Australian Government
Department of Defence
Defence Science and
Technology Organisation

The Design and Implementation of Persistence in the Annex System

Anton V. Uzunov and Duncan A. Grove

Command, Control, Communications and Intelligence Division
Defence Science and Technology Organisation

DSTO-TN-0907

ABSTRACT

Orthogonal persistence in operating systems has been a topic of research for a number of years. Several commercial, as well as research projects, have implemented orthogonal persistence as an essential part of their design. Amongst these we can count the Annex software system developed at DSTO. In this technical note we consider the design and implementation of persistence in Annex against the background of two other capability-based, orthogonally persistent operating systems. This background is used to highlight the approach taken by the Annex system, and the reasons why such an approach was used. The description given discusses the current Annex prototype and explores the directions persistence in Annex will take in the future as a result of lessons learned.

Published by

Command, Control, Communications and Intelligence Division
DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh South Australia 5111 Australia

Telephone: (08) 8259 5555
Fax: (08) 8259 6567

AR-014-608
August 2009

APPROVED FOR PUBLIC RELEASE

The Design and Implementation of Persistence in the Annex System

Executive Summary

"*Persistence* is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created)." [Booch 1994] A software system is said to implement persistence if all its objects are persistent in the above sense. When all objects in such a system remain unaware of being persistent, the system is said to implement *orthogonal persistence*.

Orthogonal persistence in operating systems has been a topic of research for a number of years. Several commercial, as well as research projects, have implemented orthogonal persistence as an essential part of their design. Among these is the Annex software system developed at DSTO. In this technical note we consider the design and implementation of persistence in Annex against the background of two other capability-based, orthogonally persistent operating systems: KeyKOS and Grasshopper. This background is used to highlight the approach taken by the Annex system, and the reasons why such an approach was used. The description given discusses the current Annex prototype and explores the directions persistence in Annex will take in the future as a result of lessons learned.

Authors

Anton V. Uzunov
Command,Control,Communications Intelligence Div

Anton Uzunov received his B. Math. & Computer Science from the University of Adelaide in 2004. In mid-2006 he was awarded an Honours degree in Pure Mathematics, and later that year began to work for the Advanced Computer Capabilities group in DSTO. His research interests include distributed and object-oriented software systems, parallel computing and languages, operating system architectures and various aspects of computer security.

Duncan A. Grove
Command,Control,Communications Intelligence Div

Dr Duncan Grove graduated with first class honours in Computer Systems Engineering from the University of Adelaide in 1997, and received a PhD in Computer Science from the same institution in 2003. Following his doctoral studies he joined DSTO's Advanced Computer Capabilities where he is Science Team Leader for the Annex long range research task, which is developing Multi Level Security devices and next generation networking technologies. His research interests span all aspects of computer and network security, including secure operating system design and implementation, secure programming languages, retrofitting COTS software into secure environments and securing 802.11 and Bluetooth wireless networks, as well as next generation IPv6 networks including Mobile IPv6, embedded computers, and parallel computing.

Contents

1. Introduction

Persistence in operating systems has been an active area of research in computer science since around 1970. Many systems have been created which make use of persistence in one way or another: Clouds [Dasgupta et al. 1988], MONADS [Rosenberg 1990], Grasshopper [Dearle, di Bona et al. 1993], the Walnut kernel [Castro 1996], the L3 microkernel [Liedtke 1993] and its successor L4 [Skoglund, Ceelen & Liedtke 2000], KeyKOS [Hardy 1985] and Charm [Dearle & Hulse 2000]. There is a great difference in design between these systems, and an equally great difference in their approach to persistence and its implementation. Some researchers have considered what a system implementing persistence requires [Kemikli & Erdogen 1998], or whether traditional operating system paradigms are capable of supporting the modern concepts of persistence [Dearle, Rosenberg et al. 1992] and [Dearle & Hulse 2000]. Others have simply sought to introduce persistence into their particular operating system, without regard to how it should be designed a priori. As a result of these differing approaches, some of these systems implement persistence as a fundamental part of their design, while others implement it as a layer or as an additional feature to be added on top of already existing functionality. Among these many systems which implement persistence in one form or another, we can count the Annex software system developed at DSTO. Annex is an object-capability based, distributed software system constructed to provide a secure computing environment [Grove et al. 2007], [Grove et al. in-prep]. Annex consists of a security kernel and a set of object collections which are built to run on top of an underlying host OS. The implementation of checkpointing in Annex uses an external checkpointing module (ECM), i.e. a module that is external to the system constructs of Annex, residing at the layer of the OS underlying Annex.

In this paper we will give an overview of the design and implementation of the persistence mechanism for the current Linux-based prototype of the Annex system, in light of two other capability-based operating systems: KeyKOS and Grasshopper. These two operating systems were chosen for comparison because of their unique designs and the sharp distinction these designs have between each other. The approaches to persistence of these systems are described here in enough detail to convey their general ideas and design goals, and to offer a contrast with Annex. This contrast, and the background it affords, helps to highlight the approach taken by Annex, and the reasons why such an approach was used. Finally, we explore the directions persistence in Annex will take in the future as a result of lessons learned, and consider how the persistence mechanism will alter under the new circumstances.

1.1 Overview

The outline of the paper is as follows. In Section 1.2 we distinguish between two types of systems which implement (orthogonal) persistence in some way. In Section 2 we consider the top-level design and persistence implementations of two capability-based operating systems: KeyKOS and Grasshopper. As derivatives of KeyKOS, and using a very similar approach to persistence, EROS and CaprOS are mentioned in passing. In Section 3 we consider the Annex system. As with the other systems, the top-level design is outlined first, after which the design and implementation of the persistence mechanism is discussed in greater depth. This section also explores how certain future directions in the Annex system will modify the design and implementation of its persistence mechanism. Section 4 offers a comparison and discussion of

the various approaches taken to implement persistence in the three systems, with an emphasis on Annex. A link is made with Section 1.2, giving another viewpoint on the types of systems which provide implementations of persistence. In Section 5 we conclude.

1.2 Orthogonal persistence

"Persistence is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created)." [Booch 1994] A system is said to implement persistence if all its objects are persistent in the above sense. Orthogonal persistence is persistence which has been implemented so that all objects in a system remain unaware of being persistent.

Broadly speaking, we can distinguish persistent systems as being either *strongly* or *weakly persistent*. A strongly persistent system is one in which all data – whether in volatile or non-volatile storage - is treated as being persistent. From an abstract point of view, objects in a strongly persistent system no longer exist in memory, or disk etc., but in a logical persistent store [Cooper & Wise 2006], [Kemikli & Erdogen 1998], which abstracts over storage and, with respect to the user and developer, is viewed as storage independent. In such a persistent system a programmer creates applications that reference addresses in the persistent store, which naturally implies that a consistent addressing scheme must be used by the persistent system to translate addresses from the persistent store to addresses relating to the underlying storage [Dearle & Hulse 2000]. Such a system must also be able to manage the persistent object store by transparently moving data from non-volatile to volatile storage throughout the lifetime of the system. An example of such a strongly persistent system is Grasshopper [Dearle, di Bona et al. 1993].

In contrast, a weakly persistent system is one in which all objects are persistent (i.e. their existence transcends time and/or space), but which retains the characteristics of the original system, i.e. the system may still distinguish between volatile and non-volatile memory storage, perform filesystem I/O as well as standard memory addressing. This latter system need not use a persistent store, since objects still exist in memory or on disk and need not be managed in the same way as in a strongly persistent system. The objects that exist in non-persistent storage, however, are periodically or otherwise transparently transferred to persistent storage so that they transcend system shutdown/restart cycles. In this sense, a strongly persistent system is a special case of a weakly persistent system. Examples of systems implementing persistence are Annex and the L4 micro-kernel [Skoglund, Ceelen & Liedtke 2000].

In either case, to implement some form of persistence a system must have a mechanism for saving and restoring data in memory to persistent storage, e.g. disk, in a coherent way that will ensure data and system consistency. This mechanism is usually referred to as checkpointing. A system may be checkpointed at some time and restarted again later from that checkpoint. Of course, merely saving the data to persistent storage is insufficient to guarantee data consistency when the system is restarted. In addition to checkpointing, the persistence mechanism must include all necessary system support to guarantee data consistency. In the case of strongly persistent systems (using a persistent store) a checkpoint

mechanism simply has the role of saving non-persistent data to the persistent store. In the more general case of a weakly persistent system, checkpointing has the role of taking a "snapshot" of the volatile system state for restoration at some later point or in case of system failure.

2. Persistence in KeyKOS and Grasshopper

2.1 KeyKOS

2.1.1 Background

KeyKOS is an object-capability based system "originally designed to solve the security, sharing, pricing, reliability and extensibility requirements of a commercial computer service in a network environment" [Hardy 1985]. KeyKOS refers to capabilities as keys, which are the determinants of authority and access control in the system. Keys are held by objects to refer to other objects within the system.

Objects are implemented in KeyKOS via what are called domains. Domains are a process execution abstraction which contains a key-referenced address space (called the address segment) and program instructions within this address space (called the domain code). Objects may have one of the following states: running, available and waiting.

An object holding a key is able to use (invoke) the key, thereby affecting state changes on the object to which that key refers. The actual invocation of keys between objects is implemented via a message passing mechanism. The latter is realised by using a gate key, which provides the authority to send a message to a designated domain, after which that domain handles all pending messages from the calling domain. The message itself is interpreted as parameters to a subroutine, while the invocation of the (gate) key is interpreted as a subroutine call. A KeyKOS object can thus be considered as a domain, together with a set of specific keys for various functions within the system, while key invocation can be considered simply as a method call from one object to another.

2.1.2 Top-level persistence design

KeyKOS implements orthogonal persistence in that it is able to checkpoint the complete state of all applications without any knowledge or cooperation from the application programs themselves [Landau 1992]. KeyKOS persistence is handled exclusively by the kernel, which implements a persistence mechanism by taking regular system-wide checkpoints of the system state. In KeyKOS, system state is determined by its object-based architecture: objects (and hence domains) are implemented by two rudimentary system entities – pages and nodes – which in turn are implemented on top of the virtual memory system. KeyKOS saves the system-wide state by saving the state of all pages and nodes, thereby saving the state of all objects. This is done by designating certain areas on the disk – a working, checkpoint and home area - and conceptually cycling the pages and nodes between these areas as they are

swapped in and out of memory. The checkpoint mechanism is thus closely integrated with the virtual memory system.

2.1.3 Checkpoint mechanism in KeyKOS

The checkpointing mechanism in KeyKOS can be summarised as follows (following [Landau 1992]):

Step 1: (stabilisation)
All active entities (domains) are halted.

Step 2: (serialisation of state)
The corresponding pages and nodes of the system entities, including any used by the kernel, are written to the working area on disk via the paging mechanism of the virtual memory system. During this process the state of the active entities, such as the registers used, running state etc., is also saved. The role of the working area is then interchanged for the role of the checkpoint area, and vice versa. Any persistence-incompatible state in the system – such as network connections etc. - is not saved and must be restarted by other means after the system is restarted.

Step 3: (migration of state to home area on disk)
The relevant state – pages and nodes – which is now written on disk in the checkpoint area is moved (migrated) to the home area. Only state that has been modified since the last checkpoint is migrated. Migration is done concurrently with normal system operation.

As explained previously, the checkpointing mechanism is implemented entirely within the kernel; only a part of the migration phase utilises user-space code.

The restart mechanism uses the same approach as the checkpoint mechanism.

Step 1: (restoration)
When the system is restarted, the disk area currently designated as the checkpoint area is used as a swap area by the virtual memory system, from which all the relevant pages and nodes are loaded back into memory. The system is thus restarted from exactly the same memory state that it had before.

Step 2: (stabilisation)
Any relevant data structures are re-built after the saved state has been restored, for example, the page table inside the KeyKOS kernel.

2.1.4 Checkpoint reliability

KeyKOS enforces fault tolerance for checkpoints by data redundancy: all checkpoints are mirrored on a second disk. The reliability of the last checkpoint is thus ensured insofar as it completed succesfully. Granted that the checkpoint mechanism itself generates consistent checkpoints, consistency is ensured by always allowing the migration phase of each checkpoint to complete before the next checkpoint is taken. Moreover, the kernel runs a

consistency check of all relevant data structures before checkpointing, in order for there to be greater certainty that the saved system state will be consistent. Failure recovery of checkpoints can be implemented by using older versions of pages from one of the designated areas on disk.

2.1.5 EROS and CaprOS

EROS is a capability-based operating system derived from KeyKOS, designed for high reliability and security [Shapiro 1999], [Shapiro & Hardy 2002]. EROS supports what the authors term global orthogonal persistence, i.e. orthogonal persistence that saves the entire system state to disk. As in KeyKOS, this process is intimately linked with the virtual memory system. The checkpointing mechanism consists of periodically creating a snapshot of the whole system state, and writing it using copy-on-write techniques to a disk area called the checkpoint log. When the entire snapshot has been written, data in the checkpoint log is migrated to another area on disk called the home area. The system is then notified that the checkpoint process has finished and that a new checkpoint can be taken. If the system fails, it resumes from the most recently stabilised snapshot, which is deemed consistent by virtue of its construction [Shapiro 1999]. Since EROS is derived from KeyKOS, it is unsurprising that the persistence mechanism in EROS is similar to the one used in KeyKOS [Shapiro, Farber, Smith 1996]. The most complete account of the EROS persistence mechanism is [Shapiro & Adams 2002].

EROS is the predecessor of CaprOS, another capability-based OS with similar aims. The design of the checkpoint implementation in CaprOS is based on both that of EROS and KeyKOS, with some minor differences; details can be found in [Shapiro 2008].

2.2 Grasshopper

2.2.1 Background

The Grasshopper operating system claims to provide explicit support for orthogonal persistence by virtue of its design [Dearle, di Bona et al. 1993]. Grasshopper uses several abstractions to implement both security and persistence: containers, loci and capabilities. Containers are data abstractions that can hold the code and data of an executing process or task; they have both a user-space and a kernel-space component. Loci are abstractions over execution, and can be viewed as being the context of an executing process or task within a particular container. Loci can make invocations that call into other containers, such that the loci become tied to the invoked container. Multiple loci can execute in a single container at any one time, thereby implementing a form of concurrency. Memory or, in this case, data sharing between containers is achieved via mappings from one container to another. Capabilities are the means used by Grasshopper to reference system entities (e.g. containers) and to provide a security policy.

2.2.2 Top-level design of the persistence mechanism

The Grasshopper system can be thought of as a kernel together with a set of containers and a set of loci operating in these containers. Persistence is implemented for all three of

Grasshopper's abstractions, i.e. containers, loci and capabilities, however, given that containers are Grashopper's only data abstraction, the majority of the effort expended by the persistence mechanism concerns containers and their related data.

Containers are almost entirely managed in user-space by container managers. Container managers allow for the following operations to be defined: physical and virtual memory allocation, stable storage allocation and stabilisation of data. When data is to be checkpointed, stabilisation is not achieved by globally suspending the system (as in KeyKOS for example), but by stabilising each container independently. The system-wide consistent state (called a consistent cut) is achieved in Grasshopper by an algorithm that considers all the previous stable states of the individual containers (after the last system-wide stable state) and determines which individual-stable states are suitable to create the global stable state.

In the Grashopper kernel, persistence is implemented only for the kernel-space components of the container, loci and capability functionalities. The kernel state to be saved thus consists of the kernel-related container and loci data, including invocation call-chain for loci, their host containers, any container mappings and the relevant capabilities. Grasshopper implements a separate kernel-level persistent store for kernel data, which is stabilised (as for containers at the user-level) independently. This store uses persistent arenas, i.e. “large-size regions that may have any number of versions created independently of each other” [Lindstrom, Dearle et al. 1995].

Grasshopper attempts to address consistency issues by using a form of consistency tracking between loci [Dearle & Hulse 2000]. The method involves keeping track of page-faults and locus dependency information so that the kernel can determine a consistent global state whenever this is required.

2.2.3 Checkpointing mechanism in Grasshopper

The persistence mechanism in Grasshopper is a three stage process:

Stage 1: (causal data dependency determination)
The Grasshopper kernel determines certain causal dependencies between loci and their associated containers in order to create a consistent and recoverable representation of a subsection of the system.

Stage 2: (stabilisation)
Container managers are asked to stabilise their data in order to create a global stable state. When the user-space data is stabilised, the kernel stabilises its own data.

Stage 3: (serialisation)
The stabilised state is written to disk in the form of metadata from which the previous system state can be recovered. User-space data and kernel-space data are kept distinct throughout.

2.2.4 Discussion for Grasshopper

Although Grasshopper uses system abstractions that are uncommon to most modern operating systems, there is still a distinction between kernel-space and user-space entities. Consequently, their states have to be saved and managed separately. With respect to its handling and management of persistent data, Grasshopper takes this approach one step further and uses different mechanisms for kernel entities (kernel-space container and loci data) and user-space entities (containers via container managers).

3. The Annex System

3.1 Background

Annex is an object-capability based distributed software system that aims to provide a secure computing environment [Grove et al. 2007], [Grove et al. in-prep]. The Annex system consists of a security kernel and a set of object collections, all built to run on top of an underlying host OS (see Figure 1). Annex objects are memory separated entities that run as processes on the host OS. They provide a security separation mechanism which, when paired with capabilities for naming and access control functionality, allow the implementation of various security policies. Objects may reside on remote computer systems; inter-object communication is implemented via an RPC-based, kernel-controlled message passing mechanism. In the current implementation, this is realised via four main types of messages: object-to-object, object-to-kernel, kernel-to-object and control messages. Control messages are used as an interrupt mechanism within the message passing framework, whereas the three other message types embody the semantics of object method calls. Objects are allowed to make method calls on other objects provided that they possess a capability and the required set of method permissions on that capability (see [Grove et al. in-prep]). The security policy of the system is thus determined by the semantics of the interconnected object-capability graph.

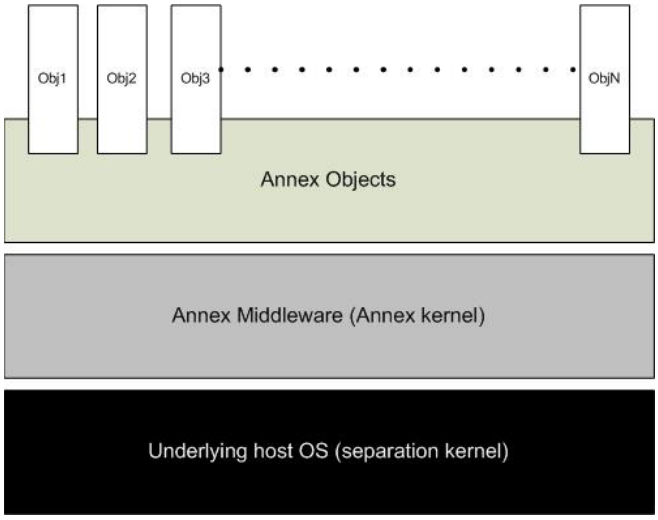

Figure 1: Conceptual design of the Annex system

As in a standard object-based system, Annex objects have a unique identity, state and behaviour. Behaviour is defined by the stub-exported methods of the object. The Annex kernel keeps account of each object's scheduling, so that with respect to the kernel, objects can be classed as running – when they are running an Annex task – or idle. At system start-up all objects are required to register with the Annex kernel and are marked as being in a special registering state. After registration, objects are set to the idle state. Registration is done for the purpose of establishing the IPC links used by the underlying host OS.

With respect to the entire system, Annex is built around a micro-kernel design paradigm, with the core functionality residing in the kernel and all other non-essential functionality being delegated to Annex objects. Annex can thus be considered as a 3 layer system: the underlying host OS resides at the lowest layer; above this is the Annex kernel (or middleware); and above the kernel layer are the Annex objects, logically grouped into collections and system components.

3.2 Top-level design of persistence

Annex implements temporal persistence: i.e. all the objects can be checkpointed and restored at a later point in time. Spatial persistence is not implemented, since object-capabilities are tied to particular machine IDs, and cannot be moved or migrated to different computer systems using the Annex system. Although Annex is inherently a distributed system, at the time of writing there is only support for local, and not distributed, persistence.

The implementation of checkpointing in Annex uses an external checkpointing module (ECM), i.e. a module that is external to both the kernel and objects of the Annex system, residing at the layer of the OS underlying Annex. The task of this module is to serialise all non-persistent (but persistence-compatible) data used by Annex to persistent storage. Whenever it exists, non-persistent and persistence-incompatible data, such as open sockets for IPC, must be handled upon restoration and is not checkpointed (see later). The current prototype implementation of Annex on Linux uses the Berkeley Lab Checkpoint/Restart (BLCR) library [Duell, Hargrove, Roman 2002] as the external checkpoint module. BLCR works by serialising a process' memory space, open file descriptors and other runtime information to a file on disk or some other non-volatile storage. It should be noted that since the checkpointing module works with the memory spaces of the individual objects, as well as the memory space of the Annex kernel, it is assumed to be a trusted system component which is part of the OS underlying Annex.

In Annex objects are made up of an object layer and an object-wrapper layer providing a standard object interface to the Annex kernel. In the current prototype, persistence is handled by objects in a thin persistence layer on top of the object-wrapper layer that communicates with the checkpoint module to manage its shutdown/restart routines. The kernel also has such a persistence layer, which handles the checkpointing requests and, like the individual objects, manages the associated shutdown/restart procedures via the external checkpointing module. The delegation of all persistence-related functionality to its own dedicated layer implies that object-code can be completely unrelated to and unaware of checkpointing. Persistence can therefore be classified as being orthogonal to the system.

3.3 Implementation details

As noted previously, to implement the external checkpoint module Annex uses the BLCR library for Linux, which was originally developed for the Intel family of processors (both 32 and 64-bit versions). In order to integrate BLCR into the Annex system we undertook the task of porting BLCR to the ARM architecture. Initially this was done for the old ARM ABI, but with the advent of the newer ARM EABI we extended our ARM BLCR version to accommodate the changes. Since we also required that Annex objects have a small memory footprint, we decided to replace the standard C library on Linux with dietlibc [von Leitner], which in turn prompted a dietlibc port of BLCR.

With respect to implementation, BLCR consists of a Linux kernel-space module and a user-space library which is linked to all Annex binaries, including the Annex kernel. The Annex kernel and objects can request checkpoints via the BLCR-provided library calls, which in turn communicate with the Linux kernel-module to perform the actual checkpointing functionality. The Annex system can be restarted via Linux shell scripts using the BLCR command-line interface utilities.

3.4 Checkpointing mechanism

From the perspective of the Annex kernel, a system checkpoint is triggered either via a control message sent to the kernel externally (by an object), or via a signal from the underlying (host) operating system. Such signals occur, for example, during a machine shutdown/restart process and are sent in order to preserve the system state across power-cycles. The user can also enforce a checkpoint at any point in the system's operation. In either case, the Annex kernel initiates the checkpoint by a two phase process.

Phase 1: Stabilisation:

The whole system is required to be in a quiescent state. This is achieved by running the kernel in what is called latent mode, in which pending object-to-kernel messages are collected on the kernel's queue, while kernel-to-object messages are barred from being dispatched. A quiescent state thus ensures that no new method calls are being generated and that all objects are in an idle state prior to checkpointing.

Phase 2: Serialisation:

Once the system is quiescent, the kernel sends a control message to all Annex objects instructing them to checkpoint themselves. At this point all non-persistent and persistence-incompatible data must either be destroyed gracefully, or handled on a per-object basis: objects are allowed to individually serialise some of their own data, provided that they have a mechanism for restoring it that does not interfere with the system as a whole.

As the final step in the process, the kernel checkpoints itself using the same mechanism as the objects.

Both the objects and the kernel checkpoint themselves as processes of the underlying operating system by using the ECM, which, as noted earlier, serialises their memory spaces and related structures used by the OS underlying Annex.

The restart process is complementary:

Phase 1: Restoration:
The kernel is restarted, which restores the Annex kernel process to its previously running state. The kernel then calls on the checkpointing module to restore all Annex objects, one by one, which were running when the checkpoint was last taken. Any non-persistent and persistence-incompatible data is recreated on a per-object basis (i.e. each object restores its own non-checkpointed data), as on system start-up, so that when the Annex objects re-register with the kernel the system will be in a consistent state. IPC for objects is handled gracefully by the persistence layer, which calls into the object-wrapper layer and re-initialises the IPC data structures independently of any object-specific code.

Phase 2: Stabilisation:
Each object is waited on until it registers again with the kernel. Since the same is done upon checkpointing, we can be sure that all objects have valid checkpoints individually, such that if the restoration process by the external checkpoint module is error-free, then all objects will be restored consistently. Once all objects are initialised, the kernel resumes normal operation, delivering any queued messages not dispatched before the checkpoint was triggered.

3.5 Checkpoint reliability

Under the assumption that the ECM always creates consistent checkpoints, the only elements of unreliability in the Annex checkpoint mechanism are when an individual object fails to checkpoint and when the kernel itself fails to checkpoint. The first of these cases would leave the whole system open to restart failures and in an inconsistent state. Annex attempts to solve this problem by ensuring that all objects report back to the kernel to indicate that they have checkpointed successfully. If an object has failed to checkpoint, the kernel can give an indication of this. Although this is theoretically a recoverable error, in the current implementation this case is not handled, and the checkpoint must be discarded. If the kernel itself fails to checkpoint in this scheme, then the error is always unrecoverable, and in such a case an implementation of a roll-back mechanism is required to restart the system from previously created (consistent) checkpoints.

Although a small (finite) subset of the series of checkpoints created can be kept, i.e. the last N checkpoints, there is currently no management or version control of consistent checkpoints. The last checkpoint taken is the first to restart a system upon power-cycling or system failure; if a previous checkpoint is required to restart the system this must be done manually.

3.6 Future directions

Some future directions for persistence in Annex include:

1. Partial support for distributed persistence. This implies at least a greater degree of synchronisation between remote systems, as well as additional mechanisms for restoring persistence-incompatible data (e.g. network connections and associated state).
2. Support for dynamic object upgrades, whereby an object can be upgraded either without disruption to the whole system, or with additional mechanisms in place that restore normal system operation in the presence of the new object.

3.7 Discussion for the current prototype of Annex

In contrast to the preceding operating systems, Annex is (currently) implemented to run on top of another operating system, and therefore is limited by its use of an external checkpoint module. The separation between kernel and objects necessitates that the checkpointing process is twofold, treating objects and kernel as logically separate entities. The process used to checkpoint them, however, is identical since (from a design and implementation point of view) their memory space is simply serialised to non-volatile storage. Periodic checkpoints are not enforced as in the other operating systems considered, but capability for such a feature is fully compatible with Annex. Since the system reaches a quiescent state during stabilisation, there is no need to enforce copy-on-write mechanisms for the memory spaces of the various objects (as in KeyKOS [Landau 1992], CapROS [Shapiro 2008] or L4 [Skoglund, Ceelen & Liedtke 2000]). On the other hand, the necessity for a quiescent state and the delay this brings about in the system makes it inappropriate for real-time applications. This holds true for other systems whose persistence mechanism requires halting system activity during the stabilisation phase (e.g. KeyKOS). For the Annex system, however, this is not an issue for consideration, as Annex was never intended for real-time tasks.

The fact that capabilities are stored in the kernel shows a single point of failure for the security in the system – if the kernel state is not checkpointed consistently, then the security of the system can be compromised, e.g. by incorrect assignment of capabilities on system restart. The latter is true for all other capability-based systems which store the capabilities (whether as a single data structure or as multiple instances) in a single location or subsystem, e.g. systems which use the segregated capability architecture. Since all the systems considered here (i.e. KeyKOS, Grasshopper and Annex) use this architecture, checkpoint consistency is especially important, which is why each system attempts to provide some mechanism for its enforcement.

3.8 Lessons learned and the future of the Annex persistence mechanism

Sections 3.1 – 3.6 described the design and implementation of the persistence mechanism for the current Annex prototype. Since Annex is a research system, it is liable to change and what is described here can only ever be a snapshot of the current state of development of the system as a whole. Nevertheless, the experience gathered from implementing the current persistence mechanism has given valuable lessons about how it should be implemented once the system

evolves to the next stage. Below we discuss some of these changes, and what bearing they should have on the persistence mechanism.

The first difference is that the underlying host OS will be a separation kernel. In this model the Annex kernel will run as a server, with Annex objects layered on top of the micro-kernel provided abstractions. The four types of user-visible messages described in Section 3.1 will disappear, to be replaced solely by kernel controlled object to object RPC semantics. The move away from Linux also implies a different IPC mechanism, which, unlike sockets under the current implementation, will no longer be persistence-incompatible. The distinction between files and memory will also be removed; objects and memory should be the only abstractions dealing with storage. As well as the new IPC mechanism, all data in Annex should be persistence compatible, which implies that the requirement to handle persistence-incompatible data disappears.

Despite these changes, the overall system architecture will remain conceptually identical: Annex could still be considered as a 3 layer system: with the underlying host OS residing at the lowest layer, above which is the Annex kernel, and above the Annex kernel layer the Annex objects. While the various changes will affect the implementation of the persistence mechanism itself, the fundamental design as discussed in section 3.1 – 3.3 should remain unchanged – i.e. an external checkpoint module residing at the level of the underlying host OS will be utilised to perform the actual checkpoint functionality itself.

Perhaps the greatest change to the design of the Annex persistence mechanism will be that objects will no longer have a user-visible wrapper layer. The impact of this change for persistence is that objects will no longer be responsible for calling the external checkpoint module directly via their dedicated persistence layer, but will instead have to rely on the Annex kernel to perform this function on their behalf. The control of checkpointing should thus be shifted from user-space objects to Annex kernel-space exclusively, thereby bringing the overall design rationale closer to the kernel controlled mechanisms of KeyKOS and Grasshopper.

4. General Discussion

In Section 1.2 we distinguished between strongly and weakly persistent systems, by saying that the former can be thought of as a specialisation of the latter. This is certainly a valid viewpoint, however, another way to view this distinction is to consider a spectrum - between generic systems which implement persistence, on the one end, and strongly persistent systems, on the other – which measures the degree of persistence ubiquity in a system. In the latter sense, and of the systems we have considered thus far, Annex lies at one extreme of this spectrum. It is a weakly persistent system, which implements persistence as a layer within its whole design; persistence does not pervade the entire system, which functions as a normal OS built on top of another. At the other end of the spectrum is Grasshopper, which is a strongly persistent system. Every object exists in a persistent store, which is managed by the Grasshopper kernel and container managers. Grasshopper is a system in which persistence pervades the whole design. The other systems considered earlier – KeyKOS, EROS and

CaprOS – lie in the middle of the spectrum. Persistence is certainly an essential part of their design, however, these systems allow for the nominal distinction between files (logically a form of non-transient data) and memory (logically a form of transient data). Underneath this view of the system, everything is implemented by pages and nodes which the kernel manages via the virtual memory system and the checkpointing mechanism. The contents of files are thus also stored in memory, and checkpointed to disk periodically.

In a system like Grasshopper, which is designed to be strongly persistent, the persistence mechanisms can be very fine-grained and lead to a very flexible but involved implementation. KeyKOS uses the virtual memory system to implement its pages and nodes, which are the fundamental system entities making up the system state. A checkpoint in KeyKOS is thus a snapshot of the state of a collection of pages and nodes. It would be either inefficient to simply checkpoint the state of higher-level entities, given that they are built exclusively from lower-level pages and nodes, or doing so would reduce again to checkpointing the pages and nodes themselves.

In contrast to all the above, Annex is constrained to run on top of an underlying operating system. The approach taken to persistence thus differs from the other systems considered. The actual checkpointing functionality is implemented in the Annex external checkpoint module, which resides at the layer of the underlying host OS. This approach allows for the characteristics of the system to be retained, namely, that objects in Annex are used and referenced from within applications and the system itself using the semantics of the underlying OS. Checkpoints simply move non-persistent data stored in the underlying OS to persistent storage, thus implementing orthogonal persistence without requiring a persistent store.

5. Conclusion

In this paper we explored the persistence mechanisms and their relation to system design in three capability based, orthogonally persistent operating systems, with an emphasis on Annex. Systems were distinguished into strongly and weakly persistent systems. Strongly persistent systems, such as Grasshopper, were seen to have persistence ubiquity, which meant that the ordinary distinction between volatile and non-volatile memory was removed. This required the use of a logical persistent store. Weakly persistent systems, on the other hand, were seen to implement persistence without the need for a persistent store, allowing for salient system characteristics (including the distinction between volatile and non-volatile memory) to be retained. The various approaches taken to implement persistence were used to highlight the approach taken by Annex, which resides on top of another operating system, and the reasons why such an approach was used. The description given discussed the current Annex prototype and explored the directions persistence in Annex will take in the future as a result of lessons learned.

6. Acknowledgements

We would like to thank Dr. Paul Hargrove from Berkeley Labs for his prompt and helpful responses to our questions concerning the open-source BLCR package, as well as for his technical support during the porting of BLCR to the ARM architecture.

References

[Booch 1994] – Booch, G. (1994), "Object-Oriented Analysis and Design", 2nd Ed., Benjamin Cummings, CA, USA

[Castro 1996] – Castro, M., "The Walnut Kernel: A Password-Capability Based Operating System", PhD Thesis, 1996

[Cooper & Wise 2006] – Cooper, T. & Wise, M., "Critique of Orthogonal Persistence", Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96), 1996, pp. 122 - 126

[Dasgupta et al. 1988] – Dasgupta, P., LeBlanc, R., Mustaque, A. & Umakishore, R., "The Clouds Distributed Operating System", Technical Report 88/25, Georgia Institute of Technology, 1988

[Dearle, di Bona et al. 1993] –Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindström, A., Rosenberg, J. & Vaughan, F., "Grasshopper: an orthogonally persistent operating system", Computing Systems, v.7 n.3, Summer 1994, pp.289-312

[Dearle, di Bona et al. 1994] – Dearle, A., di Bona, R., Lindstrm, A., Rosenberg, J. & Vaughan, F., "User-level Management of Persistent Data in the Grasshopper Operating System", Universities of Adelaide and Sydney, Technical Report GH-08, 1994.

[Dearle & Hulse 2000] – Dearle, A. & Hulse, D., "Operating system support for persistent systems: past, present and future", Software – Practice and Experience, Special Issue on Persistent Object Systems, 2000, vol 30 n. 4, pp. 295 - 324

[Dearle, Rosenberg et al. 1992] - Dearle, A., Rosenberg, J., Henskens, F. A., Vaughan, F. & Maciunas, K., "An Examination of Operating System Support for Persistent Object Systems", Proceedings of the 25th Hawaii International Conference on System Sciences, vol 1, Hawaii, USA, ed V. Milutinovic and B. D. Shriver, IEEE Computer Society Press, 1992, pp. 779-789.

[Duell, Hargrove, Roman 2002] – Duell, J., Hargrove, P. & Roman., E., "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart". Berkeley Lab Technical Report (publication LBNL-54941), December 2002.

[Grove et al. 2007] –Grove, D.A., Murray, T.C., Owen, C.A., North, C.J., Jones, J.A., Beaumont, M.R. & Hopkins, B.D., "An Overview of the Annex System", Proceedings of the Annual Computer Security Applications Conference, Miami Beach, 10-15 December, 2007.

[Grove et al. in-prep] – Grove, D. A., Owen, C. A., Newby, T., North, C. J., Murray, T. C., Murray, A. P., Uzunov A. V. & Cuthbertson T. J., "The Second-Generation Annex TCB", In preperation, 2007

[Hardy 1985] – Hardy, N., "The KeyKOS Architecture", Operating Systems Review, v. 19 n. 4, October 1985, pp. 8 – 25

[Kemikli & Erdogen 1998] – Kemikli E. & Erdoğan N., "Design of a Persistent Operating System Kernel", Proceedings of the 9th Mediterranean Electrotechnical Conference (Melecon'98), Israel, 1998. Vol.II, pp.1304-1307.

[Landau 1992] – Landau, C.R., "The Checkpoint Mechanism in KeyKOS", Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems, IEEE, 1992

[Liedke 1993] – Liedtke, J., "A Persistent System in Real Use – Experiences of the First 13 Years", Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems (IWOOS '93), Asheville, NC, December 9 – 10, 1993

[Liedtke 1995] – Liedtke, J., "On u-Kernel Construction", Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95), Copper Mountain Resort, CO, December 3 – 6, 1995

[von Leitner] – von Leitner, F. n.d., "Dietlibc", Available online 21 Apr 2008: http://www.fefe.de/

[Lindstrom, Dearle et al. 1995] – Lindström, A., Dearle, A., di Bona, R., Norris, S., Rosenberg, J. & Vaughan, F., "Persistence in the Grasshopper Kernel", Proceedings of the Eighteenth Australasian Computer Science Conference, ACSC-18, ed. Ramamohanarao Kotagiri, Glenelg, South Australia, February 1995

[Rosenberg 1990] – Rosenberg, J., "The MONADS Architecture: A Layered View", The Fourth International Workshop on Persistent Object Systems, eds A. Dearle, G. M. Shaw and S. B. Zdonik, Morgan Kaufmann, 1990

[Shapiro 1998] – Shapiro, J. (1998), "The [CaprOS] Checkpoint Mechanism", Strawberry Development Group (2008), Design notes on CaprOS checkpointing mechanism, Available online 21 Apr 2008: http://capros.sourceforge.net/design-notes/Checkpoint.html

[Shapiro & Hardy 2002] – Shapiro, J.S. & Hardy, N., "EROS: A Principle-Driven Operating System from the Ground Up", IEEE Software, 2002

[Shapiro 1999] – Shapiro, J.S., "EROS: A Capability System", PhD Thesis, 1999

[Shapiro & Adams 2002] – Shapiro, J.S. & Adams, J., "Design Evolution of the EROS Single-Level Store", Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, pp. 59 – 72

[Shapiro, Farber, Smith 1996] – Shapiro, J.S., Farber, D.J., & Smith, J.M.. "State Caching in the EROS Kernel-- Implementing Efficient Orthogonal Persistence in a Pure Capability System", 7th International Workshop on Persistent Object Systems, Cape May, N.J., 1996

[Skoglund, Ceelen & Liedtke 2000] – Skoglund, E., Ceelen, C. & Liedtke, J., "Transparent Orthogonal Checkpointing through User-level Pages", Revised Papers from the 9th International Workshop on Persistent Object Systems, September 06-08, 2000, pp. 201 – 214

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION
DOCUMENT CONTROL DATA

1. PRIVACY MARKING/CAVEAT (OF DOCUMENT)

2. TITLE

The Design and Implementation of Persistence in the Annex System

3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)

Document	(U)
Title	(U)
Abstract	(U)

4. AUTHOR(S)

Anton V. Uzunov and Duncan A. Grove

5. CORPORATE AUTHOR

DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh South Australia 5111 Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO-TN-0907	AR-014-608	Technical Note	August 2009

8. FILE NUMBER	9. TASK NUMBER	10. TASK SPONSOR	11. NO. OF PAGES	12. NO. OF REFERENCES
2009/1111005	LRR 07/14	DSTO	16	25

13. URL OF ELECTRONIC VERSION

http://dsto.defence.gov.au/corporate/reports/DSTO-TN-0907.pdf

14. RELEASE AUTHORITY

Chief,
Command, Control, Communications and Intelligence Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for public release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS Yes

18. DSTO RESEARCH LIBRARY THESAURUS http://web-vic.dsto.defence.gov.au/workareas/library/resources/dsto_thesaurus.shtml

operating systems, distributed systems, software architecture, computer security

19. ABSTRACT

Orthogonal persistence in operating systems has been a topic of research for a number of years. Several commercial, as well as research projects, have implemented orthogonal persistence as an essential part of their design. Amongst these we can count the Annex software system developed at DSTO. In this technical note we consider the design and implementation of persistence in Annex against the background of two other capability-based, orthogonally persistent operating systems. This background is used to highlight the approach taken by the Annex system, and the reasons why such an approach was used. The description given discusses the current Annex prototype and explores the directions persistence in Annex will take in the future as a result of lessons learned.