, distributed ones especially, but come as something of an afterthought to most CDNs.

3. A final and certainly very important reason for considering a distributed file-system model for named-information systems is that it is a familiar model *that most applications we have can already use*. Word-processing software, photo-manipulation software, spreadsheets, source-code management systems, you name it, they can all deal with files — actually, they can usually *only* deal with files. And, as often as not, these applications cannot deal with Internet objects: objects must be downloaded using a browser, saved to a file and can only then be manipulated by word processors or what-have-you. Writing objects back to where they came from after modification is almost beyond contemplation today.

   All of that is easy if the objects are files already. On Windows, one could view the objects in the internet as files in a special drive: `Q:\www.diversiorum.org\sape\index.html`; on Linux, one can mount the Internet in the local tree: `/net/www.diversiorum.org/sape/index.html`.

These three points are important. The Internet is not a secure place today, as all the spam we receive bears witness to. Firewalls are useful for redirecting incoming connections to the hosts designed to deal with it, but their protocol filtering (TCP only and then a few ports only) only serves to hinder performance and doesn't add much to security at all.

The importance of using a familiar access model and backward compatibility is also hard to overestimate. Named objects are a very familiar model and files and folders even more so. The wrong that was created when sockets came to Unix — everything was no longer a file with *read* and *write*, but we suddenly had *send* and *receive* as well — is being righted by modelling the whole Internet as a giant file system.

2. ΠP

We have set out to design a set of protocols that enables the management of named information in the Internet. The protocols are predicated on the assumption that a large set of autonomous systems agree on a common set of protocols that achieve the goals of named-information systems: the recognition and consolidation of duplicated information requests, and that these systems arrange themselves in *federations* of systems working together to manage subsets of named objects.

In other words, it is certainly not the intent to build a single, world-wide distributed storage system with some sort of central control. The idea is that, based on loose federations of clients, caching nodes and servers, the goals of content distribution can be met.

But content distribution is not the only issue in today's Internet. Another important issue is that proper support for mobility and wireless connectivity are sorely lacking. Mobility implies frequent address changes and wireless connectivity means that connections can be brittle.

We are working on a small set of protocols that can replace a large set of protocols used in the Internet today. Among these are certainly RTP, TCP, HTTP, HTTPS and Mobile IP, but not IP itself. Our protocols will layer perfectly on top of IP (v4 or v6, with or without UDP).

Media data is one of the most prominent data types in the Internet today and interactive forms of media data are used in internet telephony, teleconferencing, video calls and more. Supporting the lowest possible end-to-end latency in interactive media transport is essential (and, if that improves latency, some data loss should often be tolerated). And low latency is also useful, of course, for all other forms of interactive communication.

Support for *mobile devices*, mobile data and even servers that move around in the cloud is becoming more essential all the time. It will not be long before the majority of devices on the Internet is mobile . One thing that characterizes many mobile devices is the use of wireless networks and the handover from one antenna to another, from one wireless technology to another and from wireless to wired or *vice versa*. What is common in all wireless networks is variable latency, variable reliability and variable bandwidth. And what is common too is that mobiles cannot always hold on to the same address. ΠP is designed to deal efficiently with address and network changes and with intermittent connectivity.[1]

Security is not built in to any of the lower layers of the Internet. As a result, denial-of-service attacks are easy to carry out and most corporations need elaborate protection against random incoming connections: there are too many machines in a corporation that are vulnerable to attack.

Firewalls provide two types of protection. They restrict the set of machines that can be contacted from the outside and they restrict the set of protocols that can be used to contact those machines. There is no doubt that the first type of protection is eminently useful. As far as the protocol restrictions go, there is a tendency to restrict protocols to the extent that HTTP(S) over TCP is almost the only protocol left and this is leading to the layering of a wide variety of services on top of HTTP(S). From the viewpoint of protecting resources, this development is of doubtful utility. Instead, by providing security as an integral part of communication (where anonymity is explicitly supported but needs to the approval of communicating principals) a great deal of trouble can be prevented. ΠP does this.

3. ΠP Protocol Overview

The πp protocol covers the OSI model's Session and Transport layers and perhaps a piece of the Presentation layer as well. It is a *client/server* protocol in which a client makes requests to a server; the server carries them out and returns replies. This is a tried and proven communication model in distributed systems.; Amoeba [Mullender et al., 1990] used it and so did V [Cheriton, 1988]. A πp client can be an application or a cache. A server can be a cache, a file server or web server, or an application that provides its interface as a service. If there is a cache between a client and a server, the client interacts with the cache and the cache (independently, as far as πp is concerned) interacts with the server.

The protocol's components are **Sessions**, **Message Transactions** and **Operations.**

Operations are clumped together in *operation*groups and sent as a unit in a *request group*. After receiving the group and processing the operations, the server returns a *reply group* to the client to complete the *message transaction*[2]. Message transactions belong to a *session* which keeps track of the reliable execution of these transactions, the authenticity of client and server and the management of mobility. We'll discuss the components in turn.

---

[1] In today's Internet, the TCP connections that form sessions for carrying out credit-card transactions over mobile connections break often enough that merchants always ask for an email address before starting the actual transaction (so the result of the transaction can be mailed if the connection fails).

[2] Message transaction is a term from Cheriton's [1988] V distributed system, see also RFC 1045

3.1. Operations

The operations are the basic commands for manipulating named objects that we'll call *files*, whether or not they really are stored on a disk. In our model, files have hierarchical names, they have contents — an array of bytes — and they have attributes — name/value pairs. The attributes can have arbitrary names (which are UTF-8 strings) and values (byte arrays); some of the attributes are predefined, with semantically restricted content, and some may be read-only. Implementations may place restrictions on the sizes of names and values.

ΠP took much of its inspiration — including the name — from the 9P protocol of Plan 9 from Bell Labs [Pike et al., 1995]. ΠP also has the notion of T and R operations, of Tsession, Tattach, Topen, Tflush and more,

The files are *versioned* and versions are identified by a signed 64-bit time stamp (in nanosecond before (negative) or after (positive) the epoch (the start of 3rd millennium). A timestamp value *t* identifies the version with the highest time stamp less than or equal to *t*; that is, the version *current* at time *t*. Every update of a file creates a new version that becomes *immutable* and visible when the updates are *committed*. It is implementation dependent whether non-current versions of a file are retained, or even whether the current version is retained — files can be *synthetic*, that is, their content created on demand.

Objects can be *opened* for reading and updating; new files can be created by opening them with appropriate parameters. An open file is identified by a file identifier, FID, which is a small integer.

A *read* operation transfers file data (identified by FID, offset and size) from the server to the client. A *replace* operation replaces a section of the file (identified by offset and size) by new data (with a given length). The *replace* operation can be used to delete slices from a file, insert data into a file, or overwrite pieces of the file. It was inspired by *Gostor* [Ionkov, 2011] and is a more versatile version of the *write* operation familiar from Unix.

Rdattr and *wrattr* are used to read or create/replace attribute values. Attributes with large values can also be opened as if they were files in their own right.

When a file is *closed*, the FID is removed. If the file had been opened for creation/updating, the close operation also commits the new version: the version becomes immutable and it becomes the new current version, visible to other users.

There are other operations, but it's probably useful to look at other components of the πp protocol suite first.

3.2. Groups

One of the problems encountered in networked file systems is the latency inherent in the protocol: opening a file, reading a chunk, discovering end-of-file, these often cost a round trip each. Latency reduction has been a primary concern for us, so extra round trips should be avoided.

This is what *groups* are all about. A number of descriptions of operations can be concatenated and sent as a unit. The descriptions of the operations have been cast into forms that allow useful combining. An *open* operation, for example, sets the resulting FID to be *current* so that a *read* or *replace* operation following it can operate on the file just opened. The *read* operation reports reaching end-of-file to eliminate a round trip just for discovering the end has been reached.

The operations in a group are executed by the server one-by-one, in sequence. If an error is encountered, further operations *in the group* are not executed and a partial reply is returned (with the error code as the last item).

A request group is sent by the client to the server and has to fit in a single packet (e.g., a UDP packet). After the server has processed the request it returns a reply group, also in a single packet. The client is responsible for requesting something that doesn't overflow the reply packet.

Request and reply groups are identified by a temporally unique *tag*. The tag is used by the client to match the reply to the original request. A client can send many requests simultaneously (but needs to use different tags for them) and the server may respond to the requests in any order. Operations that depend on each other must either be sent in one request (in the correct order) or in non-overlapping round trips.

3.3. Sessions

A session holds the shared state between client and server. This consist of four major items:

1. The authentication state: identities of communicating parties, certificates, and encryption keys for session-related traffic.
2. The connections between client and server. There may be more than one. Mobiles, that can use multiple wireless networks simultaneously, can use multiple connections for increased reliability and faster recovery from connection breakage.
3. The FID space which represents the active files
4. The *tag* space which represents the active group operations.

The session is identified by a *Session Identifier* or SID. Each group message (and the control messages the protocol uses for making communication reliable) is preceded by the SID. Everything else in the message may be encrypted with the session key.

Request and reply groups can be used in unreliable networks such as UDP. The session manager adds reliability as required by control messages, timeout and retransmission.

When connections break and new connections become established, session control messages can be used to inform the peer of changed addresses or changed connections. A name service, not discussed here, serves as a backup for existing session participants to find relocated peers and, of course, for initiating sessions.

4. Caching

Caches are essential components in content distribution systems. ΠP describes how clients communicate with caches and how caches communicate with servers, but not how clients choose caches, what caches do and how caches find servers to fetch content from.

These essential details will be discussed in a separate paper, but a few remarks are in order here.

A first thing to observe is that both caches and routers are in a position of trust: they can potentially modify or steal content as it passes through them. In networks, stealing and tampering are prevented by encrypting content. Network connections are invariably point to point, so encryption is easily arranged end-to-end.[3] This doesn't work well for caches (although it can work under appropriate circumstances) because the content-distribution function of the cache cannot be exercised if the content arrives encrypted for just one particular consumer. Alternatively, key-distribution techniques can be used to allow multiple clients to consume and understand the same encrypted content. Although nothing in our design precludes such tactics, the general assumption is that a cache must be trusted by client or server (or both). A cache that is not trusted by either, cannot be trusted not to give away confidential data, nor can it be trusted not to tamper with the data.

If content is public (i.e., giving it away doesn't matter), it can be protected from tampering by adding a *signature* attribute to the file. Clients can use the signature to verify the authenticity of the data and caches cannot forge signatures.

Again, the general model assumes a degree of trust. At the start of a session between client and server (and this also goes for client and cache, cache and cache, or cache and server), client and server run an authentication protocol of which the intended outcome is that the client knows the identity of the server it has connected to and vice versa.

If the server is a cache, it is likely that the cache will have to communicate to the server on behalf of the client. The cache cannot authenticate to the server as the client, but it can show proof that is has been authorized by the client. In the run of the authentication protocol, the client can provide to the cache a certificate, signed by the client that authorizes the cache to act (with limited power) on behalf of the client. A cache trusted by a server may similarly be able to show to the client a certificate signed by the server.

[3] But only if you know what you're doing. Roger Needham [1993] once observed: "I do not know to whom should be credited the important truth 'Whenever anyone says that a problem is easily solved by cryptography, it shows that he doesn't understand it.' "

Given satisfactory authentication, many clients can fetch data from a single cache and the cache will not fetch data more than once from the server. Since the cache is trusted by the clients (or the server), it can also be trusted to give the information it caches only to clients on the access control list for that information (file).

Since files are versioned, any version is immutable and safe to cache. Protocol is being designed to allow caches to cache information about *which version* is the *current* one (information that is *not* immutable). This will speed up open.

Caches will also be involved in real-time content distribution, something that could also be referred to as *multicast*. The protocols are designed to be capable of fetching dat with an absolute minimal delay. Caches, moreover, can do read-ahead and this make sure responsiveness is optimal.

A single client cache can be used to access servers all over the network. To direct a cache (or a client) to the correct server, path name prefixes are used: Each client (and, for purposes of talking to a server, a cache is a client too) has prefix table with entries of the form { prefix, server }. A prefix is the initial portion of a file name: cs.bell-labs.com/osprey is a prefix of cs.bell-labs.com/osprey/doc/piepea.pdf. The server in the prefix table is the name of a server or service (and this could point to another cache). The name allows the client to connect (or reconnect) to the server or service. To this end, a name is looked up in a *name server* that provides the address to use. The indirection here is important for supporting mobility.

More than one prefix could match a give file name (e.g., cs.bell-labs.com/osprey and cs.bell-labs.com in the example above). The match that is used is always the *longest prefix*.

5. Conclusions

ΠP is a practical protocol for accessing and manipulating named object. Named objects can be files in a regular file system, web pages on a web server, microphones and headsets in telephone conversations, or even bank accounts, flight status information, &c. ΠP's objects resemble files in the way they are accessed, but they don't have to be files; *read* and *write*, just like *receive* and *send* are just primitives to get information to where it's needed. The interpretation of the information is left to the applications and protocols using πp.

The sessions in πp allow an association between a client and a server to survive temporary communication outages and network-address changes.

The grouping of operations allows very low-latency interactive communication. If HTTPS were used to fetch a (small) web page, there would be quite a few packet round trips before the actual data could be transmitted: TCP SYN/ACK, TSL authentication handshake, HTTP GET request and reply. Using πp, the exchange could look like this: The client sends a group with the following operations:

plaintext	*encrypted*	*comment*
Tsession		Client provides process name,user identity, and FID of authentication (meta)file
Twrite		Client writes authentication file with proof of identity and encrypts a client→server key with server's public key
	Tread	Client requests server's proof of identity (and encrypted server→client key)
	Topen	Client identifies web page and opens it
	Twrattr	Client writes an attribute of the file to provide language and other preferences
	Tread	Request to read the web page (automatic close if EOF is reached)

The server would react with a similar group containing Rsession (with server's identity), Rwrite (which acknowledges client's authenticity), Rread (with server's proof of ID), Ropen (confirming existence of web page), Rwrattr and finally Rread with the Web page's content.

In a single packet round trip, πp achieves what requires at least three round trips in HTTP. The example also serves to illustrate that the file model does not stand in the way of realizing operations that, at first glance, have nothing to do with actual files (but Unix users are used to that in, for

example, the /proc directory).

The protocol has been designed to work with real-time media streams. It allows a client to initiate downloading a media stream by sending what amounts to a sliding window's worth of requests for consecutive chunks of the media file. The server responds to each request when the data becomes available and, as the replies trickle in, the client replenishes the supply of outstanding requests.

If the request or reply for a chunk is lost, the client can choose to request it again (if there is time) or to skip over it by pretending it was never requested (or pretending it has arrived, which amounts to the same thing). The latency between capturing and rendering can thus be precisely the unidirectional latency of packets from where they are produced to where they are consumed (with a safety margin to compensate for jitter). This is the best that can be done.

ΠP allows a single protocol running over UDP/IP (or over bare metal) to be flexibly used for a wide variety of applications. The caching hierarchy is intended to facilitate content distribution. The design of the protocol itself is designed to give absolutely minimal latency and to allow the protocol to be used in a much broader context than merely content distribution.

6. References

[Cheriton, 1988]
D. Cheriton, “The V Distributed System", *Communications of the ACM*, **31**(3), March 1988, pp. 314-333
J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, B.Weihl, “Globally DIstributed Content Delivery” *IEEE Internet Computing*, **6**(5), pp. 50−58 Sep/Oct 2002

[Ionkov, 2011]
Latchesar Ionkov “Gostor: Storage beyond POSIX”, *Proc of the 6th Intl. Workshop on Plan 9*, Madrid, 2011

[Jacobson et al., 2009]
V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard “Networking Named Content”, *CoNEXT 2009*, Rome, December, 2009

[Kangasharjua et al., 2001]
J. Kangasharjua, K.W. Rossa, J.W. Roberts “Performance evaluation of redirection schemes in content distribution networks”, Computer Communications, **24**(2), February 2001, pp. 207−214

[Mullender et al., 1990]
S.J. Mullender, G. van˜Rossum, A.S. Tanenbaum, R. van˜Renesse and J.M. van˜Staveren, “Amoeba — A Distributed Operating System for the 1990s,” *IEEE Computer Magazine*, ,**23**(5) May 1990,

[Needham, 1993]
R.M. Needham, “Cryptography and Secure Channels”, Chapter 20, *Distributed Systems*, 2nd edition, Sape J. Mullender (ed.), Addison Wesley, 1993

[Pike et al., 1995]
R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, P. Winterbottom, “Plan 9 from Bell Labs”, *Computing Systems*, **8**(3), Summer 1995, pp. 221—254

A Performance Comparison of Cryptographic Hashes and Ciphers under Plan 9 and Linux

Franck Franck
Bell Laboratories, Alcatel-Lucent, Dublin, Ireland.
franck.franck@alcatel-lucent.com

December 20, 2011

Abstract

In this paper we present a comparison of the throughput of a selection of popular cryptographic functions in two cryptographic libraries. We evaluate Crypto++ under Linux and libsec under Plan 9 to understand how the libsec functions perform compared to the current state-of-the-art.
We show that Crypto++ is a more mature implementation than libsec in terms of performance. It consistently outperforms libsec in both cryptographic hash and cryptographic cipher benchmarks. We also evaluate how much impact Intel's new hardware AES extensions has on performance of the AES cipher, and show speed gains of 3x to 9x, depending on cipher mode.
We then perform a more detailed forensic analysis of the AES cipher in both libraries in order to determine the cause of libsec's performance deficit. The conclusion is that the `8c` compiler likely causes much of the slowdown observed under Plan 9.

Keywords: linux, plan9, cryptography, performance, throughput, hashing, crypto++, libsec, aes, sha, rsa

1 Motivation

Cryptography is an important part of computer security today. With data networks becoming an ever-more critical part of everyone's lives, both in the work environment, and in our personal lives, the ability to protect and authenticate data becomes more important, and this is what cryptography provides.
Data cryptography is nothing new – in fact, it has been around since ancient times, and dates back almost 4000 years in one form or another[9], but it has evolved over time, and in recent decades has had to be adapted to the age of ubiquitous high-speed computing.
In this paper, we investigate the state of the art of cryptographic cipher and hashing performance to gain insights into the workload required to operate each cipher and how this compares to current computer hardware, both in terms of processing speed and network speeds. We do not discuss or compare the cryptographic primitives in terms of the security they provide, as that is well covered in security literature[3].
The aim of this paper is to investigate and compare performance in terms of throughput for a selection of cryptographic hashes and ciphers. This will provide some groundwork for Bell Labs' research projects within HPC, and high-speed secure networking for cloud-based operating systems.

1.1 Comparison Systems

As the work presented in this paper is part of Bell Labs' OSprey project, which has it roots in the research operating system *Plan 9 from Bell Labs*, we are particularly interested in how the current implementations of cryptographic primitives in Plan 9 hold up against what can be considered state-of-the-art implementations. For this purpose, we have

chosen to use the Linux operating system with the open-source *Crypto++* library[2] as state-of-the-art comparison base. Crypto++ is a library of a wide range of cryptographic primitives implemented in C++, and is available on a variety of operating system platforms (with the notable exception of Plan 9). Table 1 lists the benchmarking setups we have

System	CPU	Platform	AES
Plan 9	E5630, 2.53GHz	libsec	no
Linux 1	E5630, 2.53GHz	Crypto++ 5.6.0	no
Linux 2	X5650, 2.67GHz	Crypto++ 5.6.0	no
Linux 3	X5650, 2.67GHz	Crypto++ 5.6.1	yes

Table 1: **Test Platforms.** We have used two different hardware configurations: An Intel Xeon E5630-based, which does not have the AES extensions, and an Intel Xeon X5650-based which does. Version 5.6.1 of the Crypto++ library is optimized to take advantage of these extensions, whereas version 5.6.0 is not.

used in this paper. For the performance comparison of Linux vs. Plan 9, we have a system based on a quad-core Intel Xeon E5630 CPU, which does not support the proprietary Intel AES extensions that have been added to newer chips to facilitate faster operation of the AES cipher. To evaluate the potential of these extensions, we use a second test system, based on an Intel Xeon X5650 CPU. We use two different versions of the Crypto++ library: One that supports the AES extensions and one that does not.

2 Performance Evaluation

In cryptography, the most frequently used types of functions can be divided into two categories:

- *Cryptographic Hashes* are non-invertible functions used to produce a fixed-length "fingerprint" that is unique to a given input.[1] Cryptographic hashes are used for collision checks, for digital fingerprinting, and for digitally signing content.
- *Cryptographic Ciphers* are functions used to conceal or camouflage content by changing it in some way defined by a *cryptographic key.* Only systems in possession of the appropriate decryption key may change encrypted content back into its clear-text representation.

We want to evaluate the performance of both types of functions here, as they are often used in conjunction in cryptographic software, and thus both have an impact on the achievable throughput.

2.1 Cryptographic Hashes

As we want to investigate the potential throughput we can hope to obtain in a cryptographic application, we take a broad view at the hashing algorithms available. We have selected nine different hash functions for our comparison:
Three hashes – *MD4*, *MD5*, and *SHA1* – have been deemed broken in a cryptographic sense[11][7][4] and thus not suitable for use in a cryptographic context, however, we feel that they may still serve their purpose if, for example, the operating environment is knows now to be hostile.
The remaining six – *SHA2-256*, *SHA2-512*, *RIPEMD-160*, *RIPEMD-320*, *Whirlpool*, and *Tiger* – have not yet suffered cryptanalytic attacks serious enough to constitute a compromise of the algorithms, and are thus considered *cryptographically secure.* Figure 1 shows the performance of

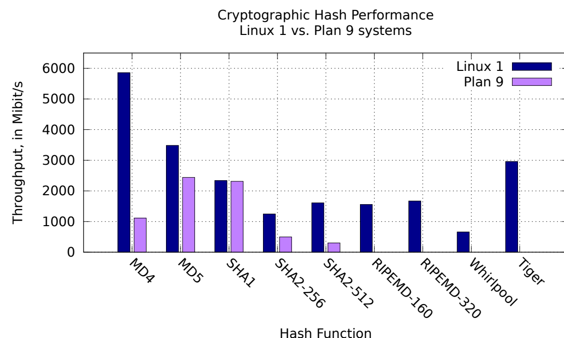


Figure 1: **Cryptographic Hash Functions Comparison.** Throughput of selected hash functions when hashing a contiguous block of 256 MB of random data. In general, Crypto++ outperforms libsec by a significant margin on the hash function benchmarks.

all nine hash functions from Crypto++ compared with the five available in libsec. From this figure, we can clearly see that the legacy hash functions

[1]Obviously, it cannot be truly unique, as it converts any-length input into a fixed-length output, but a good cryptographic hashing function should produce an output that is long enough to make collisions exceedingly rare.

have a clear performance advantage over their newer, cryptographically secure siblings. When comparing the cryptographically secure hashes, the *Tiger* hash clearly performs better than the rest of the pack. This is partly due to its optimization towards 64-bit architectures[1] (all tests were carried out in 64-bit environments), but may also be due to the fact that it provides the shortest of the secure hashes at 192 bits.

The other lesson to take home from Figure 1 is that the Crypto++ library functions seem in most cases to be far better optimized than their libsec equivalents. Only the SHA1 hash is on par between the two libraries, and at the extreme cases, Crypto++ provides almost a 6x performance advantage with SHA-512, and even more with the MD4 hash.

2.1.1 Conclusion

What we are really interested in here, is the attainable throughput we can expect when we use a hash function on some data. As might be expected, the answer depends highly on the strength of the hash function and the desired length of the hashed output.

If the usage scenario is a friendly environment, and the purpose of the hashing is merely to distinguish blocks of content or protect against transmission errors, the MD4 hash provides a formidable throughput, topping out at more than 6 Gibit/s on our test system. Its 128-bit hash length is sufficient to relegate chances of accidental collisions into the realm of the unimaginable, especially when compared to other data verification metric, such as the 32-bit Ethernet CRC[10].

If there is a need for a cryptographically secure hash, one needs to consider the hash length before choosing an algorithm; if Tiger's 192 bits is sufficient, its performance should make it the clear choice at more than 3 Gibit/s. If a longer hash is required, the SHA-2 family (SHA-1 is no longer considered secure) provide performance in the 2 Gibit/s-range, even at very large hash lengths.

2.2 Cryptographic Ciphers

The cryptographic hashing algorithms we have examined in Section 2.1 can provide protection against undetected changes to data, whether intentional or accidental. If we want to *hide*, or conceal, the data, however, we need to *encrypt* it with a *cryptographic cipher*. A cipher is an algorithm that describes how, given an *encryption key*, to perform a sequence of operation on a given piece of data such that it becomes obfuscated and the original data cannot be recovered without knowledge of the cipher and the correct *decryption key*. For *symmetric ciphers*, the encryption key and decryption key are identical, whereas *asymmetric ciphers* use two different keys.

For this comparison, we have chosen to evaluate only the most widely-used ciphers, even though there are quite a few rarely-used ciphers to choose from. The restriction has been applied in the interest of brevity of this paper. The ciphers we have chosen are: Three different AES ciphers (128 and 256-bit CBC-mode, and 256-bit CTR-mode), two Blowfish ciphers (128 and 256-bit), Triple-DES, as well as a 2048-bit RSA cipher to represent the asymmetric camp.

2.2.1 Crypto++ vs. libsec

The first comparison we perform, is a benchmarking of the chosen ciphers on our two cryptographic libraries, Crypto++ and libsec. Figure 2 shows the throughput of each of the ciphers on two of our test platforms. Recall that the Linux 1 and Plan 9 systems run on identical hardware, and therefore this is a head-to-head comparison of the two libraries. As we can see, Crypto++ provides significant performance benefits in nearly all the benchmarks. The two CBC-mode AES ciphers achieve roughly twice the throughput under Crypto++ as compared to libsec. AES in CTR-mode provides a whole difference world of performance on the two systems; on Plan 9 it performs nowhere near as well as its CBC-mode brothers, while the Crypto++-edition manages to out-perform the CBC-mode cipher with equal key size.

When it comes to the Blowfish ciphers, the performance advantage Crypto++ enjoys shrinks, but it is still a noticeable 40 percent over libsec. Another interesting feature of the Blowfish ciphers, is that the performance of the 128-bit and the 256-variants is exactly the same, so key size has no impact on the throughput of Blowfish.

Although, arguably, Triple-DES is the one of the

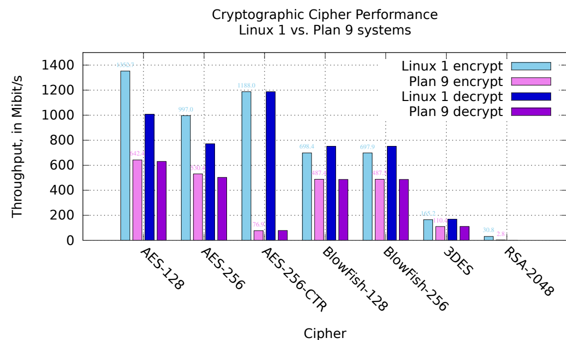


Figure 2: **Cryptographic Cipher Performance on Linux 1 and Plan 9.** Comparison of average throughput of seven ciphers. Each cipher was timed while en/decrypting a 256 MB contiguous block of random data. Generally, Crypto++ shows a clear performance advantage over libsec. The asymmetric RSA cipher is expectedly much slower than its symmetric competitors.

ciphers that provides the weakest security against cryptanalytic attacks, it also distinguishes itself by being the slowest of the symmetric ciphers (not including the libsec's AES in CTR-mode).
For the sake of throughput comparison, we have included the RSA asymmetric cipher with a 2048-bit key. The graph clearly demonstrates that the performance of the asymmetric cipher is orders of magnitude lower than the symmetric ones. Using an asymmetric cipher to encrypt large blocks of data, like we have done here, is neither the intended nor the usual practice. RSA encryption is usually used in conjunction with one of the symmetric ciphers. However, it is useful to know the tipping point at which it will be faster to use the combination of symmetric and asymmetric encryption rather than solely asymmetric encryption.

2.2.2 Intel AES Hardware Evaluation

In 2010, Intel introduced AES hardware extensions to their line of CPUs codenamed "Westmere". The purpose of these extensions is to accelerate execution of encryption and decryption using the AES (Rijndael) cipher, as well as providing security benefits over pure-software implementations.[6] Our test setup includes one machine outfitted with a Westmere-based CPU, and we use it in the Linux 2 and Linux 3 systems. We want to examine what sort of benefits can be had from utilizing these extensions.
Table 2 shows a listing of the maximum theoretical throughput attainable per core on our Xeon® X5650-based Linux 3 system.[2] As we can see in

Mode	Size	Operation	C/b	Throughput
CBC	128	Encrypt	4.15	5.49 Gibit/s
		Decrypt	1.30	17.54 Gibit/s
	256	Encrypt	5.65	4.03 Gibit/s
		Decrypt	1.78	12.80 Gibit/s
CTR	256	Encrypt	1.88	12.12 Gibit/s
		Decrypt	1.88	12.12 Gibit/s

Table 2: **Theoretical AES performance.** Maximum theoretical throughput for each of our chosen AES ciphers when executed on Linux 3. The C/b column is the processor's reference Cycles/byte during hardware-assisted AES operation. C/b and Throughput numbers are reported per-core of our Xeon® X5650-based system.[6]

Table 2, our Linux 3 system has the potential to deliver CBC-mode encryption speeds of more than 4 Gibit/s and decryption speeds of more than three times that number.
It is often the case, however, that theoretical number and real-world performance are quite different, and we want to investigate the throughput achieved by current state-of-the-art implementations. To this end, we compare the performance of two recent versions of the Crypto++ library: Version 5.6.0, which does not utilize the AES extensions, and version 5.6.1, which does. Before testing, we have performed a baseline analysis of the two versions on our Intel® Xeon® E5630-based Linux 1 system, which does not feature the AES hardware extensions. The AES ciphers' performance on the two Crypto++ versions were as good as identical, and any differences well within the margin of reporting error. We thus assume that the only change affecting performance in the AES algorithms between version 5.6.0 and 5.6.1 of Crypto++ is the support for hardware extensions.[3] Figure 3 clearly shows the impact that hardware execution of the AES cipher has on performance. Both 128-bit and 256-bit

[2]The Intel® Xeon® X5650 is a 2.67 GHz CPU with 6 cores. However, when the power- and temperature envelope of the processors allows, it will increase frequency up to 3.06 GHz, which is what we have based our calculations on. Throughput per core under high load will this be lower

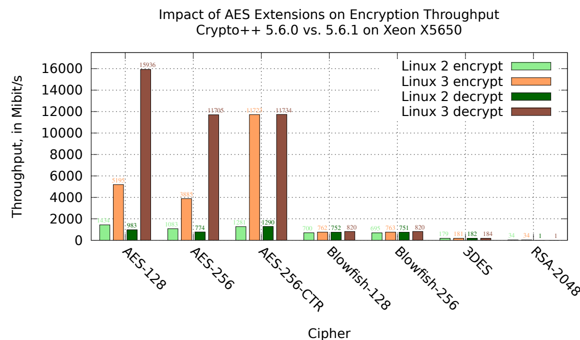


Figure 3: **Impact of AES Extensions on Encryption.** Average throughput of seven ciphers with vs. without hardware AES extensions. Benchmarks are performed on a contiguous block of 256 MB of random data. Intel's newer CPU architectures provide a sizable performance improvement to ciphers that can take advantage of them. Particularly the AES cipher in CTR-mode enjoys a massive performance boost.

CBC-mode ciphers show 3x to 3.5x throughput increase, while the performance of 256-bit AES in CTR-mode is increased by a massive 9x. As expected, there is no visible effect to any of the other ciphers in the library.

When we compare the theoretical throughput numbers in Table 2 with our real-world experiments, Crypto++ attains an impressive 94 percent of the theoretical throughput on 256-bit AES encryption in CBC-mode (3.79 Gibit/s realized vs. 4.03 Gibit/s theoretical). Looking at the numbers for our CTR-mode cipher, we see another near-perfect score of 94 percent of the theoretical maximum (11.45 Gibit/s realized vs. 12.12 Gibit/s theoretical).

These numbers show that the Crypto++ library is quite mature and well-implemented in terms of throughput performance, and leaves little room for improvement in the AES department.

2.2.3 Competing Hardware Encryption

2.2.4 Cipher Initialization

All of the symmetric ciphers we are studying here require some initialization work to be done before the actual encryption/decryption can take place. This initialization step makes sure that the P-boxes, S-boxes, subkeys, Initialization Vector, and any other state information needed for the functions of the cipher is present.

The real-world overhead this initialization incurs is highly dependent on how a given piece of software uses its ciphers, but it can be quite high if an unsuitable cipher has been selected for a particular workload. Figure 4 shows a comparison of symmetric cipher initialization performance across all our test systems. Due to differences in how the two cryptographic libraries initialize the ciphers, the benchmark shows is a measurement of how many times per microsecond each of them can instantiate a new cipher and encrypt 256 bits of data. Results are mixed, and the Plan 9 system takes wins in both the AES CBC-mode ciphers and in particular the Triple-DES test (note the logarithmic y-axis on the graph), even despite competing against a faster CPU on Linux 2 and Linux 3. We clearly see how the initialization process of the Blowfish cipher is much more compute-intensive than any of the other ciphers. This is by design[8], but is it definitely an aspect worth taking into consideration when selecting cryptographic ciphers for a particular application. It may worsen the performance of the Blowfish cipher in applications requiring frequent re-initializations.

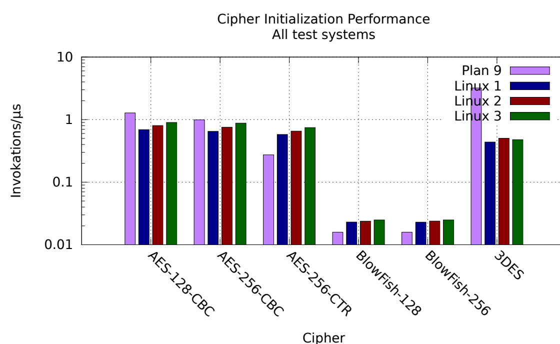


Figure 4: **Symmetric Cipher Initialization Performance.** Comparison of how fast cryptographic ciphers can be initialized and invoked on a 256 bit data block across all four systems. Plan 9 generally compares favourably with the Linux-based systems. Blowfish initialization is well over an order of magnitude slower than the other symmetric ciphers.

than the reported numbers.

[3]The Crypto++ 5.6.1 release notes state that the changes to the AES functions from 5.6.0 is the addition of AES and CLMUL support, as well as a number of bug fixes.[2]

The availability of AES extensions impacts performance only slightly. Generally, AES initialization in Crypto++ 5.6.1 is faster than in 5.6.0 due to the fact that the benchmark actually does perform encryption on 256 bits of data.

2.3 Linux vs. Plan 9

When examining the performance of our two encryption libraries in Section 2.2.1, Figure 2 showed that there is quite a large difference in throughput between the two. Particularly the AES family of ciphers enjoyed a vastly superior performance under Crypto++, and thus Linux, than under libsec, which is Plan 9-based.
We are interested in investigating further into the causes of this difference in order to provide a path toward improving libsec and bringing it on par with state-of-the-art implementations. To this end, we will take a closer look at the constituent parts of a cryptographic cipher and perform an evaluation of how Linux 1 and Plan 9 compare to each other in each of them. As before, we will mainly focus on the AES cipher implementations in the two libraries, as we expect these to be the most wide used ciphers. However, the basic operations are similar in most of the symmetric ciphers, and conclusions will thus likely benefit the implementation of all of these in libsec.

2.3.1 Deconstructing AES

To gain more insight into the performance of the AES implementations, we will briefly examine how the cipher works in order to highlight which operations it will be beneficial to investigate.
Each round of the AES cipher consists of four distinct steps: `SubBytes()`, `ShiftRows()`, `MixColumns()`[4], and `AddRoundKey()`.[5] We will take a look at each of these to examine what the underlying operations they perform are:

- `SubBytes()` simply substitutes each of the bytes in the cipher's state with a value from AES's precalculated S-box. The step thus performs two memory access operations.
- `ShiftRows()` transforms three of the four rows in the cipher state by left-rolling them one, two, or three words. Depending on the architecture of the execution environment, the exact number of instructions needed to perform this transformation varies, but they are all simple bit-shift operations.
- `MixColumns()` diffuses the information state by applying a finite-field polynomial transformation to each of its four columns. Each of these transformations consists of two finite-field multiplications and three XOR operations. The sum total is that this step performs integer multiplications, integer divisions, XOR operations, and some memory access for each byte in the state.
- `AddRoundKey()` uses a precalculated key expansion table to lookup a subkey for the current round. This subkey is then applied to the state by XORing it with each of the state columns. This step thus performs memory access and bit-wise XOR.

Having taken a closer look at the constituent steps of the AES cipher allows us to categorize the operations needed into five categories: *memory access*, *integer multiplication*, *integer division*, *bit-wise shift*, and *bit-wise XOR*. These categories are our vehicle to perform a closer profiling of our Linux and Plan 9 operating environments to examine what causes the performance difference between the two.

2.3.2 Detailed AES Benchmarks

For each of the AES operations we identified in Section 2.3.1, we perform simple benchmarks designed to stress only that particular operation. The benchmarks are implemented in portable C, and are designed to compile directly to assembly language on the target platforms. We have taken care not to invoke system calls or library routines of the operating system. The purpose is to get a picture of exactly what might be holding back the performance of the AES ciphers on Plan 9, and thus we essentially stress test the memory access throughput and the efficiency of the compilers. As mentioned, the hardware on the two test platforms is identical. Table 3 shows an overview of the exact

[4]The last round of the AES cipher does, in fact, not have the `MixColumns()` step, but for the sake of simplicity, we disregard that here.

System	OS version	Compiler
Plan 9	21-jan-2011	8c (4-jan-2011)
Linux 1	Kernel 2.6.38	gcc v. 4.6.1

Table 3: **AES Benchmark Environment.** For the detailed AES benchmarks, we delve deeper into the code behind the cipher. For reproducibility, we list the versions of the tools used for our comparisons here.

operating environment we have used for the benchmarks.

We have designed seven different micro benchmarks to test our operations: `NOP Loop` is an empty for-loop, `Seq. m/a` performs sequential memory access by reading a block of memory from one end to the other, `Non-seq. m/a` reads a memory block from both ends toward the middle, `Int. mult.` performs integer multiplication, `Int. div.` does integer division, `Bitshift` performs bit-wise left-shift operations, and finally, `Bitwise XOR` invokes the XOR operator. All tests are performed on a contiguous 32 MB memory segment, run 100 times, and the results are averaged. Figure 5 shows the

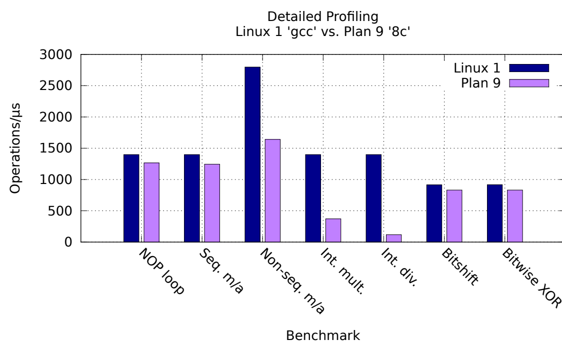


Figure 5: **Detailed AES Benchmarks.** Seven micro-benchmarks have been performed on Linux 1 and Plan 9. The benchmarks focus on compiler operation, memory access, integer arithmetic, and bit-wise operations.

results of the benchmark operations, and it is evident that for the most part, the results between the two systems are quite similar (as would be expected). Three operations stand out, however, giving Linux 1 an advantage: `Non-seq. m/a`, `Int. mult.`, and `Int. div.`. Notably, these are all micro benchmarks that include some form of arithmetic to be done in every invocation of the loop (`Non-seq. m/a` calculates an additional memory offset compared to the other tests), while the tests doing more direct manipulation of system memory exhibit very similar scores on the two systems. In other words, memory access does not seem to be hindered by the Plan 9 system, but somehow arithmetic operations seem slower than on Linux 1.

Arithmetic operations use CPU registers to store and work on the operands of the operation. How CPU registers are allocated and accessed during program execution can be very compiler-dependent, and in order to determine whether Plan 9's `8c` compiler uses a less efficient registerisation strategy than Linux 1's `gcc` compiler, we need to perform another test. Both `8c` and `gcc` will by de-

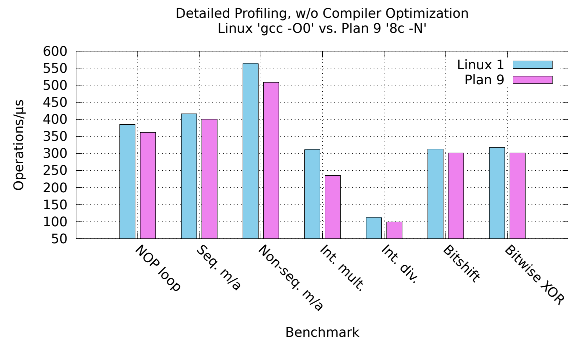


Figure 6: **Compiler Optimization Comparison.** By disabling compiler optimizations, we obtain much more even results for the Linux 1- and Plan 9 systems in our micro benchmark. Thus, the cause of Plan 9's performance deficit seems to lie with the `8c` compiler's code optimizer.

fault perform some code optimization on the code they generate, so in order to determine whether this optimization is the cause of the performance difference, we repeat the micro benchmarks with code where no optimization has been performed.

Figure 6 shows the non-optimized benchmarks, and the difference to the default code is striking. The systems are virtually on par, with Plan 9 lagging by a few percent in most tests. These results clearly lead to the conclusion that `8c`'s optimizer is less efficient than `gcc`'s, and thus that much, if not all, of the performance difference we saw in Sections 2.1 and 2.2.1 can likely be reclaimed by a more efficient code optimizer in `8c`.

3 Conclusion

In this paper, we have investigated and benchmarked the performance of a range of modern cryptographic ciphers as implemented in two cryptographic libraries: Crypto++ under Linux and libsec under Plan9 from Bell Labs.

Our goal has been to provide a reference to what the cryptographic performance of a state-of-the-art library can be expected to be.

Our findings show Crypto++ to be a very mature and high-performing library in terms of throughput of its cryptographic functions. Especially when paired with Intel's AES hardware extensions, Crypto++ was able to achieve double-digit gigabits per second of throughput, and scored around 95 percent of the maximum theoretical throughput on our test hardware.

libsec, on the other hand, is not only a less feature-rich library, it also leaves a wide performance gap up to the competition in most benchmarks. Only the process of initializing ciphers proved to be faster with libsec than Crypto++, so clearly some work needs to be done to bring the Plan 9 library up to standard.

Support for Intel's AES hardware extensions provides Crypto++ with a 4x to 9x performance increase over its own software-only implementations, so this seems to be where the low-hanging fruits may be for libsec improvement, at least for the AES ciphers.

Other improvements seem to available by improving the `8c` compiler we used for out tests on Plan 9. Our in-depth analysis of the AES performance seems to indicate that the optimizer in particular is sub-par compared to its Linux counterpart.

References

[1] Ross Anderson and Eli Biham. Tiger: A fast new hash function. In *Fast Software Encryption, Third International Workshop Proceedings*, pages 89–97. Springer-Verlag, 1996.

[2] Wei Dai. Crypto++ library 5.6.1 - a free c++ class library of cryptographic schemes:. `http://www.cryptopp.com`, December 2011.

[3] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. Wiley, 1st edition, March 2003.

[4] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2003.

[5] FIPS. *Advanced Encryption Standard (AES)*. National Institute for Standards and Technology, Gaithersburg, MD 20899-8900, USA, November 2001.

[6] Shay Gueron. Intel advanced encryption standard (aes) instructions set (white paper). 2010.

[7] M.J.B. Robshaw. On recent results for md2, md4 and md5. *RSA Laboratories' Bulletin*, (4), nov 1996.

[8] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In Ross Anderson, editor, *Fast Software Encryption*, volume 809 of *Lecture Notes in Computer Science*, pages 191–204. Springer Berlin / Heidelberg, 1994. 10.1007/3-540-58108-1_24.

[9] Bruce Schneier. *Secrets and Lies: Digital Security in a Networked World*. Wiley, 1 edition, January 2004.

[10] Andrew S. Tanenbaum. *Computer networks (3rd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[11] Xiaoyun Wang, Yiqun Yin, and Hongbo Yu. Finding collisions in the full sha-1. In Victor Shoup, editor, *Advances in Cryptology CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 17–36. Springer Berlin / Heidelberg, 2005. 10.1007/11535218_2.

A NIX Terminal

Erik Quanstrom
quanstro@quanstro.net

ABSTRACT

Starting just after the initial announcement of NIX in September, 2011[1], a project was started to replace the Plan 9 32-bit port completely with a 64-bit port based on NIX. Replacing cpu servers was relatively easy; most of the work was already done. Replacing terminals was quite a bit more involved. There was more chore than innovation. But it required enhancements to physical address mapping. Using the VESA interface required a way to execute or emulate BIOS calls. In addition, many new devices were added to support terminal hardware. A few subtile but significant bugs were encountered. The result is a Plan 9 terminal kernel that runs on real hardware and under VMWare Fusion.

Introduction

In September, 2011, NIX was announced on the 9 fans mailing list. By this time, many of the accomidations for ancient PC hardware such as 8259 interrupts, the 8254 timer, were weighing quite a bit on the 32-bit Intel kernel. The kernel also suffered from type confusion; `ulong` was used for virtual addresses, physical addresses, 32-bit registers and marshaled values. Padding appropriate for 32-bit machines was assumed. A large number of ancient devices were being dragged around. There were a total of 99,524 lines of code in the `pc` directory. Unfortunately, some of this old code is useful for embedded or old systems, so simply discarding it seemed unwise.

The approach taken was to try to duplicate *all* currently-used functionality in the 64-bit kernel. The number of 32-bit systems is fading rapidly. It seems feasible to run on only 64-bit hardware most of the time, so using 64-bits as an excuse to drop old code seemed like a viable strategy.

The announced NIX kernel while suitable as a cpu kernel, was missing a number of required features for terminals. Real mode emulation was missing, as were the proper sort of virtual memory mappings. The VGA and draw infrastructure were missing, as were support for usb and for terminal keyboards and mice. These have all been addressed, and this paper was composed on two native NIX terminals. Additionally, some work was put into the networking subsystem to allow booting of terminals under VMWare Fusion.

Address Map and Virtual Mapping

Most video hardware handled by plan 9 works by mapping a physical frame buffer into virtual address space. The frame buffer is mapped after the machine has been otherwise fully initialized, and all cores have been started. Draw operations read and write

from this frame buffer. To speed drawing, the caching subsystem is often instructed to relax coherency or turning write-combining on for this buffer via (deprecated) MTRR registers or page-table attributes (PAT).

At the start of this project, none of these requirements were met by the NIX kernel. Virtual memory mapping via *vmap* could only be done on the boot processor early in startup, and there was no way to modify the the cacheability of a memory range from the default. The reason for both limitations was similar. There was no facility for remembering the mapping. So the architectural requirement that all mappings of the same physical range have the same cacheability could not be met. Likewise, faults on a *vmap*'d range could not be resolved on other processors. *Vmap* works without tracking early in boot because the boot processor's (BSP) page tables are copied for each additional processor (AP).

To track memory cachability, it was decided to create new type of physical memory map called *adr*(9nix). The map is low duty-cycle for entry manipulations, but high duty-cycle for lookups so a simple insertion sorting scheme was used. Map entries consist of a base, length address type (e.g. memory, mmio, etc.), current use and caching flags. The memory type is a superset of ACPI memory types. In-use entries remain in the table. A subset of the programming interface is detailed here.

```
#include "adr.h"
enum {
	Anone,
	Amemory,
	Areserved,
	Aacpireclaim,
	Aacpinvs,
	Aunusable,
	Adisable,

	Aapic,
	Apcibar,
	Ammio,
	Alast       = Ammio,
};
enum {
	Mfree,
	Mktext,
	Mkpage,
	Mupage,
	Mvmap,
	Mlast       = Mvmap,
};
void    adrmapinit(uintmem base, uintmem len, int type, int use)
void    adrfree(uintmem base, uintmem len)
uintmem adrmemtype(uintmem pa, uintmem *len, int *type, int *use)
```

The map is initialized from the ***e820** configuration variable. The kernel assumes that this is populated by the bootloader, typically by calls to BIOS INT15 function E820. After the kernel starts, ACPI, PCI and other memory-mapped facilities such as I/O APICs and LAPICs add to this table. Address ranges may be allocated by address type; *adr* splits address ranges as necessary. Allocated maps are indicated by a non- **Mfree** value of When maps are freed, they are merged with adjacent maps of the same address type. Sub-page allocations are allowed to deal with devices that map less than one page of physical memory.

Adrmapinit enters a new physical mapping which is required to be disjoint with all other mappings. *Adralloc* allocates a map entry by subdividing an existing entry and changing its use and memory type. Only free entries may be allocated, thus preventing mapping of non-existent memory ranges, double maps, or inconsistent memory types. *Adrfree* returns the physical range to free status. Cachability of an unmapped range of memory may be changed on allocation since it is unused. *Adrmemtype* returns the page table memory type flags, use and base address for a given virtual address. The flags are stored in a form suitable for setting PAT flags for a 4KB page. The mmu code converts to large-page flags when mapping large pages.

Vmap also requires that we be able to generate a consistent virtual address for a given physical address. Since we have a full 64-bits of address space, we can simply map each physical address to a virtual address that is offset by a suitable constant, **KSEG2**. Now on initial *vmap*, *adralloc* can allocate the given range with a formulaic virtual address on the local processor. (A new function *vmappat* can specify PAT caching flags; it is used for mapping frame buffers.) Faults on other processors can be resolved by calling *adrmemtype* to find a matching bit of allocated memory and return its memory flags on a page fault.

This scheme works well. *Adr* is only 513 lines of code (less than 20% larger than its predecessor). There is still only one table of physical addresses. It is never necessary to store the virtual address since we chose the virtual address by formula. Lookups on page faults are much faster than the page fault itself. The benefit of forcing all entries to be entered before mapping, however, is up for debate. On the one had it does prevent errors, and has caught inconsistencies between MP and ACPI tables in describing APICs. But on the other hand, it is tedious and error prone. It remains because small errors in memory mapping can be hard to track down.

VESA BIOS Calls.

To have a terminal, at least one hardware interface needs to be selected. “VESA” was selected since most hardware will implement the VBE (VESA BIOS Extensions) interface[2]. VBE uses 16-bit real mode calls to set up the frame buffer and provide other services like screen blanking. The 386 kernel uses 176 lines of assembly to return to real mode, make the VBE call, and return to 32-bit protected mode. A register-based interface allows calls in from user space. A special file, `/dev/realmode` provides access to bits of low memory required to run VBE calls.

To make direct BIOS calls from a 64-bit kernel would require all that code, in addition to code to make the 64-to-32 bit and the 32-to-64 bit transitions. Emulation seemed simpler. Fortunately, *realemu*(8) already provides emulation for VBE BIOS calls via the same register-based interface[3]. So no transition to real mode was required. A expanded version of `/dev/realmode`, `/dev/resmem` provides access to low, and ACPI-reserved memory.

Since the kernel itself needs to blank the screen, it uses the context of the process setting up video to maintain a channel to the BIOS emulator. These calls are initiated from the clock interrupt, so it is not possible to make this call directly. The solution taken was to use a kernel process reading a queue to call out to the VBE emulator. The cost is that the emulator needs to remain running.

Unfortunately, the real mode calling interface does not work in 64-bit mode. It assumes the same structure padding as the 32-bit Intel compiler, and little-ending encoding. A proper solution to this problem has not yet been implemented, since it would require a

rework of the 32-bit kernel, the emulator, and *vga*(8) as well. The problem has simply been worked around with `#pragma pack`. It is worth noting that few calls are made through this interface, so a textual interface would be sufficient. Alternatively, 16-bit machine code could be passed in directly for emulation.

Porting Devdraw

There were two main challenges in porting the draw device and supporting libraries to 64-bits: the the dependence on the *pool*(2) allocator, which is not used by the 64-bit kernel; and two relatively obscure but important bugs.

The draw libraries assume the *pool* memory allocator. This allocator provides for optional memory compaction, to prevent a relatively small image memory from becoming too fragmented to allow for the allocation of large images. Removing this dependence required isolating the memory allocation in the *memdraw* library to a single file, which still depends on the *pool* library. Then a replacement for the kernel was written using the kernel standard allocator. It simply ignores compaction requests. Originally, a compacting allocator was planned, as has been the tradition since the blit[4] but it turns out that this optimization has not been necessary. No memory allocation failures have yet been observed, and memory use seems reasonable.

Unlike the 386, x86-64 cpus have enough registers to sensibly consider using the `extern register` construct again. This storage class is special. It does not have any relationship to `extern` nor to `register`. It means that one register should be allocated per processor for storing the given value. (That is, there will be one independent value per cpu.) The kernel stores a pointer to the local virtual machine `Mach *m` and the current process `Proc *up` using this storage class. The compiler allocates these from "the top," or R15, down, while regular register allocation starts with AX and works up. As the compiler manual notes[5], one must make arrangements that all code including libraries be compiled with the same external registers. For previous compilers using this technique, 28 or more usable registers were available and failure to observe this rule was harmless, since the kernel contains no code that has so many live registers. However, the drawing libraries can use all registers up through R15, which clobbered the external registers. The implemented solution was to prevent the compiler from allocating regular registers higher than R13, since this is more practical than recompiling all libraries for the kernel.

The second bug was with the following code in *byteaddr*.

```
uchar *a;

a = i->data->bdata+i->zero+sizeof(ulong)*p.y*i->width;
```

When `p.y` was small and negative, and a was a normal kernel address (greater than −256MB), a took on values just under 4GB. On careful examination, it was seen that this is a consequence of the Plan 9 compilers being unsigned preserving. Suppose `p.y` is −1 and the `i->width` is 3. Then the third term will be `0xfffffff4`. Since `sizeof(ulong)` is itself a `ulong`, the term is of type `ulong`. If we add this value to a 32-bit pointer, sign doesn't matter, since unsigned and signed addition are the same. But when adding this value to a 64-bit pointer, we zero-extend and end up with a large positive value. The solution to this bug is trivial, and obvious once found. Simply cast the term to an `int` to enable sign-extension. It is likely that there are other, similar bugs.

Additional Devices

To equal the functionality of the 32-bit kernels a few additional devices were required: support for more network cards, and usb. To a large extent, this was an exercise in pipe fitting, and replacing `ulong` with a more descriptive type, usually `u32int`. However USB devices require 32-bit buffers, and network drivers (especially for 10gbe) can consume more than the maximum 256MB of standard kernel heap. Since the kernel heap has physical addresses less than 4GB (typical physical addresses start at 1MB), the fact that USB has a 32-bit interface is not currently a problem.

The issues with networking are a little more interesting. Due to limitations of the instruction set, it is difficult to run code out of a virtual address that is not either a sign-extended 32-bit value, or simply below 4GB. Since we traditionally place the kernel at the top of memory, and traditionally place the heap above the stack, this limits kernel memory to 256MB. However, there is no requirement about data, so it would be possible to place the real kernel heap below the kernel text. The current kernel takes a hybrid approach. The kernel heap remains where it is, but network `Block*`s are allocated directly from physical memory. This means they are mapped below the kernel text at **KSEG2**. While this is somewhat less than ideal—the rest of kernel memory is quite constrained—network buffers may grow to fill most of memory. But it does show the way to having an arbitrary amount of kernel heap. Due to 32-bit devices like USB, however, exceeding 4GB of kernel heap may require careful memory tracking. And 4GB may be a practical limit for the heap.

Conclusion

Currently the 64-bit NIX kernel runs on a variety of AMD, Intel and emulated hardware as a terminal using VESA graphics through VBE calls. This kernel is fast, is able to use all installable memory, and up to 255 cores, and provides broadly equal hardware support to the 32-bit kernels. It is a complete and full-featured replacement for the 32-bit kernels, for hardware supporting 64-bits.

However, there are several areas worth attention in the near future. The scheduler tends to suffer with more than 8 cores. And due to the use of 2MB pages for user segments, user processes use too much memory. Using a virtual page size of 64KB, and abutting segments with increasing page sizes are under investigation.

Abbreviated References

[1]R. Minnich, `http://9fans.net/archive/2011/09/145`

[2]VESA, *VESA BIOS Extension (VBE)*, version 3.0, September 16, 1998.

[3]G. Friedemann (cinap_lenrek), *realemu*, `http://9fans.net/archive/2011/03/5`

[4]R. Pike, L. Guibas, and D. Ingalls, *SIGGRAPH'84 Course Notes*, May 21, 1984.

[5]R. Pike *How to Use the Plan 9 C Compiler*, `/sys/doc/comp.ps`

Access Control for the Pepys Internet-wide File-System

Tommaso Cucinotta, Nilo Redini
Bell Laboratories, Alcatel-Lucent Ireland

Gianluca Dini
University of Pisa, Italy

October 31, 2012

Abstract

This paper describes the Access Control Model realized for the novel Pepys distributed, Internet-wide, file-system. The model design has been widely inspired to various existing standards and best practices about access control and security in file-system access, but it also echoes peculiar basic principles characterizing the design of Pepys, as well as the ΠP protocol, over which Pepys itself relies. The paper also provides technical details about how the model has been realized on a Linux port of Pepys.

1 Introduction on Pepys

Pepys is an innovative distributed file-system born to meet the increasingly growing demand, from users, to always have their data available anywhere.

Pepys is composed of a multitude of servers that, together, present a collection of files organized in trees or volumes. It uses a hierarchy of caching file *servers* and a set of archival storage servers, brought together through a common set of protocols for data access and control. Moreover, in order to design a fault-tolerant system, files may be replicated among servers; doing so it is even possible to improve the speed of files fetching.

In Pepys, when a new file is created, it is not necessary that every directory present into the path is present. For example, the file named `/a/b/f` can exist in the file-system without requiring existence of `/a/b` and/or `/a`. In the Pepys design, the traditional distinction among files and folders is replaced by the ideas that a collection of files (called *objects*) reside in the file-system. The existing object having a name with the longest prefix matching the name of another object merely becomes the *guard* of said other object. For example, if `/a` and `/a/b/f` exist and `/a/b` not, then `/a` is the guard of `/a/b/f`. The guard relationship among objects ultimately regulates how exactly access control is performed, within the Pepys file-system, as it will be detailed in Section 3.

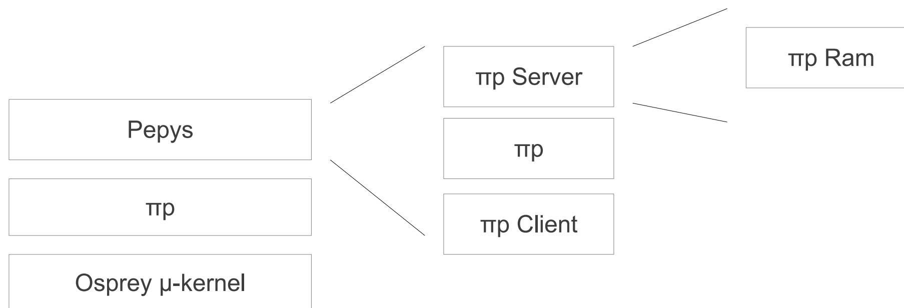


Figure 1.1: Pepys components.

Pepys is a versioned file-system, i.e., when a file is modified, a new version of the file is added to the system, that keeps storing all the previous versions. This way it is always possible to keep track of the files history. Versioning allows for an efficient caching of files.

Moreover, files in Pepys may have *attributes*. These are defined in the same name space as for the regular files. For example, if `owner` is a valid attribute for the file `/a/b/f`, then its complete name is `/a/b/f/owner`. To avoid confusion between files and attributes, a special character is used in the file operations when referring to attributes.

Furthermore, Pepys uses a new transport protocol (called ΠP) in order to minimize the round-trip message exchanges, between a client and a server, necessary to perform file transfer operations. The protocol allows to send, to the server, multiple consecutive requests in a single packet.

Being still under heavy development, Pepys has various features still under implementation, or merely at a design stage. For example, Pepys was not including any mechanism for access control, yet. This document describes the work that has been done in order to add an Access-Control Model to the Pepys distributed file-system, complying with the general principles behind the Pepys design.

Pepys file-system is currently implemented on top of a new operating system called *Osprey* [9] (see Figure 1.1), providing an alternative approach to cloud computing, and specifically aiming to improve latency and predictability of cloud applications and support for mobility. A key component in the overall architecture is the ΠP protocol, supporting all Pepys operations, including various interactions with the Osprey kernel itself.

As shown in Figure 1.1, the original Pepys server we modified included ΠP Ram, basically an in-RAM file-system. As explained in 4.1, this has been extended to keep files on a Linux (and generally POSIX) file-system, and to support our new AC model.

1.1 Paper Organization

The reminder of this paper is organized as follows. In Section 2, we put our work in relationship with related existing works in the literature. In Section 3, we present the Access-Control Model (ACM) we designed for Pepys, highlighting the most important design choices. Section 4 provides some implementation and further architectural details. Finally, in Section 5 we describe possible future work we plan to do on the topic.

2 Related Work

In order to design an efficient and state-of-the-art access control model, some of the most widely known and deployed standards for file-system access control have been considered, and specifically:

- Unix File-System permissions [11] and Linux extensions [13]
- New Technology File-System (NTFS) permissions [12]
- POSIX Access Control Lists (ACLs) [1, 5]
- Role-Based Access Control (RBAC) [10]
- Discretionary Access Control (DAC) [7]
- Mandatory Access Control (MAC) [7]
- HTTP authentication mechanism [4]

Our work was greatly inspired to the POSIX Access Control Lists (ACLs) [1, 5]. POSIX ACLs overcome some of the limitations of the old UNIX file-system [11], allowing for the definition of multiple per-user and per-group rules, providing a great liberty of flexibility in expressing access-control rules. The access-control model proposed in this paper is also based on attaching lists of access-control rules to files, therefore our model is also referred to as an ACL model, even though there are various differences with the standard POSIX ACL (see Section 3 for details).

In order to represent the set of allowed permissions for users or user groups, the classical concept of a *bit-mask* has been used, similarly to the UNIX file-system [11]. However, the set of allowed permission bits does not match perfectly UNIX. For example, we do not support the right of execution for files (that would not have sense in a distributed system); also, taking inspiration from NTFS [12], the *co-owner* bit has been added, used in ACL entries to define which users are co-owners of the file, i.e., they can manage its ACL settings.

Also, in our model the concepts of users and groups are somewhat unified, being also possible to define arbitrary nesting levels among groups of users. This behavior can be thought of as a flexible way to define users' roles and their hierarchical or nesting relationships, hence can be compared to the expressiveness often found in RBAC [10] models.

Our model design allows users to manage their own files permissions, allowing for a completely discretionary access-control, as found in DAC [7] models. At the same time, it is provided the possibility, for a system administrator (or specific set of privileges users), to define “upper-bound” rules that cannot be overcome by regular users, stealing some of the characteristics of typical MAC [7] models, and taking inspiration from similar characteristic available in in NTFS.

Our implementation did not address comprehensively *authentication*, yet. However, a basic authentication mechanism has been realized, taking inspiration from HTTP-Auth [4], used in the HTTP protocol, in which clients send their hashed password to authenticate to the server. The authentication mechanism also re-uses the “everything is a file” old paradigm of UNIX and further developed in the Plan9 OS [3]. Furthermore, we support a primitive mechanism for *delegation [6]* of authority through off-line delegation certificates resembling Amoeba *capability lists* [8, 2].

Various other access-control models for file-systems have been proposed in the literature, such as the WebOS [14] work, including a mechanism allowing entities to delegate other entities in order to act on their behalf on a set of defined file-system objects, or others. A comprehensive list of such works is out of the scope of the present paper.

3 Access Control in Pepys

One of the basic concepts behind the Pepys access-control model design is the one to create an environment in which:

- the traditional distinction between users and groups is replaced by a unified vision of such entities;
- AC rules can be specified at a generic abstraction level, considering sets of files and sets of users, then refined for specific subsets of those files and/or users;
- each user is free to define the access control rules for its own objects, in the most flexible way possible;
- however, each user freedom is constrained by the rules dictated by system administrators, if any;
- re-using the “everything is a file” approach to manage as many operations as possible, including operations involving the administration of the access-control operations, such as editing of ACL rules or creation of users.

More details on the specific aspects are reported below.

3.1 Entities

The difference between users and groups has been overcome by introducing the concept of *entities*, representing users or groups of users, that can be authorized or denied the access to portions of the file-system.

In order to make the system security administration as scalable as possible, entities (i.e., users and groups) can belong to others entities; if needed, a system can be configured in such a way that a nesting relationship becomes valid when both involved entities agree about it. An entity has to be aware of the fact that, adding another entity in the set of entities belonging to it, is equivalent to giving them all the access rights to which it is entitled, unless otherwise overriden by more specific rules.

There are no limitations for the nesting level of the belong-to relationship, which is to be considered a transitive relationship. Hence, a "belong-to" relationship between two entities can be:

1. Direct
2. Indirect (if transitively inherited).

The first kind of relationship is considered stronger than the second one, from an access-control (AC) perspective, meaning that an AC rule referring to a direct father of a user has priority over an AC rule referring to a generic ancestor. The direct and indirect ancestors of an entity can be visualized in a "belong-to" relationship priority tree in which the entity under consideration is the root of the tree (see Figure 3.1).

Moreover, as we will see, an entity authentication is not mandatory: an entity can decide whether or not to authenticate itself into the system.

Two system-level entities are always defined in the system, called `others` and `nobody`. Each entity defined in the system belongs implicitly to `others`, but only in the weakest possible sense (see Section 3.2.1). The `others` entity is a convenient way, in ACL rules, to refer to any authenticated user in the system. Also, unauthenticated entities, as well as entities just logged onto the system, and about to authenticate, are treated by the system as implicitly being the `nobody` entity. The `nobody` entity may be conveniently used in AC rules to refer to any unauthenticated user. Also, is the system implicitly considers that `others` belongs to `nobody`, as shown in Figure 3.1. The purpose of these two entities is further detailed in Section 3.3.

Finally, since nesting relationships can be arbitrarily added by users, loops are possible in the belong-to tree. Such a situation, albeit unusual, is still handled by the implementation consistently.

3.2 Access Control Model

Each object in the file-system owns an ACL table which contains the access rules governing access to it; each rule names an entity and its permissions to the object.

Each ACL can have one or more *co-owners*, which can manage the rules in the ACL. At least one co-owner has to be always present, so to ensure that there is always someone able to manage the object security settings. Therefore, the system forbids the operation of deleting the ACL rule for the last co-owner.

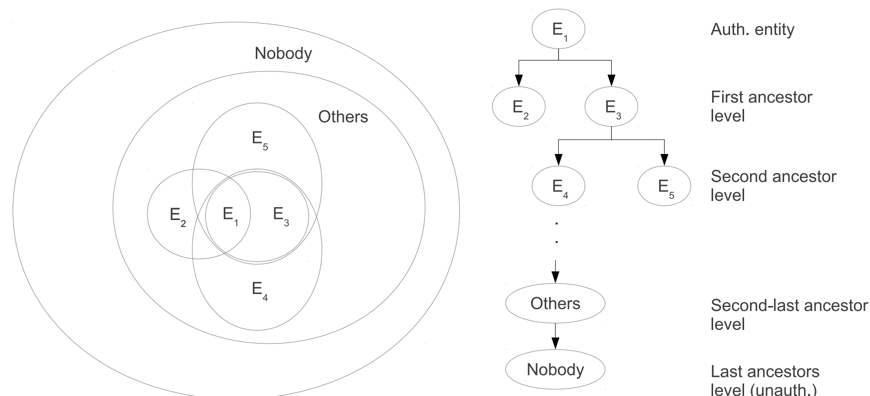


Figure 3.1: Belong-to relationship tree, rooted at a generic entity E_1.

ACL rules apply generally to the object they are attached to, but are implicitly and dynamically inherited also by all the objects having it as a guard (i.e., the children file), and any other further object down the containment/guard hierarchy of objects (i.e., the whole subtree rooted at the object). Normally, a rule attached directly to an object takes precedence over a rule attached to its guard (father), or a rule attached to its guard's guard, etc. However, there is a special type of rules, called non-overridable rules (*o-rules*), that forcibly apply to the the whole subtree of the guarded files and cannot be overcome. Such rules are designed to be used typically by system administrators to restrict the AC settings that regular users may be willing to configure for their own created contents.

As a result, a rule in an object (both a regular rule or an o-rule) stating that an entity has certain permissions is effective only if there are not any o-rules, in its guards chain or in the object itself, stating otherwise.

An ACL rule mentioning the `others` entity can be used to grant or deny access to any user known to the system, when acting as an authenticated user. Also, An ACL rule mentioning the `nobody` entity can be used instead to grant access to any user connected to the system, but not having authenticated (yet). However, authentication is only partially addressed in Pepys (e.g., server authentication is unaddressed, so far), as a full mechanism will have to be integrated with cryptography at the ΠP protocol level.

The type of supported permissions in the current design and implementation is inspired to traditional UNIX file-systems: read and write of files, traversability of guards, ACL management (co-ownership). However, this tentative set of permissions can easily be extended to more complex permissions or permission set (e.g., adding a delete permission or others, as found on NTFS file-systems). It is noteworthy to mention that, whilst on traditional file-systems, the read permission over a folder refers to the ability to read the folder contents, in Pepys it is planned to provide distinct permissions to read a guard's children (the guarded/contained objects), and to read any files contained in the corresponding

sub-tree. Another feature that is being discussed, from the ACM perspective, is the one in which there are multiple guards for the same object, a situation resembling the concept of link in traditional UNIX file-systems.

3.2.1 Decision Algorithm

At the core of the Pepys ACM there is the algorithm deciding whether or not to grant a given user access to a given file for a given operation. The central idea for such algorithm is: "more specific rules take precedence over more generic ones". This means that, if the entity can reach an object, AC rules directly attached to it have priority over AC rules inherited by guard objects or other ancestors (the o-rules described above are the only exception, when present).

The decision algorithm locating the proper permissions applying to a given entity for a given operation (e.g., write) on a given object, can be expressed shortly in these few steps:

1. Traversability check: the system checks that the entity has the right to traverse (e.g., 'x' permission bit) all the existing guards going from the file-system root down to the desired object, looking at those guards ACL tables; the traversability permission, in such tables, can either be granted directly to the entity attempting the access, or indirectly through any of the entity parents or ancestors, in the belong-to relationship;

2. Check if there is a rule for the entity in the object ACL;

   (a) if there is a match, its permissions mask are used to determine the access;

3. Check if there is a rule for any ancestor of the entity (i.e., as due to the belong-to specified relationships), giving priority to rules naming direct ancestors, then 2nd level ancestors, etc.;

   (a) as soon as a match is found, its permissions mask are used to determines the access;

4. Get the inherited rules from the object guard and start again the algorithm from step 2;

5. If there are no rules about the entity or for one of its ancestors the access is denied.

It is important to say that the o-rules affecting a given entity are combined with the permissions mask returned by the algorithm above; this operation gives us the effective permissions which the entity owns on the object.

Moreover, as we can see, it is been decided to give the priority to the entity ancestors, named in the specific object, rather than a possible rule for the applicant entity in the object guard; this because we consider more accurate the rules contained in the specific object rather than those in its guard.

Furthermore, since every entity belong to `nobody` entity, if the `nobody` ACL rule is present, it allows to inhibit inheritance of guards rules; since the algorithm would break at step 3. Same reasoning can be made for `others` applies to every authenticated entity.

This makes the algorithm very flexible since is possible decide when the decisional process has to stop.

3.2.2 Delegation

Entities can delegate others entities to act in their behalf, on a given object.

Each delegation is associated with a specific object and contains: the name of the delegator, the name of the delegee, a set of permissions assigned to the delegee and an expiration date. Clearly, the permissions granted by delegation cannot be higher than the ones held by the delegator on the object.

Two kinds of delegation are possible:

1. On-line.
2. Off-line.

In the first one the delegator issues the delegation to the system, merely specifying in it who is the delegee, its permissions and the file-system object on which the delegation is applied. In the second method the delegator issues a signed delegation to the delegee which, when it wants to perform an action on behalf of the delegator, will present it to the system.

In the first case the signature is not required, since the system knows who is the delegator and its permissions (for the anonymous cases see below); instead in the second case the system, before approve the delegation, has to know who is the issuer; hence the delegation has to be signed by the delegator.

The delegations are taken in account using the same algorithm described above, and only if the access is denied using the regular ACL rules.

3.3 Authentication

Authentication of users has been temporarily realized as a simple (hashed) password verification. Authentication is not mandatory, to connect to the server. An entity can have access to the system without having authenticated itself. In this case, the system considers the connected entity as being the `nobody` entity, thus the access-control permission specified for such entity throughout the file-system apply. While being connected to the system, an entity can authenticate itself whenever needed, upgrading its session from the rights corresponding to the only `nobody` entity to the rights associated with its actual name.

One of the goals of the Pepys file-system is to become a content-distribution platform. Supporting an unauthenticated state of the session is useful, in such context, to realize a sort of "incognito" mode of access by which public contents can be distributed worldwide without requiring users to reveal their identities.

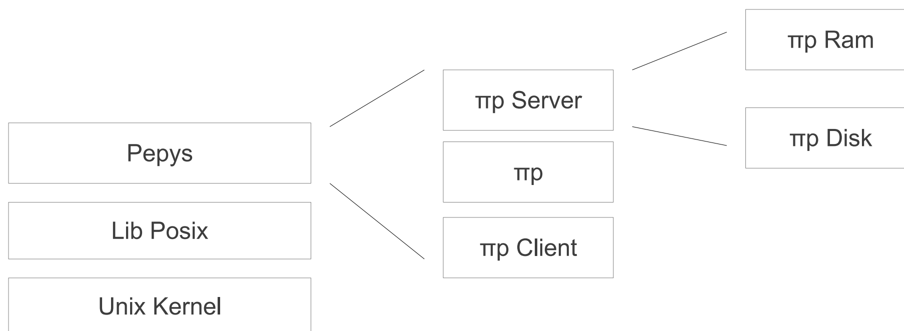


Figure 4.1: Porting on Linux implementation.

Hence is clear that the system must be able to treat in a different way the authenticated entities from the other ones; this is achieved by using the couple `others` and `nobody`. Indeed `others` refers to entities which are logged into the system, instead `nobody` to every entity present in the system (including both authenticated and not).

To understand better how these two entities are used, consider an ACL table. An ACL entry referring to the `others` entity applies to "every user logged and authenticated into the system but for which no other ACL entries have been found in the ACL table"; an ACL entry referring to the `nobody` entity, instead, applies to "every user logged onto the system, either authenticated or not". ACL entries for `others` have priority over the ones for `nobody`, i.e., the AC engine behaves as if the former entity were a subgroup of the latter one (see Figure 3.1).

Finally, if a server needs to authenticate users before allowing access to its contents, this can always be done by specifying the permissions wanted for the authenticated entities following the rules above (and using `others` if needed) and no access rights for the `nobody` entity.

4 Implementation Notes

4.1 Porting on Linux

The first step in our work was to unplug the Pepys file-system from its original structure (shown in figure 1.1) and therefore build a layer, called *Lib Posix*, in order to make the Pepys file-system runnable on UNIX machines.

In order to allow operations of swapping/loading objects from/into RAM, a new component has been added to the Pepys server, called *Pipdiskfs* (since the original structure provided only a RAM file-system).

Our porting relies on FIFO queues, provided by UNIX file-systems, in order to exchange ΠP messages between the server and the clients (however, we plan to switch to UDP-based communications).

The implementation is shown in Figure 4.1.

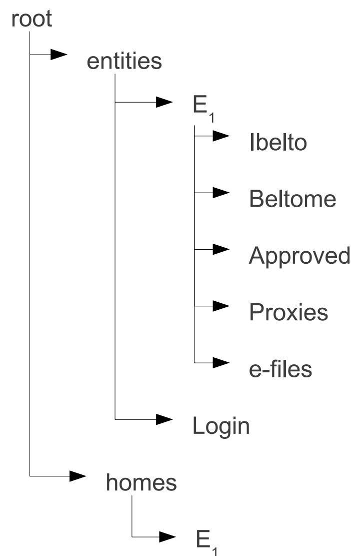


Figure 4.2: File-System structure.

The developed software included, in addition to the Pepys server, a few other tools:

1. Administration tool allowing for initializing the file-system, specifying:
   (a) Entities allowed and their login password.
   (b) Server name.
   (c) Mount point (on the underlying Linux file-system) to allow for swapping/loading of objects from/into the RAM.
   (d) Path of directory that will contain temporary files (i.e., named FIFOs currently used for client/server communications).
2. An interactive terminal in which is possible interact with the Pepys file-system (create files, administer ACL settings).
3. A set of "ad-hoc tests" to test the main file-system features.

4.2 File-System Structure

Inside the file-system the entities are represented by special guards, which own a set of special files. Moreover, as we can see in Figure 4.2, each entity is associated with a `home` object (folder) over which it has full control.

Particularly each entity object guards (i.e., contains):

Ibelto/Beltome: necessary to establish a new relationship.

Approved: list of entities which the named one belongs to

Proxies: provides a mechanism for permissions delegations.

e-files: others entity files as, for example, entity public key.

Each entity guard is managed by the guard above called `/entities`, which holds also a special file called `Login` in order to allow entities authentication.

An ACL table is represented by an object attribute, which can be changed only by the co-owners as reported in such ACL.

Instead an object delegation is a special object attribute, managed by the system, and hidden from the user's point of view.

The motivations behind this implementation is discussed in Section 4.4.

4.3 Entities relationship

When an entity wants to become member of another entity's users group, it writes the name of the other entity in its `Ibelto` file (over which it normally has write permission). The other entity (or the system administrator), on its own, has to write the name of the first entity in its `Beltome` file, in order for the new relationship to become effective. For each entity, the effective `Ibelto` relationships are reported in the `approved` special object within the entity folder, normally accessible to it for reading.

It is impossible for an entity to remove from its parents the system entities `nobody` and `others`. Also, depending on how the system is being administered, it is possible to allow users to write to their own `Ibelto` file, enabling them to propose changes to their belong-to relationship, including their removal from groups they belong to. On the other hand, it is equally possible to forbid such write operations, leaving the administration of users and groups entirely to system administrators, as it commonly happens in nowadays operating systems.

4.4 Delegation

As we said, two kinds of delegation are possible. In the on-line case, when an entity wants to delegate other entities it has to write the delegation in its `proxies` file. Specifically it has to indicate who is the delegee, its permission, an expiration date and the object to which the delegator is referring to. After that, the system will consider the delegation as effective only if it is compliant with the delegator's permissions on the specified object (i.e., an entity cannot delegate permissions it does not possess over a file-system object).

In the off-line delegation method, the delegee must specify in its `proxies` file who is the delegator and a valid path where the signed delegation is stored. The system then checks the delegation signature using the public delegation key available in the entity folder, and, only if the verification succeeds, is the delegation considered effective.

A valid delegation acts like a temporary ACL entry. However, the delegee might not have permissions to administer an object ACL, still be willing to

delegate some other entity to perform actions on its behalf on that object. The proposed model allows for this kind of scenarios, merely allowing to each entity to solely write/read its own `proxies` files. As a consequence, the system will make the requested delegations effective, or ignore them, if they are invalid.

4.5 Authentication

When a client logs onto a Pepys server, it is not required to authenticate immediately, resulting in a session being in an unauthenticated state. This means that the `nobody` access rights apply for the client, whenever an operation on the file-system is attempted. The client can authenticate itself at any time by using the special file `Login`. Specifically, when an entity wants to upgrade its session, it has to write its (SHA-256) hashed password using a write command. The server compares the hashed password with the one stored in the entity password file, and, if they match, the entity session is upgraded to an authenticated state. From now on, the actual name of the entity is used for checking the access rights of the user.

Note that the `Login` file is a special file, in that it does not really store any password. Such file can be opened by multiple remote clients concurrently without problems, as in the implementation the authentication material being provided by each client is kept into a separate buffer associated with each session.

Note that, thanks to the characteristics of the ΠP protocol to group multiple requests in the same message, it is possible for a remote client to stuff, within a single round-trip interaction with a Pepys server, the set-up of a session, opening of the `Login` file and writing of the password, opening of the target file-system file and issue of the desired read or write operation. However, the very simple authentication protocol realized so far is also relatively weak, in that it is easily subject to replay attacks, thus it can be improved by adding a time-stamp to the hashed password to be written into the `Login` file, or a server-provided random number (i.e., a *nonce*). Though, the last mechanism would require at least two round-trips with the server.

Finally, we plan to review and improve the authentication mechanism by integrating it with cryptographic extensions of the ΠP protocol which are being designed at the time of writing, that will allow for having encrypted client-server interactions.

5 Conclusions and Future Work

In this paper, an access-control model for the Pepys Internet-wide distributed file-system has been proposed, highlighting the main characteristics of its design. The proposed model takes into account the basic principles behind the well-known POSIX ACL standard and other widely used file-systems, enriching the model with characteristics that are inspired to the general principles of the Pepys distributed file-system.

The paper provided also a few notes on how the model has been implemented in a Linux port of the Pepys current code base.

Possible future work on the topic include: complete the implementation (under way) of digitally signed delegations; extend the delegations features including control of the delegation chain depth; integration of Pepys and particularly of the current authentication mechanism with properly designed cryptographic extensions to the ΠP protocol; evaluate the performance of the current ACM features, and possibly optimize the most recurrently used code paths.

References

[1] *IEEE Std 1003.1-2001, Open Group Technical Standard–Standard for Information Technology–Portable Operating System Interface (POSIX)*, 2001.

[2] Goerge Coulouris, Jean Dollimore, and Tim Kindberg, editors. *Distributed Systems: Concepts and Design*, chapter Amoeba. Addison-Wesley, 1994.

[3] Russ Cox, Eric Grosse, Rob Pike, David L. Presotto, and Sean Quinlan. Security in plan 9. In *Proceedings of the 11th USENIX Security Symposium*, pages 3–16, Berkeley, CA, USA, 2002. USENIX Association.

[4] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. Http authentication: Basic and digest access authentication. RFC 2617, jul 1999.

[5] Andreas Grünbacher. Posix access control list on linux. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[6] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, November 1992.

[7] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, Crystal City, Virginia, 1998.

[8] Sape J. Mullender and Andrew S. Tanenbaum. Protection and resource control in distributed operating systems. *Computer Networks*, 8(5-6):421–432, 1984.

[9] J. Sacha, J. Napper, S. Mullender, and J. McKie. Osprey: Operating system for predictable clouds. In *Proceedings of Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1 –6, june 2012.

[10] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The nist model for role-based access control: towards a unified standard. In *Proceedings of the fifth ACM workshop on Role-based access control*, RBAC '00, pages 47–63, New York, NY, USA, 2000. ACM.

[11] Guido Socher. File access permissions. 2000.

[12] William R. Stanek. File and folder permissions. In *Microsoft Windows 2000 Administrator's Pocket Consultant*, page Chapter 13, 2002.

[13] Stephen Tweedie. Ext3, journaling filesystem, 2000.

[14] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. Webos: operating system services for wide area applications. In *Proocedings of High Performance Distributed Computing, 1998. The Seventh International Symposium on*, pages 52 –63, jul 1998.

Atomic increments

Enrique Soriano−Salvador
Laboratorio de Sistemas
Universidad Rey Juan Carlos
esoriano@lsub.org

Gorka Guardiola Múzquiz
Laboratorio de Sistemas
Universidad Rey Juan Carlos
paurea@lsub.org
11/8/2012

ABSTRACT

Following a discussion in the NIX mailing list, we have measured the cost of different implementations of atomic increment using a µbenchmark. These implementations are (i) the standard assembler routine using a hardware protected addition instruction; (ii) a lock−free assembler routine using compare−and−exchange; and (iii) a spin lock protected counter. In this paper we present the implementations, the results of running a µbenchmark against them and our conclusions about which one we should keep. Quite surprisingly, spin locks are better than the other implementations for a high contention scenario in a 32 cores multiprocessor.

Introduction

While implementing various synchronization mechanisms in NIX [2], we started a discussion about the cost of different atomic operations. In particular, one of the simplest operations we can imagine is incrementing or decrementing an atomic integer (`ainc`). Several mechanisms have been implemented in the C library and in the kernel while trying to optimize this operations. Specifically, we found two assembler implementations of `ainc` and a C implementation using spin locks [1].

Ainc implementations

The first assembly routine we found uses a hardware protected addition instruction (Intel's LOCK prefix). As the AMD64 manual states [3]:

> The LOCK prefix causes certain kinds of memory read−modify−write instructions to occur atomically. The mechanism for doing so is implementation−dependent (for example, the mechanism may involve bus signaling or packet messaging between the processor and a memory controller). The prefix is intended to give the processor exclusive use of shared memory in a multiprocessor system.

The routine for AMD64 is:

```
TEXT lainc(SB), 1, $-4
    MOVL    $1, AX
    LOCK; XADDL AX, (RARG)
    ADDL    $1, AX
    RET
```

The routine for 386 is:

```
TEXT lainc(SB), $0
    MOVL    l+0(FP), AX
    LOCK;   INCL 0(AX)
    RET
```

The version of ainc we show next is also an assembly routine, but it is subtler. This implementation is *lock free* [4]. It copies the value of the counter, updates it and tries to put it back into the counter with compare-and-exchange. If the value of the counter changed in the window, the attempt fails, and the operation is retried.

The routine for AMD64 is:

```
TEXT lfainc(SB), 1, $0
ainclp:
    MOVL    (RARG), AX      /* exp */
    MOVL    AX, BX
    INCL    BX              /* new */
    LOCK; CMPXCHGL BX, (RARG)
    JNZ     ainclp
    MOVL    BX, AX
    RET
```

The routine for 386 is:

```
TEXT lfainc(SB), $0 /* int ainc(int *); */
    MOVL    addr+0(FP), BX
ainclp:
    MOVL    (BX), AX
    MOVL    AX, CX
    INCL    CX
    LOCK
    BYTE    $0x0F; BYTE $0xB1; BYTE $0x0B   /* CMPXCHGL CX, (BX) */
    JNZ ainclp
    MOVL    CX, AX
    RET
```

The third version uses the standard Plan 9's C library spin lock implemented with test-and-set (which is just the XCHGL instruction which acts as if having and implicit lock prefix) and back off:

```
void
lock(Lock *lk)
{
    int i;

    /* once fast */
    if(!_tas(&lk->val))
        return;
    /* a thousand times pretty fast */
    for(i=0; i<1000; i++){
        if(!_tas(&lk->val))
            return;
        sleep(0);
    }
    /* now nice and slow */
    for(i=0; i<1000; i++){
        if(!_tas(&lk->val))
            return;
        sleep(100);
    }
    /* take your time */
    while(_tas(&lk->val))
        sleep(1000);
}
```

Test-and-set for AMD64 is:

```
        MOVL    $0xdeaddead,AX
        XCHGL   AX,(RARG)
        RET
```

Test-and-set for 386 is mostly the same:

```
    TEXT    _tas(SB),$0

        MOVL    $0xdeaddead,AX
        MOVL    l+0(FP),BX
        XCHGL   AX,(BX)
        RET
```

Finally, this version of the atomic increment is:

```
        lock(&l);
        counter++;
        unlock(&l);
```

Clearly, having three different implementations for such a simple operation is an overkill. The question is which one should we keep. We have measured them in various scenarios to try to answer to this question.

Methodology

The machine(s) in which we took the measurements are a 32-core AMD K10 Opteron 6128 running NIX and a 2594MHz Pentium IV running Plan 9. For the AMD multiprocessor, we have taken measurements using 1 and 32 cores. NIX uses an AMP scheduler when running on multiple processors, and a SMP scheduler for one processor. We measured the cost for the three implementations of `ainc` showed in the previous section.

In AMD64 (only for the implementation based on the LOCK prefix) we have taken the measurements for 64 byte aligned values and unaligned (i.e. 32 byte aligned) values.

The µbenchmark measures the time taken to increment the value with different levels of contention. The program spawns a number of processes that try to increment the counter in a closed loop. The contention depends on two factors:

•The number of processes spawned. The µbenchmark has been executed for 1, 2, 5, 10 and 20 processes. In addition, the multiprocessor test has been executed for 50 and 100 processes.

•The amount of time wasted between increments in the loop. The µbenchmark has been executed for 0, 1, 50, 100, 200 and 500 ns. This time, of course, is not taken into account when measuring the increment. The rationale behind adding this parameter is twofold. First, small values of this delay (`waste`) try to compensate the effect on contention of the time taken to perform the `ainc`. A faster `ainc` would have a higher rate of operations executed, and thus, higher contention. When `waste` is much bigger than the time taken to perform an `ainc`, the contention only depends on the number of processes and size of `waste`, and not on the time to perform of the operation itself. Second, the delay allows us to measure the operation in different contention scenarios for the same number of processes and processors.

The core of the program is (in pseudocode):

```
poll until all procs ready
for i in  1..1000
    t1 = taketime()
    ainc(v)
    t2 = taketime()
    times[i] = t2 - t1
    do
        wastesometime()
        t3 = taketime()
    while t3-t1 < Waste
endloop
```

We take the time using the time stamp counter, by using the RDTSC instruction. Notice that in the multiprocessor case, the hardware counters are synchronized on boot. We booted the machine for each execution of the µbenchmark. The drift of the counters for the time taken to perform the measurements is negligible.

Results

What follows depicts Tukey diagrams of the measurements. The labels shown in the graphs are:

•`lainc-align`: the `ainc` implementation based on the LOCK prefix, value aligned to 64 byte.

•`lainc-noalign:` the `ainc` implementation based on the LOCK prefix, value unaligned to 64 byte, aligned to 32 byte.

•`lockfreeainc`: the lock-free `ainc` implementation based on compare-and-swap.

•`lock`: the `ainc` implementation using spin locks.

The most surprising result is that for the multiprocessor case, the spin lock version is much faster than the other implementations. This is not what we expected. All the implementations end up using a LOCK prefix instruction (note that there is an implicit LOCK prefix in the XCHGL instruction). In addition, the spin lock implementation

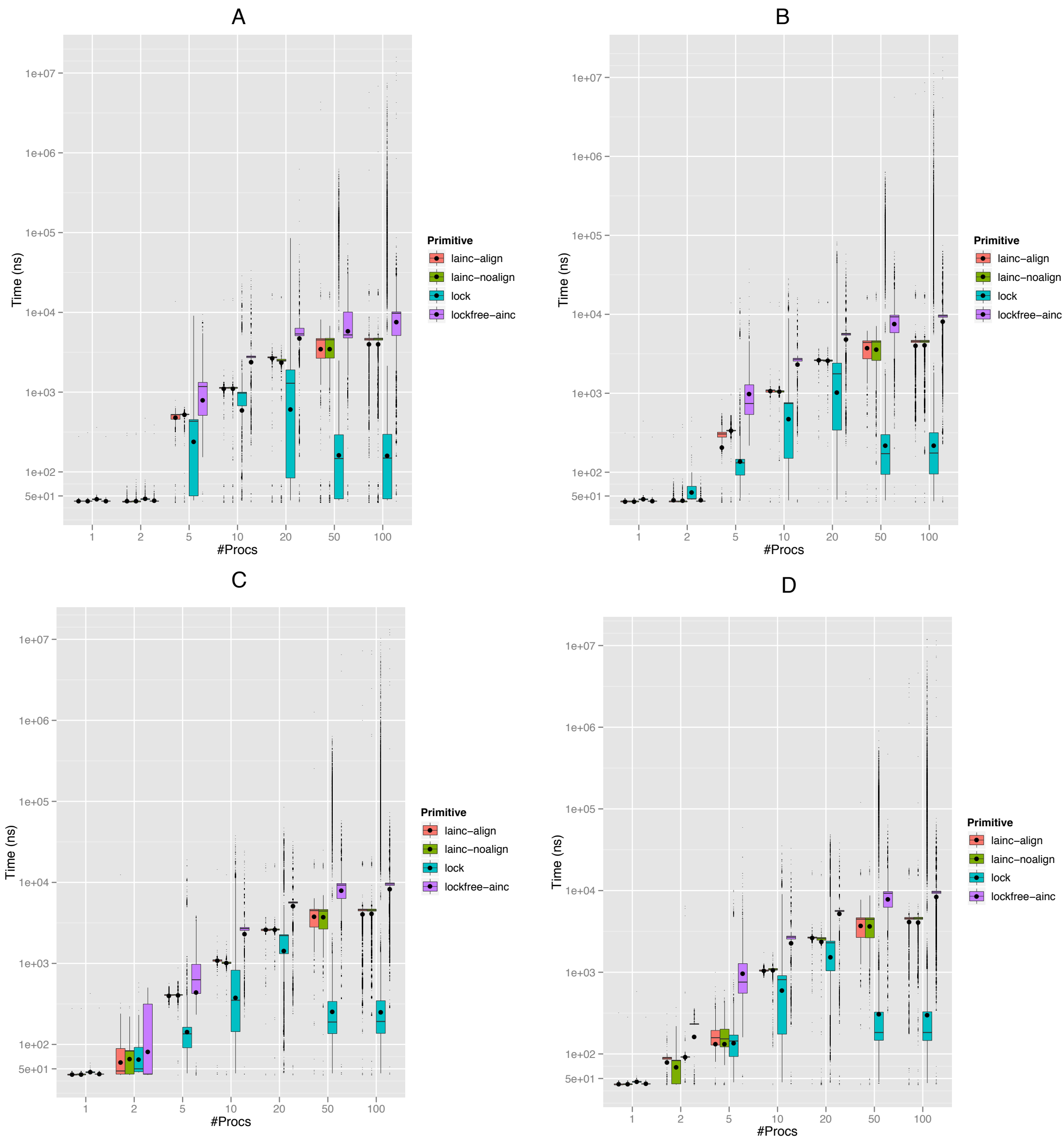


Figure 1: *NIX on AMD64, 32 cores, (A) waste = 0 ns, (B) waste = 50 ns, (C) waste = 200 ns, (D) waste = 500 ns*

requires to execute more instructions and an extra function call (unlock).

It may well be that the backoff of the spin locks is responsible for their efficiency in the multicore case. Nevertheless, note that in Figure 2 (A), for 1 process (i.e. no contention), the spin locks are better than the other implementations. This may be due to the effect of extreme flooding in the coherency fabric. This kind of effect is what we tried to correct by using different waste values. There seems to be no effect associated to the alignment of the values.

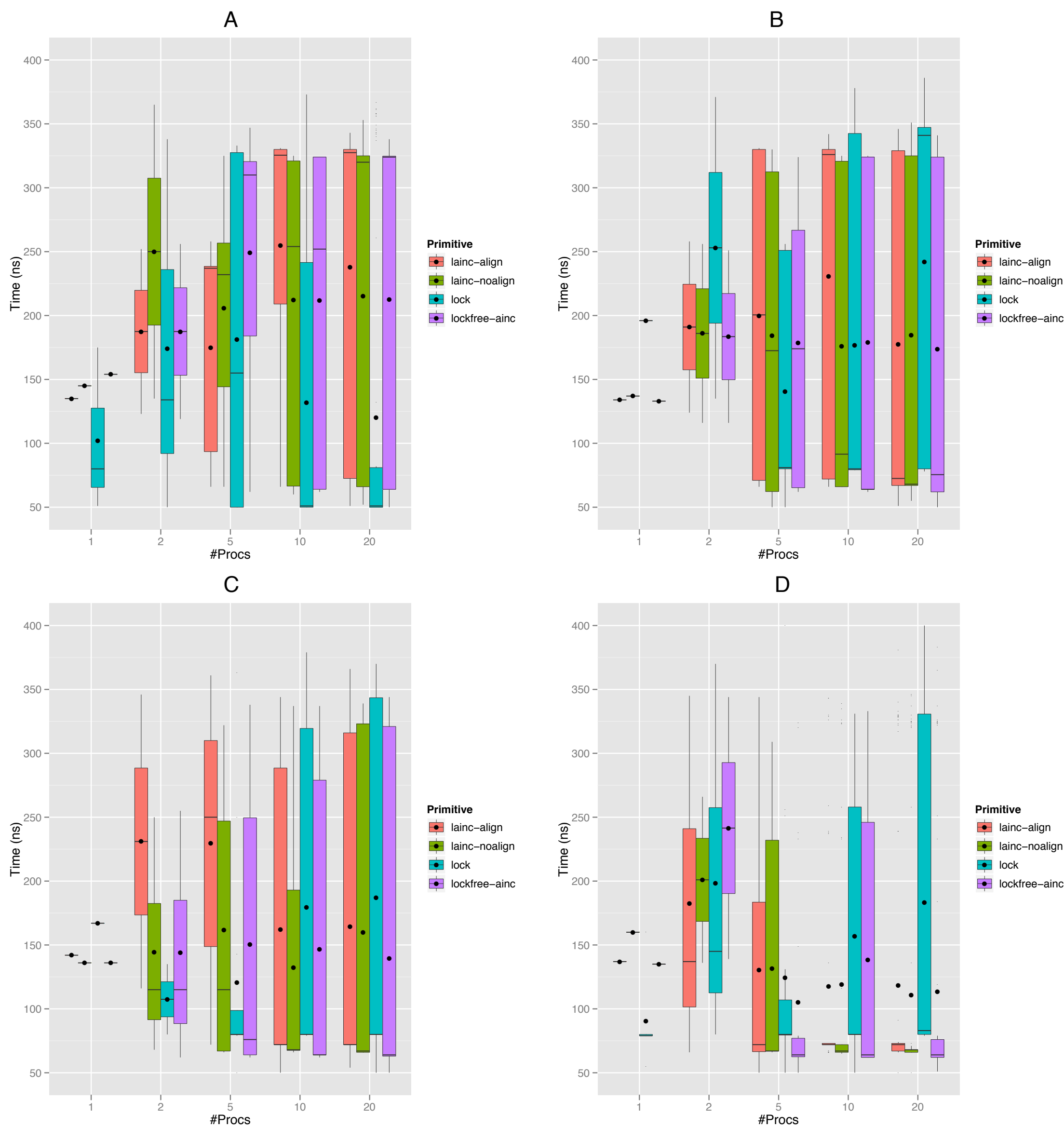


Figure 2: *NIX on AMD64, 1 core, (A) waste = 0 ns, (B) waste = 50 ns, (C) waste = 200 ns, (D) waste = 500 ns*

Notice also that, except for one case (Figure 2 (C)), the lock-free version seems to perform equal or worse than the other implementations.

In the 386 experiments, the results show more predictable results. The interesting result in these experiments is that the lock-free implementation is still worse than the others. This is easy to understand if the CMPXCHGL is as expensive as the XADDL with the LOCK prefix, because the lock-free implementation requires more instructions and may need several attempts to perform the operation.

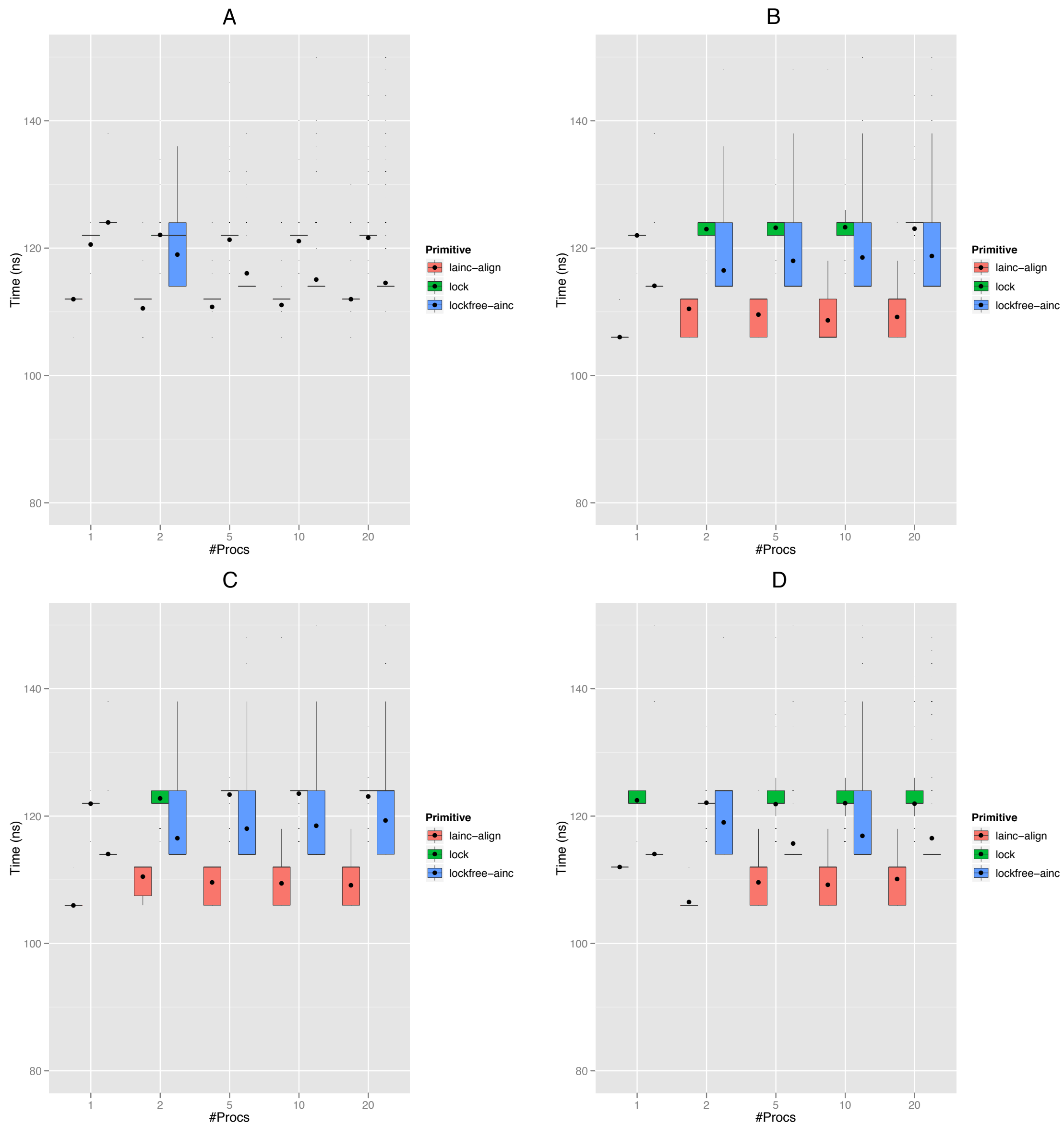


Figure 3: *Plan9 on 386, 1 core, (A) waste = 0 ns, (B) waste = 50 ns, (C) waste = 200 ns, (D) waste = 500 ns*

Conclusions and Future Work

We can conclude that the lock-free implementation is not the best choice in any case, at least with the current implementation of the instructions for these architectures. Between the two other implementations, it is difficult to conclude which one is better. For the 386 experiment there is a 10% gain when using the lock prefix implementation. On the other hand, in most cases for the multiprocessor it seems to be better to use the spin lock implementation. When using only 1 core, the results are (more or less)

balanced. Nevertheless, when using 32 cores, the spin lock version is much better. Another argument in favor of the spin lock version is that spin locks are already needed as a general purpose synchronization primitive.

Last, note that the AMP scheduler of NIX is new, so there may be lurking bugs affecting the results of the experiments.

References

1. T. E. Anderson, E. D. Lazowska and H. M. Levy, The performance implications of thread management alternatives for shared-memory multiprocessors, *IEEE Transactions on Computers 38*, 12 1631-1644.
2. F. J. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich and E. Soriano, Nix: a case for a manycore system for cloud computing, *Bell Labs Technical Journal 17*, 2 41-54.
3. A. M. D. Inc., AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions.
4. T. Johnson, Characterizing the performance of algorithms for lock-free objects, *IEEE Transactions on Computers 44*, 10 1194-1207.