Considerations for the SDP Operating System

Document Number.....SDP Memo 063
Document Type.....MEMO
Revision.....01
Author.....N. Erdödy, R. O'Keefe
Release Date.....2018-08-31
Document Classification..... Unrestricted
Status..... Draft

Lead Author	Designation	Affiliation
Nicolás Erdödy	SDP Team	Open Parallel Ltd. NZ SKA Alliance (NZA).
Signature & Date:	DocuSigned by: 8F8689DA5C594AB...	10/21/2018 8:19:36 PM PDT

With contributions and reviews greatly appreciated from		Affiliation
Dr. Richard O'Keefe	SDP Team, NZA	University of Otago - Open Parallel (NZA)
Dr. Andrew Ensor	Director, NZA	AUT University (NZA)
Piers Harding	SDP Team, NZA	Catalyst IT (NZA)
Robert O'Brien	Systems Engineer / Security	Independent
Anonymous Reviewer	CEng (UK), CPEng (NZ)	Manager, NZ Govt

ORGANISATION DETAILS

Name	Science Data Processor Consortium
Address	Astrophysics Cavendish Laboratory JJ Thomson Avenue Cambridge CB3 0HE
Website	http://ska-sdp.org
Email	ska-sdp-pa@mrao.cam.ac.uk

1. SDP Memo Disclaimer

The SDP memos are designed to allow the quick recording of investigations and research done by members of the SDP. They are also designed to raise questions about parts of the SDP design or SDP process. The contents of a memo may be the opinion of the author, not the whole of the SDP.

Acknowledgement:
The authors wish to acknowledge the inputs, corrections and continuous support from the NZA Team Members Dr. Andrew Ensor, Peter Baillie, Piers Harding and TN Chan.

2. Table of Contents

3. List of Figures

4. List of Tables

5. List of Abbreviations

ABI	Application Binary Interface
AIX	Advanced Interactive eXecutive
AltArch	Alternative Architecture
AMD	Advanced Micro Devices, Inc.
API	Application Programming Interface
ARM	Advanced RISC Machine
ASLR	Address space layout randomisation
BSD	Berkeley Software Distribution
BIOS	Basic Input/Output System
BLAS	Basic Linear Algebra Subprograms
CI/CD	Continuous Integration/Continuous Delivery
C&C	Component & Connector
CD	Compact Disc
CDC	Control Data Corporation
CIA	Confidentiality, Integrity, Accessibility
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
DALiuGE	Data Activated Liu Graph Engine
DEC	Digital Equipment Corporation
DLL	Dynamic-link Library
eBPF	Extended Berkeley Packet Filter
EC2	Amazon Elastic Compute Cloud
EFI	Extensible Firmware Interface
EU	European Union
FFT	Fast Fourier Transform

F/OSS	Free Open Source Software
FPGA	Field Programmable Gate Array
GB	Gigabyte
GNU	GNU's Not UNIX!
GPU	Graphic Processing Unit
HAL	Hardware Abstraction Layer
HFS+	Hierarchical File System Plus
HPC	High Performance Computing
HPE	Hewlett Packard Enterprise
IBM	International Business Machines
IETF	Internet Engineering Task Force
IME	Intel Management Engine
I/O	Input/Output
IoT	Internet of Things
IP	Internet Protocol
ISA	Instruction Set Architecture
JeOS	Just Enough Operating System
JIT	Just-In-Time
JNI	Java Native Interface
KAISER	Kernel Address Isolation to have Side-channels Efficiently Removed
KB	Kilobyte
KPTI	Kernel page-table isolation
LDAP	Lightweight Directory Access Protocol
LLNL	Lawrence Livermore National Laboratory
LSM	Linux Security Modules
LTS	Long-term Support
LTTng	Linux Trace Toolkit Next Generation

LXC	Linux Containers
LXD	(extension for) LXC
MB	Megabyte
MCP	Master Control Program
MIT	Massachusetts Institute of Technology
MPI	Message Passing Interface
NZA	New Zealand SKA Alliance
OS	Operating System
PAM	Pluggable Authentication Modules
PDP	Programmed Data Processor
PHP	Personal Home Page (now PHP: Hypertext Preprocessor)
PMC	Performance Monitoring Counter
PMU	Performance Monitoring Unit
POSIX	Portable Operating System Interface
RAID	Redundant Array of Independent Disks
RAM	Random-access Memory
RAM	Reliability, Availability and Maintainability
RHEL	Red Hat Enterprise Linux
RRDtool	Round-robin Database tool
SCS	bullx SCS – bullx SuperComputer Suite
SDLC	Software Development Life Cycle
SDP	Science Data Processor
SECCOMP	Secure Computing Mode
SELinux	Security-Enhanced Linux
SHA	Secure Hash Algorithm
SHMEM	Symmetric Hierarchical MEMory
SKA	Square Kilometre Array

SNMP	Simple Network Management Protocol
SoC	System on Chip
SPARC	Scalable Processor Architecture
SSD	Solid State Drive
SUSE	Software und System Entwicklung
TB	Terabyte
TCP	Transmission Control Protocol
TOSS	Tri-labs Operating System Stack
UCI/OIT	University of California, Irvine – Office of Information Technology
UEFI	Unified Extensible Firmware Interface
ufw	Uncomplicated firewall
UNSW	University of New South Wales
USAF	United States Air Force
VAX	Virtual Address Extension
VM	Virtual Machine
ZFS	Z File System

6. Introduction

The present IT industry has managed to successfully articulate and demonstrate that software is not static because the business rules, and operational environments that it supports are ever changing. This means that delivering a viable software service is a constantly changing and organic product that needs continual renewal and revision. New and innovative projects are by their very nature, likely to discover new and innovative ways of solving problems. At each revision a project is typically constrained by hardware limitations, budget constraints, available skills and time scales of the present day. In order to satisfy these constraints, and deal with an acceptable level of risk, the building of such systems is evolutionary by nature. Evolutionary discovery becomes build something that will work "now", but know that the ground rules will continue to change over time - technology improves, requirements and objectives continually refine, discovery informs direction.

In a project like the SKA the data centre operations and software will continually change to meet evolving experimentation based on new learnings, and advances in technology that make new approaches possible. The SKA will be doing this for the foreseeable future and continue to guide the advances in computation over the life-time of the project. Software is an area of constant research and innovation.

In particular testing and validation of the SDP platform cannot help but be new and novel, because the technology and scale is new and novel. As no other system in existence has handled the same throughput and workloads anticipated in the SKA, new testing and validation strategies will need to be employed to deal with it.

“Future HPC systems will be characterised by extreme heterogeneity. We will see increasing heterogeneity in virtually every aspect of node architecture from computational engines to memory systems. We will see increasing heterogeneity in applications, including heterogeneity within applications... We will also see increasing heterogeneity in the shared services (i.e. storage and visualisation systems) that are connected to HPC systems.”

“All of this increasing heterogeneity is certain to create new challenges in the design and implementation of operating and runtime systems. There will be new kinds of resources to manage and many resource management tactics will be invented (and some re-discovered and adapted) to address the new heterogeneity. In essence, we will tacitly agree that the operating and runtime systems need to adapt to enable the inevitable integration of new technologies, applications, usage models, and shared services. While this agreement is critical for our ability to make incremental progress, we, as community, must step back and ask the relevant question: Does the OS or runtime system bear the brunt of the adaptation, or will we be able to insist on changes in the technologies, applications and environment?” (RD01)

These “Considerations for the SDP Operating System” aim at these relevant questions by initially discussing the scope of the OS for the SKA and listing some options, highlight how security in the OS will be important, and reason toward a “minimalist OS” which might be the most appropriate route for the SDP to follow given that SKA would like portability of SDP to

mixed software environments (SDP_REQ-813) -where the SDP Operational System software is more portable because it does not depend on specific vendor's hardware variations, instead depending on an OS image Core Infrastructure Services and a standard set of APIs (RD06 - Platform Services C&C View).

The memo also covers a view of Linux Performance Monitoring Systems and reports some preliminary results on measuring system overheads in and out of containers.

7. Executive Summary

For the purposes of this document, the “operating system” is the software layer or layers beneath the application layer. SKA OS focus is on *kernel, drivers,* and *libraries.*

The SKA SDP does not have a typical HPC workload. It is in many ways closer to a real-time system. Large variations in system call time may be a concern.

All systems of the TOP500 list of supercomputers have a flavor of Linux for OS. We list a dozen plus of candidates for the SKA, before making the case for a “Minimalist OS” for the SDP.

The SDP is a system of systems and through the expected 50 years’ life of the SKA we expect frequent modifications for the software, mainly in the form of new or revised algorithms and applications.

Operating systems that are known to run on several kinds and several generations of hardware offer us some hope that they may be usable in the future. The top four candidates appear to be Alpine Linux, LinuxKit, Ubuntu Core, and Ubuntu Server configured for a minimal VM.

They are all known to work well in a container environment and are actively maintained by credible organisations that are not likely to disappear overnight.

Security should be fundamental to all design, not bolted on to an old paradigm. It’s not enough to build something and try to make it secure after the fact. We propose to build security through progressive layers that deliver true defense in depth, from the bottom up through the whole SDP and the SKA.

The SDP will probably rely on Docker provisioning for upgrades and so to a large extent won't *need* the management services provided by the IME, so the IME should be disabled to the extent practical (which is not 100%).

The SDP's connections with the outside world should go through a firewall that allows the minimal possible range of services in either direction, and should run on a different architecture that lacks an IME analogue.

Communication *within* the SDP needs to use a small number of known and carefully audited protocols.

Computation within the SDP will use a fixed set of programs, so we can probably get away with disabling KPTI internally. But gateway machines should enable it.

8. References

8.1. Applicable Documents

The following documents are applicable to the extent stated herein. In the event of conflict between the contents of the applicable documents and this document, **the applicable documents** shall take precedence.

Reference Number	Reference
AD01	

8.2. Reference Documents

The following documents are referenced in this document. In the event of conflict between the contents of the referenced documents and this document, **this document** shall take precedence.

Reference Number	Reference
RD01	Arthur B. Maccabe. 2017. Operating and Runtime Systems Challenges for HPC Systems. In Proceedings of ROSS'17, Washington, DC, USA
RD02	Exterminate All Operating System Abstractions. D. Engler and M.F. Kaashoek. MIT Laboratory for Computer Science, Cambridge, MA.
RD03	SKA - SDP. B.Nikolic, University of Cambridge. Presentation at NRAO, 2013.
RD04	Google - Security in the Cloud. 2018. Urs Hölzle. https://www.blog.google/perspectives/urs-hoelzle/security-cloud/
RD05	SDP Memo 051. Cloud Native solution architecture for the SDP. Piers Harding
RD06	SDP Platform Services Component and Connector View. April 2018. SKA-TEL-SDP-0000013. J. Garbutt, J. Taylor, P. Harding, A. Ensor, V. Allan, P. Wortmann
RD07	SDP RAM report. SKA-TEL-SDP-0000115. April 2018. Ferdl Graser, L. Christelis.

9. Scope for the SKA Operating System

This section introduces goals and scope for the SKA OS and presents a list of OS candidates.

9.1. Goals

The SKA operating system

• may be heterogeneous: the compute nodes might run a stripped down version while management nodes might run a full version

• must be reliable

• must be maintainable

• must be scalable

• should be proven for HPC

• run-time timing should be consistent.

9.2. Scope

• For the purposes of this memo, the “operating system” is the software layer or layers beneath the application layer. The interface is like POSIX or Win64.
• Linux rules supercomputing: Out of the current TOP 500 list published in June 2018, all systems provide “Linux” as the operating system.
• The SDP will be built using commodity boards and so BIOS/UEFI will be provided by the board manufacturer and GPU drivers will also be provided; these things are out of scope.
• The underlying hardware is expected to change over the lifetime of the system, and at times the SDP may contain a mix of different hardware types. This should not require major software changes, so the selected system should be known to work at least on x86-64 and AArch64 (ARM).
• Hypervisors are out of scope.
• It is all but certain that containers will be used and highly likely that Docker will provide container services. It is desirable that the “operating system” should support the applications *and* containerisation. This leaves Solaris (with zones and Docker) and FreeBSD (with jails and Docker) and Linux (with LXC and Docker) as candidates.
• The operating system has to be adequate in principle, workable in practice, and acceptable in politics. In popularity, Linux is a clear winner.
• The budget is tight and tightening. Whatever system is chosen, the less modification needed for the SKA the better.
• There is a notion of a “Data Centre Operating System”, covering such things as Spark, Mesos, and arguably DALiuGE. This is out of scope, even though Mesos has kernel components.
• There are no clear Reliability, Availability, and Serviceability requirements, but it is clear that these issues will be important. The Reliability, Availability & Maintainability (RAM) requirements of

the SDP are architectural drivers for the software and hardware design selection and are presented in the SDP RAM report (RD07).
• So will file system latency and throughput. Modern UNIX systems typically support several file systems, so a full survey about OSs should seek information on which systems are used. What suits a data centre might not suit the SKA.
• One of the distinguishing features of Linux distributions is the user interface kit. This is quite irrelevant for the SKA compute nodes.
• Another distinguishing feature is the range of applications that have been ported. Most of this is also irrelevant for the SKA compute nodes (but is a consideration with support services, admin tooling and DevOps tooling that will likely run on every node).
• That means the focus is on *kernel, drivers,* and *libraries.*
• The SKA SDP does not have a typical HPC workload. It is in many ways closer to a real-time system. Large variations in system call time may be a concern.

Requirements for the SDP OS can be found in section 12.5

9.3. TOP 500 list

Table 1 shows the operating system figures provided by the TOP500 website for June 2018 (www.top500.org/statistics/list)

Share by System	Share by Performance	Linux Variant
50.8% (254)	29.5%	“Linux” not otherwise specified
23.2% (116)	12.5%	CentOS
9.8% (49)	14.5%	Cray Linux Environment
2.6% (13)	2.4%	SUSE Enterprise Server 11
2.6% (13)	2.9%	bullx SCS
2.2% (11)	1.3%	TOSS
1% (5)	0.5%	RHEL 7.3
0.8% (4)	0.6%	Ubuntu Linux
0.8% (4)	6.5%	Red Hat Enterprise Linux
0.8% (4)	11.4%	RHEL 7.4
5.4% (27)	17.9%	Others

Table 1 - TOP500 list – OS statistics

According to the full table, 2 of the systems are Kylin, but 206 of the TOP500 are systems located in China (41.2%), so the correct number could be higher. There is also at least one Ubuntu 14.04

system. The table is misleading in another way: bullx SCS (bullx supercomputer suite) "is a comprehensive, modular, integrated and open suite of products dedicated to managing supercomputers, HPC applications running on those supercomputers as well as their data". Apparently it's not an operating system itself, but an additional layer running on top of SUSE Linux Enterprise Server 11.

To further simplify the table, CentOS, Bullx Linux, and Scientific Linux are all derivatives of Red Hat Enterprise Linux. TOSS (LLNL's "Tri-Lab Operating System Stack") is also based on Red Hat. Another RHEL systems are hidden amongst "others", making a total of 157 RHEL-derived systems (31.4%).

10. Linux on Data Centres

The top four Linux distributions for data centre and cloud systems are
- Red Hat Enterprise Linux, claiming 65–80% of "enterprise" systems and 16% of OpenStack clouds;
- SUSE Linux Enterprise Server, claiming 25% of "corporate Linux";
- Ubuntu Server, claiming 55% of OpenStack clouds; and
- CentOS, with 20% of OpenStack clouds.

Count	Linux Variant
209.49	Ubuntu
88.49	Unspecified Linux
29.17	Unspecified Windows
17.17	Red Hat
13.69	CentOS
6.50	Fedora
4.29	Debian
1.42	SUSE
<1	All others combined

Table 2 - The Cloud Market report on OS in Amazon's EC2

10.1. The Initial List

As discussed above, all the candidates are Linux variants.

• **Kylin**, used on Tianhe-1A and Tianhe-2, is the Chinese “an official flavour of Ubuntu”. There is a paper that describes some of the modifications they made for Tianhe.

• **Ubuntu** is a strong candidate, due to OpenStack popularity.

• **Red Hat** Enterprise Linux and its derivatives, notably including CentOS.

• **CentOS** officially supports only x86-64. There are “AltArch” releases for AArch64 and PowerPC8. CentOS seems to have some preference in the scientific and OpenStack communities.

• **Scientific Linux**. This is a “rebuild” of RHEL “sponsored by Fermilab”.

• **SUSE**, used for example at the Irish Centre for High-End Computing. They claim that SUSE Linux Enterprise Service is good for HPC.

• The Trilab Operating System (TOSS) is “a custom derivative of” Red Hat. It’s used at LLNL, Los Alamos, and Sandia.

• Whatever they are using at “HPC at UCI/OIT”. They say they are using “Son of Grid Engine”, but don’t say what the OS is. Indeed, it’s not clear that all the nodes run the same OS.

• Rocks Cluster, originally based on Red Hat but now based on CentOS.

• Compute Node Linux, used on Cray systems, and apparently based on SUSE Linux Enterprise Server.

• HPE Apollo servers, as used at several HPC sites like the Texas Advanced Computer Center in Austin, support both Red Hat Enterprise Linux and SUSE Linux Enterprise Server.

• Qlustar is Debian/Ubuntu based and “has been the software engine for a large number of Linux HPC clusters running in industry and academia since more than ten years.”

• The operating system of the K computer is a Linux derivative and is Open Source.

• The Argo exascale operating system should also be examined.

11. Minimalist OS for the SDP

In this section we’ll reason toward why a “minimalist” OS might be the most appropriate route for the SDP to follow.

11.1. Scope and Planning Horizon

The Science Data Processor is a system of systems. This section is concerned with the operating system(s) running on the compute nodes, which will support a relatively stable workload and where scheduling is very important.

The SKA itself is supposed to have a 50 year lifetime. However, technology continues to change. We expect frequent modifications to the software during that lifetime, mainly in the form of new or revised algorithms and applications.

Hardware will certainly change out by 100% if only because of increase in performance/power. The price/performance of various processors is likely to change. While complete replacement of the computing hardware is relatively unlikely, additions to support greater workloads may well use a different mix of components. As an example, Fujitsu have switched their supercomputer development from SPARC v9 to ARM8. For that matter, porting an operating system between successive generations of the “same” ISA (Instruction Set Architecture) is not always trivial. Operating systems that are known to run on several kinds and several generations of hardware offer us some hope that they may be usable in the future.

There are two ongoing changes that may require changes in the relatively near term (five years).

One is the rise of SSDs. A 2011 laptop has 500 GB of rotating magnetic disc. A current one from the same brand comes with 500 GB of SSD. A couple of years ago, Seagate announced a 60 TB SSD with a 12 Gbps SAS interface and using 15W of power. In the same timeframe Toshiba demonstrated a 100 TB SSD (9W active, 0.1W idle). These things are still astronomically expensive. If/when the price comes down, the mix of storage devices and the scale of the file system will change. Apple, for example, are basically dropping the HFS+ file system[2], having introduced a new “Apple File System”[3] designed, amongst other things, to better exploit SSDs.

The other ongoing change is the introduction of bulk non-volatile memory. We might, for example, see the return of the CDC 6600’s “Bulk Store”, used for holding less frequently needed code and/or data. (The CDC’s 10 Peripheral Processors ran the operating system, not the main CPU, which is one of the reasons they were good at true real time.)

2 https://en.wikipedia.org/wiki/HFS_Plus

3 https://en.wikipedia.org/wiki/Apple_File_System

11.2. What rights do we need?

A question could be raised whether the operating system should be Free/Open Source Software or something else. It is useful to distinguish between several different rights we might need.

The Right to Use - Clearly we have to be able to use the software for a very long time. Some alternatives are

• Completely free software.

• A licence in perpetuity.

• Annual rental.

Rental would pose a serious risk.

One important issue is that the SKA may well outlast some of the current software companies, and is very likely to outlast their interest in the particular software we need.

The Right to Abandon - Can you abandon the software and use something else without penalty?

The Right to Support - The software has to be supported by somebody. What is the annual budget for software maintenance? Maintenance of stuff that is shared with other versions of Linux can be rented. Maintenance of stuff that is specific to the SKA is another matter.

It is particularly important that the SKA may outlive existing companies. This does not necessarily rule out commercial versions of Linux. It simply means that commercial software components must be put in escrow so that if the vendors go out of business or lose interest in maintaining it the SKA organisation gets full sources with the right to maintain them.

The Right to Inspect - There are two rights here but it is hard to disentangle them. It is clearly useful to be able to examine the sources to look for flaws and possible efficiency issues. It is very important to SKA operations to be able to monitor, trace, and measure the system, but this requires some degree of knowledge of the system internals in order to know *what* to trace and what the results signify.

The Right to Modify - Again there are two hard-to-disentangle rights here: the right to sustain existing function by corrective or adaptive maintenance and the right to improve the system by adding new features or improving non-functional aspects of existing ones. The SKA needs both.

The Right to Distribute - F/OSS software generally comes with the right to not only to modify but to distribute modified versions. It is not clear that the SKA needs this.

The Right to Control Rights - Software purpose-written for the SKA comes with this right.

It seems clear that the SKA needs the rights to use, abandon, inspect, and modify all the software it uses, in order to ensure that it can continue in use for the life of the system.

That does not mean that the software needs to be free, or open source. It just means that we need most of the rights that F/OSS grants. If these rights can be purchased for an affordable sum, well and good.

But it also seems clear that an annually renewed licence or even a triennially renewed licence won't do. The risk that a renewed licence might not be available (if the company loses interest in the product, as companies do), or might be unaffordable, or might be available only on unacceptable terms is a risk that the SKA might be rendered inoperable by the lawful actions of someone else.

11.3. What does "Minimalist Operating System" mean?

In this document, when we speak of a "kernel" we are referring to an operating system kernel, not the software discussed at the SDP Kernel Workshop[4].

When people speak of “an operating system”, they typically mean Windows, MacOS, Linux, Solaris, or something like that. But those names cover a wide range of different types of software of different criticality, including but not limited to

• hardware abstraction layer

• kernel

• platform-defined libraries (POSIX, or X11, say)

• application support libraries like Simple DirectMedia Layer, MVAPICH2 (MPI 3.1 over a wide range of transports), the GNU MultiPrecision integer and MultiPrecision Floating-point library, numeric libraries including FFT and BLAS, …

• programming language support libraries

• utility programs like linkers and installers

• end-user application programs/suites like iWork, LibreOffice, TeX, R, Sage, …

• network monitoring and management

• software development support like compilers, assemblers, profilers, development environments, …

From this point of view, the “operating system” is “everything I don’t have to develop or have developed myself”. This meaning is closer to “distribution”.

An important thing about a distribution is that the various components should be known to work together. For example, there is a programming language for BioInformatics called Darwin that was developed by Gonnet. I have it on a computer running the Darwin kernel (MacOS). But it doesn’t work any more. The mere fact that it is on the machine’s disc does not mean that it is part of a coherent distribution. (It no longer works because the software for executing PowerPC programs that used to be part of MacOS is no longer part of the MacOS distribution.)

Engler and Kaashoek of MIT, in their paper “Exterminate All Operating System Abstractions” (RD02), wrote:

"we define the operating system as [including] any piece of software that the application cannot either change or avoid. User-level device drivers, privileged servers, and kernels are all included by this definition.”

Daniel Ingalls wrote:

“An operating system is a collection of things that don’t fit into a language.

There shouldn’t be one.”

He meant by this that there should be nothing that doesn’t fit smoothly into the language, so that someone using a language needs to be aware of that language and its libraries and not some other thing as well. Smalltalk, Lisp, and Erlang have all been run on “bare metal” and scsh showed that Scheme could be used as an OS command language. Similarly Sing# and Midori showed the possibility of a system that was “C# all the way down”.

From the Engler/Kaashoek and Ingalls point of view, the “operating system” is “all the stuff that gets in between my program and the hardware that gets in the way of efficiency and flexibility”, which is, roughly speaking, the kernel.

4 http://ska-sdp.org/publications/sdp-kernel-workshop

11.4. Minimalist Operating System - Why?

A minimalist operating system is not a goal in itself but a means to an end. The ultimate goal for the SKA is to support radio astronomers in gathering and processing data over multiple decades while adjusting to advances in hardware and changes in computational tasks.

Linux has become very much bigger over the years, to the point where apparently Linus Torvalds himself has called Linux “bloated and huge”.

• Linux 2.0.0 was under a million lines of code.

• Linux 2.6.0 was over 8 million lines.

• Linux 4.13 is over 22 million lines.

Of these, well over half are optional device drivers. It has been estimated that typical desktops “need” only about 5% of the total, so 1–2 million lines seems about right for the SDP.

Reduced size has several advantages:

• **Consistency.** As noted in the previous section, the important thing about a “distribution” is that all the pieces that might be expected to work together are *known* to work together. This means that the more pieces are in the distribution, the more testing is needed, and this scales worse than linearly.

This is one of the key arguments for application-as-container; you can include just the parts of the distribution that you actually need in the container image, and different containers can include different versions of components without “DLL hell”.

• **Reliability.** Code you don’t include can’t cause problems.

• **Stability.** There are new Linux kernel releases every 2-3 months. Release 4.18 came out in August 12, 2018 and already Linus Torvalds published the first Release Candidate for Linux 4.19, two weeks after the Linux 4.18 kernel series was launched. From the perspective of the SKA, two-and-a-bit years is not “Long Term”.

The less that we use, the less we shall be affected by revisions. And we cannot afford to ignore revisions.

• **Portability.** Operating systems have to include a lot of non-portable code. Candidate CPUs for the SDP include x86, ARM, and perhaps Power family members. It is likely that the CPU choice will change depending on price, performance, power use, and availability during construction, so the SDP may end up with a mixture of CPUs. The less code is included in the OS, the less effort it takes to test that it works in the new environment.

• **Performance.** A kernel whose code and data fit comfortably into cache is going to perform better (and be more predictable) than one that spills over into main memory.

The “Kernel Size Tuning Guide”[5] at elinux.org identifies three important “aspects of kernel size”:

1. The size of the kernel image stored on persistent storage. For a distribution this would include the libraries and tools. Data deduplication between containers can amortize this cost.

2. The static size of the kernel in memory. This includes executable code as well as data. For a distribution, it would include the library and other code that needs to be in memory in order for the main application to run.

3. The amount of memory needed dynamically (thread stacks, resource descriptions, buffers, and so on).

For something of the size of the SDP, it may seem absurd to worry about memory, if you save 1 MB on 1000 cores, you have another GB to hold useful data. As Everett Dirksen said, “A billion here, a billion there. Pretty soon, you’re talking about real money.”

The central problem here is that the more services an application makes use of, the more the “operating system” must provide. If a program is allowed to execute

```
system("node foobar.js");
```

then it depends on everything that node.js depends on, which could be pretty much everything.

The same applies if the application either calls or is driven by Python, for example, as the system then needs to provide everything Python might want, and Python tries to provide access to pretty much everything. Not good.

As just one example, “A wormable code-execution bug has lurked in Samba for 7 years”[6]. System that did not install Samba were not exposed to this risk. Researchers found “110,000 devices exposed on the Internet that appeared to run vulnerable versions of Samba. 92,500 of them appeared to run unsupported versions of Samba for which no patch was available.”

The CVE vulnerability database[7] lists a lot of vulnerabilities. The graph on the Linux entry page shows an upwards trend. It is difficult to figure out which vulnerabilities still exist[8], but it is clear that the more Linux, the more problems. In 2017, 26 memory corruption vulnerabilities[9] have been reported, most of them in drivers, and several others in crypto code.

5 http://elinux.org/Kernel_Size_Tuning_Guide

6https://arstechnica.com/information-technology/2017/05/a-wormable-code-execution-bug-has-lurked-in-samba-for-7-years-patch-now/

7 https://www.cvedetails.com/vendor/33/Linux.html

8 https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

9 https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/year-2017/opmemc-1/Linux-Linux-Kernel.html

11.5. Requirements

1. The operating system must run on the SDP hardware.

2. It must run under whatever container system is adopted.

3. The host kernel and the guest kernel should be as small, taking as little “installed memory real estate” as practical.

4. System calls should have low latency.

5. The SDP is not a hard real time system, but it’s not far from one. In order to schedule well, it is desirable that delays should be predictable. And hardware failure be planned for.

6. The OS must support the system-wide work distribution and monitoring system, whatever that turns out to be. Component failure is inevitable. The whole system needs to be resilient: OS must support checkpoint, recovery time, etc.

7. In order to support the system-wide distribution and scheduling of work, it is very desirable that applications should take repeatable times. The SDP has to process some data in pseudo-real time.

As one instance of this, it should be possible for an application to lock its code and data into memory and not page (that is, mlock(2) must work).

8. It must support the communication librar{y,ies} used by the astronomers’ programs, whether that’s MPI, OpenSHMEM, or something else.

9. SDP’s code isn’t intended to be written by astronomers but it’s OS must support the execution of the astronomers’ programs as such.

10. It must be straightforward to install revised or additional astronomy programs.

11. It need not support software development; revised or additional programs should be developed on a separate machine or machines and deployed via the container system.

12. It should support testing. We argue that the CI/CD pipeline should run on the SDP platform to improve test quality and eliminate platform dependent issues as early as possible. (RD05)

13. It must support the storage and communication devices used by the SDP.

14. It should not include support for any others.

15. The system must support multiple cores, multiple GPUs (as in multiple instances, not necessarily multiple times on the same board), and multiple FPGAs (ditto). User code will be parallel and threaded and threads may issue system calls, so the kernel itself needs to be multithreaded.

16. It must support tracing and diagnostics well. A particularly interesting facility here is the “extended Berkeley Packet Filter”[10], which is an abstract machine running in the kernel which is sufficiently constrained that code running in it cannot break the kernel, and which is now JIT

compiled on x86-64, ARM, and PowerPC. This is used for network packet filtering, tracing (monitoring, debugging), I/O latency monitoring, filtering system calls, all sorts of stuff. Systemtap and DTrace remain as alternatives.

17. The OS must have staying power. There are many interesting small kernels, often POSIX-compliant, which have been abandoned. Similarly, many small distributions have come and gone. Longevity is I believe more important than sexiness or minimality.

18. The choice will have to make business sense. Longevity is one aspect of that. Licencing is another. The particular choice will have an acceptable cost of ownership, involving cost of licenses, cost of maintenance, cost of overhead on software development having to develop for a specific platform, cost of maintaining all the other tooling that runs on it. And not just the cost - but the penalties of being locked into a particular solution - lost upgrades, features, architectural options, etc. The budget for the SKA is large, but the hardware will take a lot of it. We should prefer a popular OS that is good enough to a technically superior one that needs a higher maintenance budget.

10 https://qqq.iovisor.org/technology/ebpf

11.6. Some Candidates

• The “Linux Tiny" small kernel project[11] is dead. Its repository was last updated in 2011.

• The Linux Kernel Tinification project[12] has superseded Linux Tiny. It has many sub-projects aimed at making it straightforward to build a small installation of a standard release. “make tinyconfig” makes a small kernel.

For a minimal kernel, a small configuration of a standard kernel could well be the best business choice. Amongst other things, it entails no modifications to the source code.

• Tiny Core Linux[13] is “a nomadic ultra small graphical desktop operating system capable of booting from cdrom, pendrive, or frugally from a hard drive.” (Having used UNIX v7 on a PDP-11/60 with 256 kB of memory, I do not regard a kernel that wants 48MB of RAM as tiny in any interesting sense.) It typically sees about 3 releases per year. At the time of writing, the latest version was 9.0, released in February 2018.

It is one of at least 30 Linux variants intended to be able to run entirely from RAM, and is the basis of at least two others, including Damn Small Linux, which seems to have been abandoned since 2012. (Unlike Tiny Core Linux.)

“Core (11MB) is simply the kernel + core.gz — this is the foundation for user created desktops, servers, or appliances. …Command line tools are provided. TinyCore is Core + Xvesa.tcz + Xprogs.tcz + aterm.tcz + fltk-1.3.tcz + flwm.tcz + wbar.tcz …16MB.”

Most of the extra features in TinyCore are directly concerned with supporting X11, a window manager, and a widget set; these are of no use to a compute node. Core (also known as Microcore) seems closer to our needs.

For what it’s worth, the TinyCore .iso file was about 16 MB. Loading it into VirtualBox, it started faster than any other OS I’ve used in recent years, and the 80-odd processes it was running took a total of 50 MB of memory.

• Many of these “run in RAM” systems focus on running an entire desktop and toolset in RAM rather than minimising the kernel.[14] Since compute nodes do not need to support a graphic desktop or office suite, these systems include components we do not want. Conversely, they omit libraries of great interest to us.

• Jailhouse is a hypervisor intended to run real-time and/or safety tasks on asymmetric multicore platforms and also one or more Linux instances. It really is tiny, but does not itself provide POSIX services.

• OpenWrt[15] “is a highly extensible GNU/Linux distribution for embedded devices (typically wireless routers) …using …a recent Linux kernel”. It is actively maintained with a release in 2016.

• There is a general “Just enough Operating System” (JeOS) idea with instances based on SUSE[16] and Ubuntu[17] amongst others. One member of this family is

• CoreOS, “a Linux streamlined to support server workloads” where “containers are the *only* way to run applications”. It is now known as “Container Linux”[18][19]. According to Wikipedia, it “is an open-source lightweight operating system based on the Linux kernel and designed for providing infrastructure to clustered deployments, while focusing on automation, ease of application deployment, security, reliability and scalability. As an operating system, Container Linux provides only the minimal functionality required for deploying applications inside software containers, together with built-in mechanisms for service discovery and configuration sharing.”.

This brings out an important distinction which also needs clarifying: that between an operating system that runs Docker or some other container environment (a “host OS”) , and one which runs *inside* such an environment (a “guest OS”), and indeed one that does both.

It appears that CoreOS is still only available for x86-64, which might be a problem if the SDP needs to move to other hardware.

• Alpine Linux[20]. “Alpine Linux is an independent, non-commercial, general purpose Linux distribution designed for power users who appreciate security, simplicity and resource efficiency. [It is] smaller and more resource efficient than traditional GNU/Linux distributions. A container requires no more than 8 MB and a minimal installation to disk requires around 130 MB of storage. Not only do you get a fully-fledged Linux environment but a large selection of packages from the repository.” It “can be installed as a run-from-RAM distribution” and has “A [security] hardened kernel”.

A new version has appeared roughly every six months with the latest being 3.8.0, released in June 2018. There are Standard, Extended, Vanilla, and Virtual variants. The last of these includes a “[s]limmed down kernel [o]ptimized for virtual systems.”

Alpine Linux is quite popular amongst container developers because they pack down well. It is also based on musl (instead of libc) -an example of less packages available because not every software project targets musl.

• SUSE JeOS “trims unnecessary components down to keep the size small (about 300 MB). …It is ready-to-run in your hypervisors”.

It is basically a de-bloated SUSE Enterprise Server and is a commercial product. It is not available by itself but comes with a SUSE Enterprise Server subscription, which is supported on

x86, x86-64, POWER, z/Series, and after a fashion on ARM. “A physical server with 1 or 2 populated sockets” needs a USD 799/year subscription (USD 2160/3 years). The SDP has been estimated at “2 cards per node”[21] (RD03) so that would come to over USD 7 million per year. What kind of deal they might cut for the SDP is anyone’s guess.

• There used to be an Ubuntu JeOS, which was “merged into the Server CD and …available as an option at installation time”. Ubuntu Server has worked on x86, x86-64, SPARC, ARM v7, ARM64, Power8, and z/Series, although Ubuntu Server 16.04.3 LTS is described as “64 bit only”. You can apparently choose a JeOS equivalent by selecting “Install a minimal virtual machine” at CD installation time or using vmbuilder.

Canonical list as advantages of this approach that “Updates, whether for security or enhancement reasons, will be limited to the bare minimum of what is required in their specific environment. In turn, users deploying virtual appliances built on top of JeOS will have to go through fewer updates and therefore less maintenance than they would have had to with a standard full installation of a server.”

The x86-64 version of Ubuntu Server is a free download.

• “Ubuntu Core[22] is a tiny, transactional version of Ubuntu for IoT devices and large container deployments. …[That] uses the same kernel, libraries and system software as classic Ubuntu. …It is purposely lightweight and transactionally updated system, with security at its heart. …[It] supports an unrivalled range of SoCs and single-board computers, from the 32-bit ARM Raspberry Pi (2 and 3) and the 64-bit ARM Qualcomm Dragonboard to Intel’s full range of IoT SoCs.

It is a 350 MB distribution, and is Open Source with commercial support available.

• LinuxKit[23] is a newcomer to this space but looks promising. It is described as “a toolkit for building secure, lean, and portable Linux subsystems …that can provide Linux container functionality as a component of a container platform.”

LinuxKit includes the tooling to allow building custom Linux subsystems that only include exactly the components the runtime platform requires. All system services are containers that can be replaced, and everything that is not required can be removed. All components can be substituted with ones that match specific needs.

Because LinuxKit is container-native, it has a very minimal size — 35MB with a very minimal boot time.

According to Docker’s security director “LinuxKit’s roots are in Alpine. A stronger Alpine is a stronger LinuxKit. We’ll continue to invest in Alpine.” Apparently “LinuxKit is meant to be even more flexible and easier to customize” than Alpine.

This is very promising indeed. (The 35 MB figure appears to be the core distribution, not just the kernel.)

LinuxKit is a target for Docker Swarm and Kubernetes -basically boot to cluster with the aim of having everything running in containers including OS services and housekeeping daemons. As such, it is available free as Open Source, but there is also commercial support available.

The top four candidates appear to be

• Alpine Linux : goes hand in hand with Docker, proven track record

• LinuxKit : Alpine's younger brother, easier to shrink further

• Ubuntu Core.

• Ubuntu Server configured for a minimal vm.

They are all known to work well in a container environment and are actively maintained by credible organisations that are not likely to disappear overnight.

SUSE dropped off the list because of the repeating licence fee.

11 http://elinux.org/Linux_Tiny

12 https://tiny.wiki.kernel.org

13 tinycorelinux.net

14 I have just tried out Minimal Linux Live. In my browser. Running on a PC emulator written in JavaScript. Insane!

15 wiki.openwrt.org/doc/start

16 https://www.suse.com/products/server/jeos/

17 https://en.wikipedia.org/wiki/Ubuntu_JeOS

18 https://coreos.com/os/docs/latest

19 https://en.wikipedia.org/wiki/Container_Linux_by_CoreOS

20 https://www.alpinelinux.org/about/

21 https://science.nrao.edu/science/event/RALSST2013/lsst2013-presentations/nikolic-presentation

22 https://www.ubuntu.com/core

23 https://blog.docker.com/2017/04/introducing-linuxkit-container-os-toolkit/

12. Survey Linux Performance Monitoring Systems

Performance measurement is important for both development and maintenance of HPC code. Especially in an environment where application repeatability is expected to be an issue, measurement of what is happening both in the applications and in the kernel, is important. During operations, it is also important that measurement not incur great overheads.

These really are not new observations, and there are several performance management toolkits in existence for Linux.

12.1. Summary

The Extended Berkeley Packet Filter and the services built on it, especially bcc, seem to be the best choice for performance diagnosis.

For routine monitoring, collectd seems like a good choice. It “is able to handle any number of hosts, from one to several thousand.”

OpenStack Telemetry project aim is “to reliably collect data on the utilization of the physical and virtual resources comprising deployed clouds, persist these data for subsequent retrieval and analysis, and trigger actions when defined criteria are met.” It includes a data collection service (which can interoperate with collectd), an alarm service, and time-series database and resource indexing service, and an event metadata indexing service.

12.2. Scope

Performance monitoring is an issue during

- Development
- Maintenance (checking that software changes have not made performance worse)
- Tuning (adapting to new hardware or support software configuration)
- Operations (keeping track of whether some kind of performance issue exists or is developing)
- Diagnosis (tracking down the cause of a performance problem)

Monitoring performance involves the following steps:

- Selecting events or quantities to monitor.
- Monitoring them.
- Reporting them to higher software levels.
- Filtering, transforming, combining, and aggregating them.
- Recording measurements in (semi-)persistent storage for later analysis, and/or
- Transmitting measurements for central analysis.
- Comparing present measurements with historic ones.
- Displaying information in forms suitable for human comprehension.
- Letting human analysts revise the process as they proceed through diagnosis.

We have the following concerns:

- Measurability: which events and quantities can be directly measured, which events and quantities can be computed or estimated from direct measurements, and which events and quantities cannot be reliably measured at all. Other things being equal, the more a framework can measure, the better.
- Intrusiveness: the amount of effort required to enable measurements, *e.g.*, is it a matter of flicking a switch or is it necessary to modify source code and recompile?
- Overheads: how much extra CPU time, how much extra memory, how much memory bandwidth, how much I/O bandwidth and so on are taking up by the measurement system itself?
- Data volumes: even a single CPU core can generate events at an overwhelming rate; the SDP will be able to generate events at a humanly incomprehensible rate. The SDP

needs to be able to do *some* work while measurement is happening, or what's to measure? It is not enough to *measure* less, because that reduces the useful information. The answer is to *filter*, *aggregate*, and *compress* at every level. For example, if measuring kernel events, it is desirable to avoid frequent context switching.

- Interoperability: we wish to decouple the low level measurement from the high level analytic and visualisation activities. It would be useful to use tools like RRDtool, and collectd. In particular, there are Internet standards for performance data protocols: SNMP (particularly SNMPv3, see next bullet point) and syslog. Note though that both SNMP and syslog provide *frameworks*, and have nothing to say about what the measurements might be, so are not enough by themselves to ensure interoperability.
- Security and integrity: can agents extract information they are not authorised for, and can they corrupt information or otherwise break the system? Security is probably not the major issue for the SKA, but integrity certainly is. Instrumenting a component or activating existing instrumentation must not be able to change data, disrupt flow control, or deny service by running too long.

We also have the issue of licensing and training. Is(are) the chosen framework(s) available as Open Source? How hard is it going to be to find people who can use it(them)? Are the frameworks well documented?

12.3. Tools

For the interested reader, the following tools and methods are presented in Appendix A: Visualisation, Command-Line tools, Tracing programs, Interposition, Low level facilities in Linux, DTrace, SystemTap, and Extended Berkeley Packet Filter.

13. Measuring system overheads in and out of containers.

Can a node be started rapidly? Can a function-as-a-service container be started up, do its work, and be shut down rapidly? Are there any overheads from running in a (LXD or Docker) container? How unpredictable are operating system services such as I/O and inter-process communication? How big are the advantages of using a stripped-down Linux compared with a full Linux, if any? Can the use of SSDs solve the startup time problem or is something more needed?

In order to address these questions, we are going to compare inter-process communication in Linux under several conditions. To keep things simple we shall just use two processes.

13.1. Where are the processes?

There are four fairly obvious ways to organise the processes.

- In the same hyperthread (necessarily on the same core).

- In different hyperthreads on the same core.
- In different cores in the same socket.
- In different sockets.

We are developing these benchmarks on a 4-core AMD chip, which has only one socket and does not support hyper-threading.

```
% lscpu
Architecture:          x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list:   0-3
Thread(s) per core:   1
Core(s) per socket:   4
Socket(s):           1
...
```

```
% lstopo-no-graphics
Machine (6937MB)
  Package L#0 + L2 L#0 (2048KB)
    L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)
    L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#1)
    L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2 + PU L#2 (P#2)
    L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3 + PU L#3 (P#3)
  HostBridge L#0
    PCI 1002:9850
      GPU L#0 "renderD128"
      GPU L#1 "card0"
      GPU L#2 "controlD64"
    PCIBridge
      PCI 8086:3165
        Net L#3 "wlo1"
    PCIBridge
      PCI 10ec:8136
        Net L#4 "enp3s0"
    PCI 1022:7804
      Block(Disk) L#5 "sda"
      Block(Removable Media Device) L#6 "sr0"
```

In the output of lstopo, note that for each physical CPU (P#*n*) there is one logical CPU (L#*n*). This is how we know there is no hyperthreading.

That leaves us with two setups to test on the machine, three if we add “let the operating system decide”.

13.2. How do they communicate?

There are two major alternatives that we can try:

- *message passing*, where information is passed through a special communication channel. This has several varieties:
    - System V Message Queues
    - POSIX Message Queues
    - Pipes
    - Named Pipes

    - UNIX-domain Sockets
    - TCP sockets at “localhost”.
- We expect System V and POSIX message queues to have similar performance. We expect pipes, named pipes, and UNIX-domain sockets to have similar performance.
- *shared memory*, where we set up a “bounded buffer” in shared memory, using semaphores for synchronisation. There are again several ways to do this:
    - System V shared memory
    - POSIX shared memory
    - a memory-mapped file.
- We expect all of these to have the same performance once set up.

13.3. How big are the messages?

On the machine I'm using,

```
% getconf PIPE_BUF .
4096
```

which we'll take as the upper bound on queued messages.

13.4. How is the timing done?

The basic schema is this:

1. Master process creates child process.
2. Master process sends WARM_UP (4) messages to the child, then repeats receive, send RUN_LENGTH times, then receives WARM_UP messages at the end.
3. Child process repeats receive, send RUN_LENGTH + WARM_UP times.

Whenever a message is sent or received, a nanosecond timestamp is added to it. The program reports the time between master deciding to send and slave completing receipt (mtos), the time between slave deciding to send a reply and master completing receipt (stom), and the total round trip time (total) for each message. The times are logged to standard output for analysis using R.

13.5. Preliminary results

Please refer to Appendix B for tables and results -some unexpected, i.e.

- System V and Posix message queues are significantly different
- While we expected that UNIX-domain sockets would be cheaper than TCP to 127.0.0.1, we had not expected the magnitude of the difference

14. Security at OS level in next generation HPC systems: possible applications to the SDP

In some cases, HPC systems run full operating systems for compute nodes, while in other cases are running lighter-weight Linux compatible OS or entirely custom ones. How applicable are conventional solutions to existing and emerging HPC environments, and in particular to the SDP environments (considering it as an instrument with well-defined applications and with a “pre-qualified” user base)? Is the notion of "containerisation" a key benefit to security while contributing to system robustness? How can we minimise possible attacks and vulnerabilities in the SKA?

In this section we analyse modern systems and how their OSs face these challenges , and look at the impacts of these choices on the SDLC - how does it impact on the ability of developers to participate, costs associated with maintaining a custom kernel and dependent tools, costs associated with a heterogeneous environment and the hardware refresh cycle. In sum, custom vs commodity.

14.1. Security in the Kernel: Do we have a problem?

The book “Writing Secure Code”, written by Michael Howard and David LeBlanc and published by Microsoft Press in 2002, has a forward by Brian Valentine, then a Senior Vice President at Microsoft. In it, he wrote

> We set up a Windows 2000 Web server called “Windows2000test.com”, put it out there, and waited to see what happened. We made no announcement of any kind; we didn't call any attention to it in any way whatsoever. Within a couple of hours, hundreds of people were already trying to hack it. Within days, tens of thousands of people were hammering away.

About 8 years ago one of our students had a similar experience. He set up a server in his hall of residence. Within *minutes* of being connected to the web, it was being port-scanned. It would be ridiculous to suppose that Black Hats were specifically targeting a student box; they had presumably probed a range of IP addresses and stumbled across his by accident. Valentine's case may well have been the result of Black Hats probing a range of IP addresses owned by Microsoft.

We have to assume that any machine with an Internet presence will be attacked, not because it is of specific interest to the Black Hats, but simply because it is there.

14.2. An immediate observation.

“It is estimated that the entire IPv4 based Internet can be scanned in about 10 hours” [IPv6 Security, Hong Kong] because of its 32-bit address space, which is sufficiently close to running out that blind attacks are worthwhile because they are likely to find *something*.

There is a partial answer to that, which is to use only IPv6. The IPv6 address space is 128 bits. This reduces the effectiveness of *blind* scanning, but does not mean firewalls can be discarded.

Public signature keys can be bound to IPv6 addresses, which “makes spoofing attacks much harder”.

IP security is not optional in IPv6. It “provides authenticity, integrity, confidentiality and access control to each IP packet”.

IPv6 only really protects at the transport level; buffer overflows, viruses and other malware, and password guessing are still issues.

However, it's fair to say that one important security feature we want from a kernel is mature support for IPv6. Linux has had official support for IPv6 since the 2.6 kernel in about 2003. These days it is enabled by default.

14.3. What do we mean by security?

Security can mean many things. Information Security's primary focus is the balanced protection of the Confidentiality, Integrity and Availability of data (also known as the CIA triad) while maintaining a focus on efficient policy implementation, all without hampering organisation productivity.

Security in the context of the SDP includes, without being limited to:

- The code that is running on the SDP is the code that is in general allowed to run.
- The code that is running is running at the behest of someone who is authorised to have it run.
- Information is not accidentally lost.
- Information is not vandalised.
- Information is not deliberately corrupted to be plausible but misleading.
- Information is not withheld from people who have a right to it.
- Information is not divulged to people who do not have a right to it, although in the context of radio astronomy this is unlikely to be a major concern.

There are a number of core principles for secure systems:

- **Scrutability.** If you cannot inspect the system, you cannot trust *it* but have to trust the provider, and you had better have a contract that makes them liable for breaches of system security. This is one argument for Open Source, which of course means that if you are not duly diligent in inspecting the code, you have little recourse because of the legal principle of Assumption of Risk.
- **Security Controls.** Safeguards to avoid, detect, counteract or minimise security risks. Can be classified by several criteria i.e. according to the time that they act relative to a security incident in: preventive controls (lower probability), detective controls (raise alarm) and corrective controls (lower impact).
- **Modularity and Encapsulation.** This limits the ability of a security flaw to propagate. This could be hardware, with different subsystems running on different devices, with narrow hardware interfaces; software, with things running in different processes; or high level languages offering isolation, as in Midori (M#) and Singularity (Sing#) or the much older Burroughs MCP.

- **The Principle of Least Authority:** each component should have just enough authority/privilege to do its job and no more. This is something of an ideal abstraction, but Linux capabilities (capabilities(7)) and Solaris Rôle-Based Access Control help in this regard.
- **The Reference Monitor concept.** This idea was first formulated in the context of Multics, and it is discussed in some of the Multics paper and in the 4-volume report on Multics security from the USAF's Electronic Design Systems. The key idea is that every reference to some resource you want to be secure must be monitored by some combination of hardware and software to ensure that the reference is authorised. This is the idea behind the SecurityManager class in Java. Reference monitors must be
  - small enough to test and inspect *thoroughly*
  - impossible to tamper with
  - impossible to bypass.
- The last point is why in the presence of JNI, Java security managers don't really count, and ultimately, why systrace doesn't really count either.

14.4. There can be no secure kernels on modern x86

Here's a diagram of the kind of setup we're talking about.

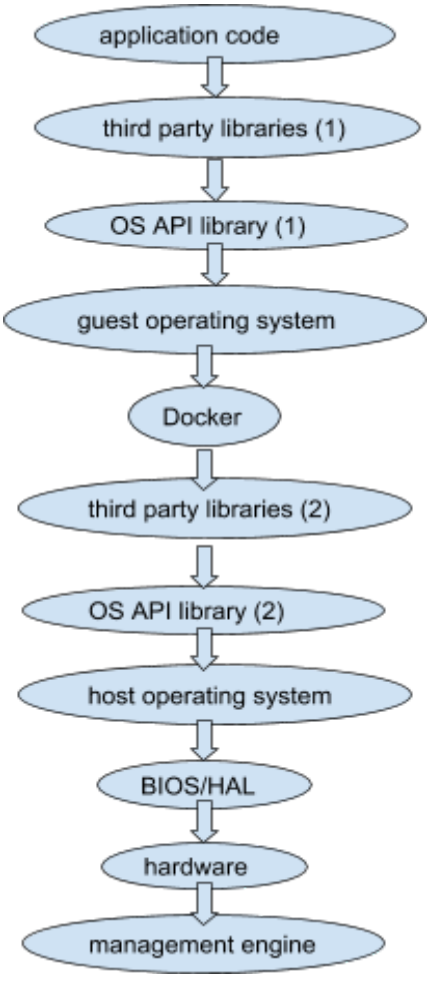

Figure 1 -Security from the bottom up - IME

No layer can be considered secure unless all the layers below it are secure. The problem here is the management engine.

There is no disputing that the management engine (IME) serves a useful purpose. Nor is such a separate management computer a new thing. Burroughs B6700s came with a "Maintenance Diagnostic Processor" that could take over the machine. DEC VAXes came with a PDP-11 that booted the VAX and could install firmware patches.

The fundamental issue is that the management engine in modern x86 chips has total access to all memory and all devices, including the network interface card, and is operating wherever the machine is connected to power, even if it is notionally turned off. Not only that, it is essential to turning the x86 itself on. It is possible to disable *some* modules of the Intel management engine, but disable too much of it, and your x86 turns into a brick after 30 minutes.

This is enough of a worry that a company called System76 are now selling laptops with just enough of the IME left to let them function, and offering upgrades to existing customers.

The current generation of IMEs are apparently based on an x86 CPU of some kind, running a hacked version of MINIX 3. As far as we have been able to find, they still include a full TCP/IP stack, a web server, and may still include Java.

Now MINIX 3 was a good choice; the kernel is just 13,000 lines of code, the rest of the system is highly modular, and the overheads of a message-passing microkernel are tolerable in a gadget that just controls another computer rather than doing most of the heavy lifting itself. Indeed, since MINIX 3 was the result of a 5 year EU project to make MINIX more reliable and secure, it may well be more secure than alternatives. But only *may* be.

The issue is *scrutability*. Nobody outside Intel knows what changes they made to MINIX 3, nor is any part of the software running on the IME available for inspection. This is a device which has *total* authority over everything (defenestrating the Principle of Least Authority), and can freely tamper with and bypass any higher-level reference monitoring.

We can't switch to AMD instead because the Ryzen generation of chips have a similar feature about which even less is known, and which therefore pose a higher risk.

After the first draft of this section was written, the Meltdown and Spectre vulnerabilities became generally known. The issues are bounds check bypass and branch target injection (Spectre) and rogue data cache load (Meltdown). Unpleasantly, the eBPF feature, which we recommended in Section 6.3 (and Appendix A.5), is a good target for such attacks.

According to engadget, Intel are promising chips without these vulnerabilities some time this year (2018). However, considering how long it took before these vulnerabilities were discovered, and the long sad history of rushed security patches, and the fact that the IME will still be present, it's fairly safe to say this won't be the last we hear of chip-level security weaknesses.

There will be a lot of HPC data centres that will make the choice not to implement the security fixes because of the loss of performance, and all that compute will be vulnerable. A well known radio-telescope network took the “bastion approach” and they “got owned”. What happens when you get owned and you don't even know it because the exploit is undetectable - other than your work throughput falling off.

The major problem at the moment is that the software measures taken to mitigate the hardware problem reduce performance. The Linux KAISER patch providing Kernel Page Table Isolation has been measured by others to reduce performance by 0.14% to 39.2% depending on the benchmark. Benchmarks with Tensorflow got 5% to 10% hits. Benchmarks with SciKit-learn and Pandas got 5% to 15%. Benchmarks with SciPy/NumPy got 0.4% to 37%.

14.5. Minimum recommendations

The SDP will probably rely on Docker provisioning for upgrades and so to a large extent won't *need* the management services provided by the IME, so the IME should be disabled to the extent practical (which is not 100%).

The SDP's connections with the outside world should go through a firewall that allows the minimal possible range of services in either direction, and should run on a different architecture that lacks an IME analogue.

Communication *within* the SDP needs to use a small number of known and carefully audited protocols.

Computation within the SDP will use a fixed set of programs, so we can probably get away with disabling KPTI internally. (relief!) But gateway machines should enable it.

14.6. Kernel security mechanisms

14.6.1. Multiple user accounts

Let's not forget important techniques just because they are old. The idea of having different users with different rights is old in computing. There are two issues here: authentication and capabilities.

Authentication means the system needing good reason to believe that you are who you say you are. The classic UNIX authentication mechanism is user+group, with group memberships determined by user and user authentication by a password. The Pluggable Authentication Modules facility — PAM(7) — lets a system administrator plug in things like Kerberos and LDAP.

Capabilities are what you are allowed to do.

Traditionally in UNIX your ability to do things was governed by your user id, group ids, the owner of the file system object (or System V IPC object) you were trying to operate on, the group of that object, and the permission bits of that object. Or, if you were the root user, you could do anything to anything.

The POSIX.1e draft broke this into several distinct capabilities(7), which can be separately granted. This draft was adopted by Linux. The BSDs have rejected it, alleging that it poses risks of its own, but we don't know yet what those are. There was a TrustedBSD project to add this to FreeBSD, but it's dead.

This feature should be used to approximate the Principle of Least Privilege.

14.7. Other features

The following features

- Address Space Layout Randomisation
- Automatic Security Updates
- File system Capabilities
- File system Encryption
- Heap protection
- Kernel Livepatches
- Linux Security Modules and Mandatory Access Controls
- No Open Ports
- Password Hashing
- SECCOMP
- Stack Protector
- SYN cookies
- Uncomplicated Firewall

are described in detail in Appendix C, with considerations relevant to the SDP -like Python be built with the Stack Protector feature enabled (section 19.11)

14.8. Do we need a Security Policy within the SDP?

We were working on "Security at OS level -applications to SDP" (TSK-1987) in the second half of 2017 -coincidental with the first release of Meltdown and Spectre vulnerabilities. Our report attracted considerable interest -more than 20 comments, among them the "meta question": Do we need a Security Policy within the SDP?

The argument was that the SDP is not a general purpose cluster, but rather a tightly controlled appliance with very few, if any, physical users. It was debated if the recommendations above are overly restrictive considering the way that SDP is intended to work. This is a relevant comment, which why we discuss it in SDP Memo 064 "Security for the SDP - Architecture Considerations" where we elaborate further on basic premises such as

a) Good security measures protect against ACCIDENT as well as attack. And accidents are common.
b) If the SKA is accessible through the internet, it WILL be attacked just because it is there. EVERYTHING on the internet is probed for vulnerabilities. However, the SKA will be a very tempting target. Sure, the information on it is probably not going to be tempting to thieves. But the SDP itself, as a computing engine, will be tempting.
c) The purpose of the task is not to set a policy (while we may make recommendations, they are just personal opinions at this stage) but to discuss mechanisms and what they can and cannot achieve, so that an informed judgement can be made later about what policies and mechanisms are acceptable, considering the price of adopting them and the risks of not adopting them.

15. Conclusion.

Until recently established software development techniques have been assumed to be appropriate for all scales of software use. This is not the case for the SKA. The way the development of systems and services need to be approached for Exascale Computing is entirely different to that for the common and small use case of today. And for future HPC systems -as we asked in the Introduction, will the OS or runtime system bear the brunt of the adaptation, or will we be able to insist on changes in the technologies, applications and environment?

It's said that the "operating system" is "all the stuff that gets in between my program and the hardware that gets in the way of efficiency and flexibility", which is, roughly speaking, the kernel.

In this document we propose a minimalist operating system not as a goal in itself but as a means to an end. The ultimate goal for the SKA is to support radio astronomers in gathering and processing data over multiple decades while adjusting to advances in hardware and changes in computational tasks.

About 1–2 million lines seems about right for a minimalist OS for the SDP. Reduced size has several advantages offering consistency, reliability, stability, portability and performance: A kernel whose code and data fit comfortably into cache is going to perform better (and be more predictable) than one that spills over into main memory.

The central problem here is that the more services an application makes use of, the more the "operating system" must provide. If a program is allowed to execute system("node foobar.js"); then it depends on everything that node.js depends on, which could be pretty much everything.

The same applies if the application either calls or is driven by Python, for example, as the system then needs to provide everything Python might want, and Python tries to provide access to pretty much everything. This is not good. We suggest that Python be built with the Stack Protector feature enabled (check Appendix C – Security in the Kernel).

Finally a word about Security:
"...we believe it's not enough to build something and try to make it secure after the fact. Security should be fundamental to all design, not bolted on to an old paradigm. That's why we build security through progressive layers that deliver true defense in depth, meaning our cloud infrastructure doesn't rely on any one technology to make it secure". (Urs Hölzle, Senior Vice President, Technical Infrastructure, Google, March 2018) (RD04).

Our minimum recommendations for Security at OS level are (check section 15.5)
- The SDP will probably rely on Docker provisioning for upgrades and so to a large extent won't *need* the management services provided by the IME, so the IME should be disabled to the extent practical (which is not 100%).

- The SDP's connections with the outside world should go through a firewall that allows the minimal possible range of services in either direction, and should run on a different architecture that lacks an IME analogue.
- Communication *within* the SDP needs to use a small number of known and carefully audited protocols.
- Computation within the SDP will use a fixed set of programs, so we can probably get away with disabling KPTI internally. But gateway machines should enable it.

16. Appendix A - Linux Performance Monitoring Systems - Tools

This Appendix should be read as continuation of Section 12.

16.1. Visualisation

This document does not discuss performance visualisation tools such as flame graphs or the Eclipse plugin that is said to have replaced the LTTng Viewer.

Most of the Linux performance visualisation stuff that we looked at is concerned with a *single* machine such as a file server or web server, or possibly a cloud system where each VM might be monitored or the hypervisor might be, but the parts are not working together as a coordinated whole.

Providing meaningful summaries and visualisations for a system with myriads of cores is a challenge. But it is a challenge that others are facing as well. Using OpenStack Telemetry looks to be a good way to let them do the work. In particular, there is some support for drraw, a Web interface to rrdtool, and for graphite.

16.2. Command-Line tools

You should probably read Brendan Gregg's Linux Performance web page instead of this. To get a high level view of system resource usage quickly he recommends using the following commands:

```
uptime
dmesg | tail
vmstat 1
mpstat -P ALL 1
pidstat 1
iostat -xz 1
free -m
sar -n DEV 1
sar -n TCP,ETCP 1
top
```

This is of course a recipe for *manual* examination/diagnosis, not for routine monitoring.

Gregg has some very useful diagrams where you can learn about the existence of the intel_gpu_top(1), intel_gpu_time(1), and intel_gpu_frequency(1), programs a super-user can use to measure an intel integrated GPU, and that led to nvidia-smi(1), aticonfig(1), radeontop(1), and radeon-profile (app).

The most notable command-line performance tool in the Linux world is perf, which has been described as “the baseline tool for every performance related analysis on Linux”. It is a “kernel-based subsystem that provide[s] a framework for ... performance analysis. It covers

hardware level (CPU/PMU, Performance Monitoring Unit) features and software features (software counters, tracepoints) as well." It can monitor a specified set of cores or a specified container.

Unlike truss, ktrace, strace, and so on, perf(1) needs special permission to run, because it is able to look into the kernel. For example,

```
% perf stat astc -E -d -m32 -mObject.new xxx.st
Error:
You may not have permission to collect stats.
Consider tweaking /proc/sys/kernel/perf_event_paranoid:
 -1 - Not paranoid at all
  0 - Disallow raw tracepoint access for unpriv
  1 - Disallow cpu events for unpriv
  2 - Disallow kernel profiling for unpriv
```

On the Linux system tried, perf list reported

```
 stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]
 ref-cycles                                         [Hardware event]

 cpu-clock                                          [Software event]
 task-clock                                         [Software event]
 page-faults OR faults                              [Software event]
 context-switches OR cs                             [Software event]
 cpu-migrations OR migrations                       [Software event]
 minor-faults                                       [Software event]
 major-faults                                       [Software event]
 alignment-faults                                   [Software event]
 emulation-faults                                   [Software event]
 dummy                                              [Software event]

 L1-dcache-loads                                    [Hardware cache event]
 L1-dcache-load-misses                              [Hardware cache event]
 L1-dcache-stores                                   [Hardware cache event]
 L1-dcache-store-misses                             [Hardware cache event]
 L1-dcache-prefetch-misses                          [Hardware cache event]
 L1-icache-load-misses                              [Hardware cache event]
 LLC-loads                                          [Hardware cache event]
 LLC-stores                                         [Hardware cache event]
 LLC-prefetches                                     [Hardware cache event]
 dTLB-loads                                         [Hardware cache event]
 dTLB-load-misses                                   [Hardware cache event]
 dTLB-stores                                        [Hardware cache event]
 dTLB-store-misses                                  [Hardware cache event]
 iTLB-loads                                         [Hardware cache event]
 iTLB-load-misses                                   [Hardware cache event]
 branch-loads                                       [Hardware cache event]
 branch-load-misses                                 [Hardware cache event]

 mem-loads OR cpu/mem-loads/                        [Kernel PMU event]
 mem-stores OR cpu/mem-stores/                      [Kernel PMU event]
```

and no tracepoints. This was a somewhat elderly system running a 2.6.32 kernel. The latest release of Ubuntu, 18.04, uses the 4.15 kernel.

16.3. Tracing programs

16.3.1. truss

This is a program that came from System V Release 4 and is present in Solaris 10 and later, including derivatives such as OpenIndiana. It executes a given command and "produces a trace of the system calls it performs, the signals it receives, and the machine faults it incurs." Events can but need not be timestamped. In particular, it is possible to get the time spent in each system call. The program can also intercept and trace existing processes. It is possible to control the set of traced system calls, signals, and faults precisely, but it is not possible to trace anything else.

There is a companion program sotruss(1) which produces a trace of calls to functions in shared libraries. Since the system call functions are in a shared library, it can do some of what truss(1) does, but does not handle signals or faults and does not offer timestamps.

A truss(1) utility is provided in AIX, Solaris, UNIXWare, and FreeBSD.

Since 10.5, Mac OS X has provided dtruss(1), a version of truss written using the DTrace framework. This was a matter of considerable annoyance because it replaced a tool used often (ktrace) with an unusable tool: "Since this uses DTrace, only users with root privileges can run this command."

16.3.2. ktrace

This is a program that came with Mac OS X up to 10.4 and some of the BSDs, including FreeBSD. It "enables kernel trace logging for the specified processes". The events it can report include system calls, page faults, I/O, namei translations, capability check failures, signal processing, context switches, sysctl(3) requests, "userland traces" and "various structures".

The first thing the manual page teaches you is how to turn it off because of the large amounts of data it can generate. For the same reason, trace data is recorded in a binary format for off-line viewing using kdump(1). And for the same volume-related reason, much processing is done inside the kernel, to avoid frequent context switches, and this requires the kernel to be built with the KTRACE option. Details are in ktrace(2). User programs can be instrumented by adding calls to utrace(2) .

The analogue of sotruss is ltrace(1), which traces calls to dynamically loaded library functions. It can report times for library calls, to microsecond resolution, and this includes system calls.

A ktrace(1) utility is provided in FreeBSD, NetBSD, OpenBSD, and used to be provided in Mac OS X.

16.3.3. strace

This program was originally written for SunOS, ported to System V Release 4 and Solaris, then ported to Linux, and now exclusive to Linux. It was inspired by trace(1), the precursor to truss(1), and has similar features and similar limitations.

The current version, 4.19, can be found on SourceForge.

A Linux version of ltrace(1) is also available. The documentation states that with both strace and ltrace, "a traced process runs slowly".

16.4. Interposition

One method of instrumenting user-level programs is interposition, where a non-standard dynamic library is “interposed” between the program and the dynamic libraries it *thinks* it is calling. This technique has been well supported at least since System V Release 4 and is still usable in Solaris and Linux. It required rather more work in Mac OS X, but was possible and may still be so.

The interposed library can do anything it wants to with a library call (including a system call). It may rewrite file names, enforce extra checks and fail calls that would not normally have calls, or even disrupt the program completely. It does not require any special privilege.

One problem with interposition is that C and Fortran compilers are allowed to know what standard functions do and to optimise them away, so that a call in the source code might never appear in a trace. It is also the case that “a system call [may] differ from the documented behaviour or have a different name”.

The fundamental problem with interposition is that the only events that can be detected are calls to *dynamically linked* library functions and returns from them. And of course calls to statically linked functions will not appear at all, still less inlined functions.

It is, however, possible for an interposition library to measure quantities like the size of tables, and to report anything that can be measured through system interfaces such as getrusage() and sysctl(3).

16.5. Low level facilities in Linux.

The following are part of the Linux kernel. Things like SystemTap rely on them. The facilities as such are not likely to go away, but the interfaces may be (and one of them is) unstable.

16.5.1. kprobes

Kprobes is a toolkit for dynamically instrumenting the Linux kernel. You can bind your own handlers, running inside the kernel, to specific kernel locations.

16.5.2. uprobes

Uprobes are described in Ptrace, Utrace, Uprobes: Lightweight, Dynmic Tracing of User Apps by Keniston, Mavinakayanahalli, Panchamukhi, and Prasad of IBM. Basically, there is a services called utrace that a kernel model can use to “track interesting events in traced processes”. The venerable ptrace interface was reimplemented to use this.

You can control threads and report events including system call and other function entry/exit, signals, and in principle anything you could put a breakpoint on. However, this requires writing and loading a kernel module.

16.5.3. perf_event

perf_event is a Linux system call. The manual “page” for it is about 44 printed pages. The basic idea is that perf_event returns a new file description which you can then read() events from. This

can do a heck of a lot, but Weaver complains that “In practice the ABI is frequently broken, and as long as the userspace perf tool still works none of the kernel developers seem to notice or care.” Weaver is the author of the Performance Application Programming Interface.

16.5.4. tracepoints

tracepoints are a way of marking an interesting point in the kernel that a probe may be attached to. The overhead of a deactivated tracepoint is low but non-zero. This mechanism is found in other operating systems, not just Linux. DTrace on Solaris relies on it, for example. The point is that things other than function calls can be probed, but you have to program tracepoints in explicitly.

16.6. DTrace

DTrace was developed at Sun Microsystems. Measurement scripts are written in a special-purpose and carefully limited language called “D” and run in the kernel. At the time it was introduced, there was nothing to match its scope. There are “tens of thousands” of predefined probes that can be activated, with no overhead when they are not.

Erlang, Java, JavaScript, Perl, PHP, Python, and Ruby have been instrumented so that DTrace scripts can meaningfully examine programs in those languages. This would be particularly important for monitoring DALiuGE. There are problems with Go (which other tools also suffer from), due to the way Go handles stacks. This may be an issue for monitoring Docker.

It is in Solaris 10 and later derivatives including Illumos. It has been adopted in NetBSD and FreeBSD and macOS. It is supported as part of Oracle Linux. For other Linux systems, there is a dtrace4linux port on GitHub, which is still seeing some activity.

The DTrace documentation is pretty good and there is a fine book about it.

16.7. LTTng

Omitted from this document.

16.8. SystemTap

SystemTap was basically developed to be the Linux rival to DTrace, just as btrfs was developed to be the Linux rival to ZFS. It has had a number of security and integrity issues over the years.

It has tracepoints in the Java and CPython VMs.

I would have written more about it, but eBPF+bcc seems to be a superior alternative. In fact eBPF is one of the possible backends for SystemTap.

16.9. Extended Berkeley Packet Filter

The Berkeley Packet Filter was a kernel-level service that allowed user programs to select packets of interest using mini programs that were guaranteed to be safe. This has been substantially extended. The eBPF is built into the Linux kernel. Scripts are still guaranteed to be safe, and can be compiled to safe native code by a JIT.

Like DTrace, eBPF can do aggregation in the kernel, reducing the overhead of passing measurement data from kernel to user-land.

Brendan Gregg, a noted DTrace expert, thinks well of eBPF. The following points were written by him in 2016:

> The Linux kernel now has the following features built in (added between 2.6 and 4.9):
>
> - Dynamic tracing, kernel-level (BPF support for kprobes)
> - Dynamic tracing, user-level (BPF support for uprobes)
> - Static tracing, kernel-level (BPF support for tracepoints)
> - Timed sampling events (BPF with perf_event_open)
> - PMC events (BPF with perf_event_open)
> - Filtering (via BPF programs)
> - Debug output (bpf_trace_printk())
> - Per-event output (bpf_perf_event_output())
> - Basic variables (global & per-thread variables, via BPF maps)
> - Associative arrays (via BPF maps)
> - Frequency counting (via BPF maps)
> - Histograms (power-of-2, linear, and custom, via BPF maps)
> - Timestamps and time deltas (bpf_ktime_get_ns(), and BPF programs)
> - Stack traces, kernel (BPF stackmap)
> - Stack traces, user (BPF stackmap)
> - Overwrite ring buffers (perf_event_attr.write_backward)
>
> The front-end we are using is bcc, which provides both Python and lua interfaces. bcc adds:
>
> - Static tracing, user-level (USDT probes via uprobes)
> - Debug output (Python with BPF.trace_pipe() and BPF.trace_fields())
> - Per-event output (BPF_PERF_OUTPUT macro and BPF.open_perf_buffer())
> - Interval output (BPF.get_table() and table.clear())
> - Histogram printing (table.print_log2_hist())
> - C struct navigation, kernel-level (bcc rewriter maps to bpf_probe_read())
> - Symbol resolution, kernel-level (ksym(), ksymaddr())
> - Symbol resolution, user-level (usymaddr())
> - BPF tracepoint support (via TRACEPOINT_PROBE)
> - BPF stack trace support (incl. walk method for stack frames)
> - Various other helper macros and functions
> - Examples (under /examples)
> - Many tools (under /tools)
> - Tutorials (/docs/tutorial*.md)
> - Reference guide (/docs/reference_guide.md)

While he concedes that “developing new tools/metrics [in] bcc right now is much harder [than in] DTrace”, there is ongoing work to address that, and eBPF is officially part of Linux, while DTrace is not. He has a website with some useful tools.

17. Appendix B - Preliminary results from measuring system overheads in and out of containers

This Appendix should be read as the second part of Section 13 -with section 17.1 (below) replacing section 13.5

17.1. Preliminary results

So far the only communication scheme tested is ordinary UNIX pipes. The same framework will be used for the other “message queue” approaches. Three process placement schemes were tested:

0	let the scheduler decide which cores to run the processes;
1	pin both processes to the same core;
2	pin the processes to different core.

There is so far little difference between 0 and 2.

Four message sizes were measured: 128 bytes, 256 bytes, 512 bytes, and 1024 bytes.

The next thing to measure will be the effect of the O_DIRECT flag in the system call to create the pipes, which puts the pipes in “packet mode”. In that mode message boundaries are respected, which is just like sending and receiving messages using message queues.

Table 1: packet = size of message in bytes, core = where the processes were placed (scheduler chose, same core, different core), user and kernel are the number of seconds spent in user code or the kernel respectively, CPU is user+kernel, and real is wall time in seconds.

packet	core	user	kernel	CPU	real
128	sched	3.14	5.28	8.42	4.522
128	same	2.98	5.88	8.86	4.795
128	diff	3.32	5.82	9.14	4.961
256	sched	2.96	5.80	8.76	4.700
256	same	3.02	6.58	9.60	5.195
256	diff	2.90	6.42	9.32	5.021
512	sched	2.94	6.56	9.50	5.137
512	same	3.08	6.60	9.68	5.232
512	diff	2.92	6.40	9.32	5.042
1024	sched	3.42	7.20	10.62	5.732
1024	same	3.24	7.00	10.24	5.508
1024	diff	3.14	6.92	10.06	5.406

Perhaps the first thing to note in this table is that the real time is roughly half the CPU time, *even when both processes were pinned to the same core*. The surmise is that much of the kernel work was done on a separate core (there are four, after all), and there does not seem to be anything we can do about that. And under normal circumstances we would not want to.

The second thing to note is that the real time grows by roughly 0.2 seconds whenever the message size doubles. That tells us that the time is not dominated by copying at this scale.

At this point we are concerned with three problems:

- getting the framework right;
- measuring the time for one communication method (classic pipes, roughly 5 μsec);
- seeing jitter.

To keep the present draft short we present summaries of just two conditions: 128 bytes same core, and 1024 bytes different cores.

```
> bm <- read.table("bm11.dat", T)
> summary(bm)
     total                mtos                 stom
 Min.   :  12203   Min.   :  1508   Min.   :  2320
 1st Qu.:  17118   1st Qu.:  2249   1st Qu.: 12383
 Median :  17385   Median :  3917   Median : 13876
 Mean   :  18306   Mean   :  4713   Mean   : 13446
 3rd Qu.:  18025   3rd Qu.:  7697   3rd Qu.: 15119
 Max.   :17245757   Max.   :17179306   Max.   :293658
> c(total=co(bm$total),mtos=co(bm$mtos),stom=co(bm$stom))
   total         mtos         stom
0.7502602  0.7453125  0.3643146
```

These are “five-number summaries”: minimum, first quartile, median, mean, third quartile, and maximum. The times are in nanoseconds, and measure real time, not CPU cycles.

The maximum figures are hugely out of line compared with the majority. The round trip time for this case is normally about 17 μsec, but was at least once a thousand times longer.

The numbers at the end are (lag-1) autocorrelations, showing that there is a weak (stom) to strong (mtos, total) tendency for long delays to be followed by long delays and short delays by short delays.

```
> bm <- read.table("bm82.dat", T)
> summary(bm)
     total                mtos                stom
 Min.   :  13506   Min.   :  2072   Min.   :  2739
 1st Qu.:  19242   1st Qu.:  2754   1st Qu.: 13597
 Median :  19494   Median :  4554   Median : 15169
 Mean   :  20117   Mean   :  5337   Mean   : 14640
 3rd Qu.:  19778   3rd Qu.:  8131   3rd Qu.: 16509
 Max.   :5547206   Max.   :5491934   Max.   :296730
> c(total=co(bm$total),mtos=co(bm$mtos),stom=co(bm$stom))
   total         mtos         stom
0.7522149  0.7280766  0.3383428
```

Finally, *here* are two graphs. The first shows the probability density for the round trip delays, on a logarithmic scale. (Think of it as a smoothed histogram.) There is a “rug” under it so that we can see the rare but large cases. Note the small peak above the 4.5 tickmark.

The second shows the round trip time for the first 2,000 messages, also on a logarithmic scale.

*(**NB** - graphs are at pg 6 & 7 at the PDF document here)*

This very clearly shows jitter. The amount of memory involved here is quite modest. The programs were compiled in 32-bit mode, amounting to one page of code and two pages of data, plus dynamically linked libraries. So it doesn't seem to be a caching effect.

There is a hint that the high-delay spikes might be periodic, with a period around 4 or 5 msec, which is a plausible clock interrupt frequency.

17.2. Message-passing results

Table 2: PIPE = plain pipe, S5SM = System V Message Queue, PXMQ = POSIX Message Queue, USCK = UNIX-domain socket, LSCK = TCP socket connected to localhost.

User and System are CPU times, CPU is their sum, and real is actual physical time, all reported in seconds.

How	Where	User	System	CPU	real
PIPE	host	1.94	4.10	6.04	3.296
PIPE	container	1.59	5.31	6.90	3.759
S5MQ	host	2.64	4.24	6.88	3.815
S5MQ	container	2.74	5.87	8.61	4.775
PXMQ	host	1.34	2.76	4.10	2.422
PXMQ	container	1.54	3.62	5.16	2.928
USCK	host	2.34	9.88	12.22	8.538
USCK	container	2.40	10.96	13.36	9.165
LSCK	host	4.32	59.10	63.42	45.635
LSCK	container	4.95	53.42	58.37	40.079

Whenever we run a benchmark, we get a surprise. In particular, had not expected that System V and POSIX message queues would be so different. And while we expected that UNIX-domain sockets would be cheaper than TCP to 127.0.0.1, we had not expected the magnitude of the difference.

18. Appendix C - Security in the Kernel - Other features

This Appendix provides details of the features listed in Section 14.7

18.1. Address Space Layout Randomisation

This is provided by the kernel and the the loader (not glibc). The idea is what whenever you run a program,

- The stack begins at a randomly chosen address.
- Memory mapped files are loaded at randomly chosen addresses, including dynamically linked libraries.
- If the executable code was compiled for position independence, that will be placed at a randomly chosen address. This has a 5-10% performance penalty on x86, but 64-bit architectures are OK.
- A randomly chosen gap will be inserted between the end of the initial code and the region managed by brk(2)/sbrk(2).

The purpose of all this is to make many attack methods fail. Even if they succeed in injecting code into a running program, they do not know what addresses to use.

This is widely regarded as a useful protection. It is still worth benchmarking for the SDP.

There is one big issue with ASLR. It used to be that checkpoint/restart was a comparatively simple thing. Simply save a snapshot of the mutable memory of a program (checkpoint) and load it to restart, with a bit of dancing around to reopen files. Already in the 1980s this was getting harder. ASLR make it extremely complicated. Weirdly, the easiest way to do checkpoint/ restart now is to run an application in a container and take a snapshot of the whole container. Since we are expecting to use containers anyway, this is not a problem.

This technique has been applied to the kernel, but in that context it is largely security theatre.

18.2. Automatic Security Updates

There's an Ubuntu configuration feature unattended-upgrade which can be set to automatically apply security updates every day. Since Ubuntu 16.04 LTS, this has been set by default.

There are *management* machines which interact with the outside world and *computing* machines which do not. For reproducibility of results, we want to change the computing machines seldom and very very carefully, relying on the outer layer. The fundamental problem here is that we have to be very sure (a) that the server the upgrades are loaded from is indeed operated by Canonical and that (b) the risk that Canonical's upgrades may make things worse is lower than the risk of succumbing to an attack exploiting an unpatched security bug.

We are not going to take the expected cheap shot at Microsoft here. Fortune magazine reported in September 2017 that thousands of Macs where security updates had been installed had *not* had the EFI firmware upgraded. This is described in more detail in a blog post which links to a

technical paper. c|net reported in November 2017 on a “botched” security update that broke file sharing for many users.

18.3. File system Capabilities

The capabilities mentioned before can be associated with files and directories using xattr(1) or setcap(1). This means that instead of a program being set-user-id to root, it can be given a much more specific set of rights. For example, a program could have CAP_NET_ADMIN without being able to read and delete everyone's files.

This association is done using extended attributes. When a program is executed, the kernel checks how to revise the process' capability sets. That is done once per exec(2).

18.4. File system Encryption

Logical volumes can be encrypted.

Private directories can be encrypted.

Phoronix have a slightly dated performance comparison of unencrypted *vs* volume encrypted *vs* directory encrypted performance. As time goes on, more and more encryption support moves into silicon, so the overheads on new systems may be smaller, but it looks very much as though the performance cost may be excessive for the SDP.

Since we aren't *that* worried about information leaking out, the safety gain from encryption is rather indirect.

18.5. Heap protection

This refers to library-level support for detecting damage to the dynamically allocated storage area, such as buffer overruns and double frees.

The simplest way to use this feature is

MALLOC_CHECK=2 *myprog its arguments*

It would be worth experimenting with some of the radio astronomy programs to see whether the overhead of checking is of practical significance.

This is *not* a kernel issue but a glibc issue.

18.6. Kernel Livepatches

Canonical provide a "Livepatch" service (free for up to 3 machines, so not free for us) where kernel security issues are patched in a way that does not require a reboot.

That's pretty cool, but given that the service isn't free at the scale we're interested in, and that we will need to manage upgrades of things other than the kernel any way, it may not be worthwhile for the SDP.

18.7. Linux Security Modules and Mandatory Access Controls

Linux Security Modules isn't a kind of module but a *mechanism* that allows loadable kernel modules to express security *policies*.

“LSM inserts ‘hooks’ (upcalls to the module) at every point in the kernel where a user-level system call is about to result in access to an important internal kernel object such as inodes and task control blocks.” — Wikipedia. It doesn't catch every call from userland to the system. Hooks are applied after preliminary error checking, when it seems likely that a core data structure will be affected.

There are several issues mentioned in the Wikipedia page. We want to mention two.

- This relies on *every* potentially vulnerable system call having the appropriate hook(s) in it. If you add a new system call without placing LSM hooks, you have just introduced a possible security hole. There *was* a project to validate this automatically, using a program called CQUAL. The last change to CQUAL was made in 2004. It appears to have been superseded by the Oink project, which was last touched a year ago. Since Oink deals with C and C++, it may be a useful tool for the SDP software anyway.
- While some care was taken to keep overheads small, LSM does add an overhead to every checked system call, even when no security module is loaded.

What can it do for you? There are several security modules, including SELinux and AppArmor. AppArmor can mediate:

- file access (read, write, link, lock)
- library loading
- execution of applications
- coarse-grained network (protocol, type, domain)
- capabilities
- coarse owner checks (task must have the same euid/fsuid as the object being checked)
- mount
- DBus API (path, interface, method)
- signal
- ptrace(2)
- unix(7) named sockets
- unix(7) abstract and anonymous sockets starting with Ubuntu 14.10

18.8. No Open Ports

When Linux starts up, it should not be listening on any network ports (other than loopbacks). This is a configuration issue, not a kernel issue.

There is an exception to this for Zeroconf, the IETF automatic network setup approach. Ubuntu allows/uses Avahi Zeroconf (see the Ubuntu spec).

Any open port is a security risk. It is not clear whether Zeroconf is necessary for the SDP, but *some* open port seems to be needed for management.

There is no performance cost here. It does raise the issue of exactly what services the compute nodes should offer and start. Fewer is better.

18.9. Password Hashing

Originally UNIX stored hashed passwords in /etc/passwd. Since many applications had good reason to read other information stored in /etc/passwd, this opened up an attack route. For a long time UNIX systems, including Linux, have split the hashed passwords out into /etc/shadow. The hashing method is SHA-512 with a “salt”.

Passwords are seldom checked. There is no point in trying to circumvent this.

18.10. SECCOMP

A Linux process can operate in one of three states:

- Default: any system call is allowed (subject to permissions and capabilities).
- SECCOMP_SET_MODE_STRICT: the only allowed system calls are read(2), write(2), _exit(2), and sigreturn(2). Note that open(2) is not allowed.
- SECCOMP_SET_MODE_FILTER: when you call seccomp(2) with this flag, you pass a pointer to a Berkeley Packet Filter program, and that decides whether any particular system call is allowed.

This would be quite tempting, but

- STRICT mode looks useful for constraining programs that are only supposed to do computing, but in UNIX the connection between something like OpenCL or OpenGL and the GPU appears to require system calls.
- FILTER mode is amazingly powerful **but** it adds overhead to every system call to check whether it is allowed (more than STRICT mode).

Linux Security Modules are somewhat like the FILTER mode.

18.11. Stack Protector

-fstack-protector is a command-line option of gcc and clang. This uses a “stack canary” where a random value is pushed on the stack at function entry and checked for at function exit; if the value is different a buffer overflow has occurred and the program should not continue. The point is to prevent (some) buffer overflow attacks (and bugs!).

There are four levels:

- None is the default, offering the highest performance and risk.
- Plain puts a stack canary in every function declaring a local character array 8 bytes or longer.
- Strong puts a stack canary in every function declaring a local array of any type, or take the address of a local variable, or declare local registers.
- All puts a stack canary in every function.

This is normally used for ordinary programs, but the Linux kernel can be built using stack protection and has been. The LWN article linked above gives size overhead figures (which are modest) but not time overhead figures.

We suggest that Python be built with this feature enabled.

18.12. SYN cookies

A SYN flood attack is one kind of denial-of-service attack using a fairly fundamental aspect of the way IP sets up connections. Daniel Bernstein invented a rather cunning way of mitigating such attacks called SYN cookies. This approach has some problems, but it is only activated when a machine detects an attack.

This is something that outwards-facing machines in the SDP can clearly benefit from. Bernstein describes SYN cookies here https://cr.yp.to/syncookies.html

18.13. Uncomplicated Firewall

The packet filtering system in Ubuntu is called netfilter. There is a suite of programs for managing this called iptables(8). On top of that Ubuntu offers ufw(8) which is said to be simpler than iptables, but iptables can do everything this can. Basically, ufw or iptables is what you use to set up a firewall, which is basically a set of rulesets saying which packets to accept and which to discard.

Packet filtering is important. ufw(8) doesn't add anything to the overheads.

19. Appendix D - Papers

A selection of recently published papers that we are currently considering for future work -in contact with some of the authors, exploring its applicability

19.1. TabulaROSA

"TabulaROSA: Tabular Operating System Architecture for Massively Parallel Heterogeneous Compute Engines" (J. Kepner et al., July 2018). MIT - Sandia - Indiana University.

The idea of using a database in a operating system role, defines key OS functions in terms of rigorous mathematical semantics (associative array algebra) directly translatable into database operations and result in OS equations. These provide a mathematical specification for a "Tabular OS Architecture" (TabulaROSA) that can be implemented in any platform. TabulaROSA simulations show 20x higher performance as compared to Linux while managing 2000x more processes in fully searchable tables.

19.2. Specialization of the HPC Software Stack

"Toward Full Specialization of the HPC Software Stack: Reconciling Application Containers and Lightweight Multi-kernels" (B. Gerofi et al., June 2017). RIKEN, Japan - INTEL, US.

This paper proposes a framework for combining application containers with multi-kernel OS enabling specialization across the software stack. Shows that containers impose zero runtime overhead even at scale, and that users benefit from lightweight multi-kernels attaining identical speed-ups to the native multi-kernel execution.

19.3. Monolithic OS Design is Flawed

"The Jury Is In: Monolithic OS Design is Flawed. Microkernel-based Designs Improve Security" (G. Heiser et al., August 2018). UNSW Sydney, Australia.

Almost all exploits are at least mitigated to less than critical severity, and 40% completely eliminated by an OS design based on a verified microkernel, such as seL4

19.4. Mitigating OS - L1 Terminal Fault

"Intel Analysis of L1 Terminal Fault (L1TF)" (Intel, Disclosure date 14 August 2018)
https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault

19.5. Measuring the Impact of Spectre and Meltdown

"Measuring the Impact of Spectre and Meltdown" (A. Prout et al., July 2018). MIT Lincoln Lab.

The Spectre and Meltdown flaws represent a new class of attacks that have been difficult to mitigate. The mitigations that have been proposed have known performance impacts. This paper measures the performance impact on several workloads relevant to HPC systems and also shows that the performance penalties are difficult to avoid even in dedicated systems where security is a lesser concern.

19.6. Cost optimisation in eScience and radio astronomy

"On optimising cost and value in eScience: case studies in radio astronomy" (C. Brokema et al., June 2018). ASTRON, NL - University of Cambridge, UK - Vrije Universiteit, NL

The paper introduces "a conceptual model to approximate the lifetime relative science merit" of a low-cost computing system. Of particular interest is the analysis of "the impact of Meltdown & Spectre on value" and the cost savings in the deployment of OS and other support services in the SKA's SDP.