DISSERTATION

ON COMPONENT-ORIENTED ACCESS CONTROL IN LIGHTWEIGHT VIRTUALIZED SERVER ENVIRONMENTS

Submitted by

Kirill Belyaev

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2017

ABSTRACT

ON COMPONENT-ORIENTED ACCESS CONTROL IN LIGHTWEIGHT VIRTUALIZED SERVER ENVIRONMENTS

With the advancements in contemporary multi-core CPU architectures and increase in main memory capacity, it is now possible for a server operating system (OS), such as Linux, to handle a large number of concurrent services on a single server instance. Individual components of such services may run in different isolated runtime environments, such as chrooted jails or related forms of OS-level containers, and may need restricted access to system resources and the ability to share data and coordinate with each other in a regulated and secure manner.

In this dissertation we describe our work on the access control framework for policy formulation, management, and enforcement that allows access to OS resources and also permits controlled data sharing and coordination for service components running in disjoint containerized environments within a single Linux OS server instance. The framework consists of two models and the policy formulation is based on the concept of policy classes for ease of administration and enforcement. The policy classes are managed and enforced through a Lightweight Policy Machine for Linux (LPM) that acts as the centralized reference monitor and provides a uniform interface for regulating access to system resources and requesting data and control objects. We present the details of our framework and also discuss the preliminary implementation and evaluation to demonstrate the feasibility of our approach.

ACKNOWLEDGEMENTS

I would like to thank my Advisor and shepherd, Professor Indrakshi Ray and Dr. David Ferraiolo at NIST for providing financial support for this research.

I want to thank my daughter Tamiris for her prayers that kept me strong along the path.

I also want to thank my friends and fellow graduate students for their help, support and advice.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

"You and I are speaking different languages, as always," said Woland, " but that does not alter the things we are talking about."

Mikhail Bulgakov, The Master and Margarita

1.1 Problem

The advancements in contemporary multi-core CPU architectures and increase in main memory capacity have greatly improved the ability of modern server operating systems (OS) such as Linux to deploy a large number of concurrent services on a single server instance [59]. Such deployments become increasingly common as large data centers and cloud-centric services become more popular [9].

The emergence of application containers [22, 40], introduction of support for kernel namespaces [43], allows a set of loosely coupled service components to be executed in isolation from each other and also from the main operating system. Service providers may lower their total cost of ownership by deploying large numbers of services on a single server instance and possibly minimize horizontal scaling of components across multiple nodes [37, 59]. Executing the individual service components in isolated containers has its benefits. If a single containerized environment is compromised by the attacker, the attack surface is limited in its scope to a single component. This, in theory, limits the possibility of disrupting the entire service. Moreover, such an approach also simplifies the management and provision of service components [8].

In conventional UNIX or Linux OS, applications can be deployed in isolated (containerized) environments, such as chrooted jails [36, 55]. Such constructs is a form of lightweight virtualization technology that provides support for OS-level virtualization [37]. Such isolated environments limit the access of the components and have the potential to offer enhanced security and performance. However, we need to regulate the access that each component has on the system resources and also control the communication and sharing of data objects between service components executing in different isolated environments.

In general, applications that need to communicate across machine boundaries use TCP/IP level communication primitives such as sockets. However, that is unnecessary for individual applications located on a single server instance [48]. Moreover, application containers currently resort to inter-container communication using available TCP/IP sockets that essentially opens wide possibilities for unregulated information flow between service components. Applications that need to communicate on a modern UNIX-like OS may use UNIX domain sockets or similar constructs. However, socket level communication is usually complex and requires the development and integration of dedicated network server functionality into an application. Modern service components also prefer information-oriented communication at the level of objects [17]. The necessity of adequate authentication primitives to prove application identity may also be non-trivial. Moreover, many localized applications may require to communicate across isolated environments but may not need access to the network I/O mechanisms. Thus, more privileges must be conferred to these applications just for the purpose of data sharing or coordination, which violates the principle of least privilege [8].

Reliance on kernel-space UNIX IPC primitives may also be problematic. First, such an IPC may be unavailable for security reasons in order to avoid potential malicious inter-component exchange on a single server instance that hosts a large number of isolated services [37]. In other words, IPC may be disabled on the level of OS kernel [35]. Aside from that, the basic IPC primitives such as various forms of pipes are simply inaccessible to components in the scope of an isolated runtime environment. That is because such primitives have been designed for centralized systems where components could have shared access to them. Second, modern applications often require more advanced, higher-level message-oriented communication that is not offered by the legacy byte-level IPC constructs. Third, UNIX IPC is bound to user/group identifiers (UID/GID) access control associations that does not provide fine-grained control at the level of individual service components [35]. Therefore, kernel-space IPC mechanisms do not offer the regulated way of inter-component interaction [8].

From access control standpoint, OS support for shared memory IPC has a number of inherent security issues, as stated above. Despite being rather complex in use, shared memory is the fastest and most common form of interprocess communication because all processes share the same piece of physical memory. Shared memory segments permit fast bidirectional communication among any number of processes. However, communicating processes must establish and follow some protocol for preventing race conditions such as overwriting information before it is read. Unfortunately, Linux OS kernel does not strictly guarantee exclusive access even if a new shared segment is created with a private flag for access protection. Also, for multiple processes to use a shared segment, they must make arrangements to use the same key, which is nontrivial. Unrelated processes can potentially access the same shared segment by specifying the same key value. Unfortunately, other processes may have also chosen the same fixed key, which could lead to conflict [50].

Optional reliance on support for Access Control Lists (ACL) in modern UNIX/Linux kernels for provision of a somewhat richer access rules to objects on the file system is also inadequate. Such a support is not enabled in mainstream kernels by default. Aside from that, the underlying file system is required to provide support for storage of access permissions via the extended attributes on file objects. That is not supported by all the file systems by default. However, the main problem with ACL usage is the fact that service components are isolated and therefore normally prevented from access to mutual data objects even in the context of basic information flow.

The usage of system-wide user-space IPC frameworks such as D-Bus [33] may also be problematic. D-Bus is the IPC and Remote Procedure Call (RPC) mechanism that primarily allows communication between GUI desktop applications (such as within KDE desktop environment) concurrently running on the same machine. D-Bus offers a relatively high-level message-oriented communication between applications on the same machine. However, it is not designed to transmit data objects such as logs. Although it is a widely accepted standard for desktop applications, D-Bus may not fit the requirements of modern server-based services. In fact, the main design objective of D-Bus is not message passing but rather process lifecycle tracking and service discovery [33]. It also does not possess a flexible access control mechanism despite its ability to transport arbitrary byte strings (but not file objects). Moreover, applications have to connect to D-Bus daemon using UNIX domain sockets or TCP sockets. Before the flow of messages begins, two applications must also authenticate themselves which adds extra complexity layer to the communication. However, the more pressing problem is the possibility that user-space D-Bus daemon in line with kernel-space IPC may be disabled on the server node for security reasons. Moreover, system-wide communication resources such as global UNIX domain socket for the D-Bus daemon may be inaccessible for applications running in isolated environments [8].

1.2 Solution

In this dissertation we describe the access control framework whose objective is to give each component minimum privileges to system resources on a need-to-know basis and also provide regulated coordination and data sharing across service components that execute in isolated environments within the context of a single OS. We introduce such a component-oriented access control framework referred to as *Lightweight Policy Machine for Linux* [8] middleware (and shorten it to just LPM in the rest of the dissertation) to address the identified challenges. Our LPM allows the formulation, management and enforcement of access policies to OS resources for individual service components and it also allows regulated inter-component communication. Our uniform framework provides a coherent business logic interface that is available to administrative personnel to manage such an access control for a set of services.

The framework consists of two types of policy objects referred to as policy classes, namely, *capabilities classes* and *communicative classes*. Each capability class consists of policies that are associated with a set of Linux capabilities [44]. The capabilities classes differ from each other on the basis of capabilities they possess. Each service component is placed in at most one capabilities class. The OS resources that the component can access depends on the Linux capabilities associated with that class. Our implementation provides a way by which Linux capabilities can be administered to the services executing in isolated environments.

Communicative classes are needed for communication of components that belong to different isolated environments. Each communicative class consists of policies for inter-component interaction. Our implementation provides a way by which such communication can be administered to a set of services.

However, we also need to implement a mechanism for the enforcement of communication policies within such communicative classes. We adapt the generative communication paradigm introduced by Linda programming model [29] which uses the concept of tuple spaces for process communication. However, the traditional tuple spaces lack any security features and also have operational limitations [7, 8]. We enhance the original paradigm and also provide enforcement rules such that only components belonging to the same communicative class can communicate using this approach. Our LPM middleware allows a regulated way of coordinating and data sharing among components using tuple spaces. Note that, such coordination and sharing will be allowed even if each of the components executes under different system UIDs and GIDs.

1.3 Contribution

The major contribution of the conducted research is a component-oriented access control framework that provides implementation of two policy classes models. We now provide a list of contributions for the research:

1. A state-of-the-art survey on access control models in the scope of our work
2. A capabilities classes model for components that administers policies for access to system resources using Linux capabilities

3. A communicative classes model for components that administers policies for inter-component interaction

4. An adaptation of tuple space paradigm for enforcement of policies for communicative classes

5. A research prototype that provides a reference implementation of the unified access control framework

6. An evaluation of the proposed framework

1.4 Dissertation Structure

The rest of the dissertation is organized as follows. Chapter 2 covers a large body of related work. Chapter 3 gives an overview of our framework that consists of two types of policy classes. Chapter 4 provides the details of the inter-component communication architecture. Chapter 5 provides the details of tuple space transactions for secure coordination and data sharing between service components. Chapter 6 gives system architecture. Chapter 7 demonstrates the feasibility of our approach by describing the implementation experience with a focus on tuple space paradigm. Chapter 8 provides discussion on properties of our framework in regard of its security and architecture. Chapter 9 concludes the dissertation.

Chapter 2

Background and Related Work

In this chapter we present the result of a systematic literature review we conducted on related research area. A systematic review is important for research activities since it summarizes existing techniques concerning a research interest and identifies further research directions. The purpose of the review described in this chapter is to compare current solutions and identify their (possible) limitations through a systematic evaluation.

Due to the vast scope of access control research that has been developed for the past several decades we limit our study of related research literature to the existing efforts in the UNIX/Linux based operating systems which is more closely related to our proposed work. In addition, we will also reference literature outside the OS domain if it cross-cuts with our proposed directions.

2.1 Access Control Fundamentals

The basic entities in access control are subjects, objects, and operations. Subjects are active entities and wish to perform operations on objects. Access control limits the operations that a subject can perform on objects. The operations that a subject can perform on an object are termed as the access rights of the subject.

The principle of least privilege is often times followed when conferring access rights to a subject which requires that the subject be given the minimal access rights that are needed to accomplish her task [12].

A simple but very fast access control mechanism is a two dimensional *matrix*, stating which subjects may access which objects. Yet, matrices have certain drawbacks with respect to their maintenance and storage. Another mechanism is an *Access Control List* (ACL), which represents a list associated with the concerned object. that includes all the subjects who can access the object and their access modes.

2.2 Previous Work

Traditionally, UNIX/Linux environments supported Discretionary Access Control (DAC) which allows read, write, and execute permissions for three categories of users, namely, owners, groups, and all others for managing access to files and directories in the user-space. Another type of supported access control is based on the Mandatory Access Control (MAC) designed to enforce system policies: system administrators specify policies which are checked via run-time hooks inserted into many places in the operating system's kernel. For managing access to system resources, typically superuser privileges are needed. Each file in the system is annotated with a numerical ownership UID. Applications needing access to system resources temporarily acquire the privilege of the superuser. The superuser is assigned UID = 0 – a process executing with this UID can bypass all access control rules. This simple model violates the principle of least privilege.

Another approach in order to model extents of access privileges is called Role based Access Control (RBAC) [24, 26, 27]. RBAC was proposed to simplify access control management. By insertion of a new level of abstraction, access privileges are not directly applied to user accounts, but to *roles*. The user's extent of access privileges is defined by the roles she is permitted to utilize. Depending on her designated actions, a user needs to switch her active role and therefore performs tasks with different sets of privileges, whereas a role can be utilized by several users. Users may be allowed to use several roles simultaneously, or they are restricted to only one active role at a time. Further, roles can reflect hierarchical relationships. The concept of RBAC can be combined with either DAC or MAC mechanisms to complement the deployment in its multi-user environment settings.

Researchers have proposed Domain and Type Enforcement (DTE) [3, 4, 31] for Linux and UNIX environments. Type enforcement views a system as a collection of active entities (subjects) and a collection of passive entities (objects) [3, 4]. The underlying idea is to classify a system's subjects and objects along a set of equivalence classes, whereas classes of *subjects* are associated with invariant access control attributes called *domains* and classes of *objects* are associated with attributes called *types*. The identifiers of domains and types are referred to as *labels*. Legitimate accesses are defined by the association of domains and types or associations between domains. In DTE, a process carries with itself the label of the domain in which it runs, and this determines its access rights. A process can enter a new domain (and hence change its access rights) only upon execution of an executable file associated with that process. Therefore, the process must be invoked in order to enter the domain [3]. DTE is designed to provide MAC to protect a system from subverted superuser processes as the access control is based on enforceable rule sets. The DTE model, in contrast to other UNIX access control approaches, avoids the concept of users and only concentrates on applications [3, 4]. Our work, in line with DTE, also concentrates on access control requirements of applications and their interaction. We also express policies in a human readable form. However, our LPM is entirely resident in user-space in contrast to DTE that offers kernel level solution. Moreover, we target the access control requirements necessary for the manageable deployment of large numbers of localized isolated services under unprivileged UIDs in isolated environments, such as chrooted jails and application containers. Such environments were outside the scope of DTE.

Security-Enhanced Linux (SELinux) [45, 60] allows for the specification and enforcement of MAC policies at the kernel level. SELinux uses the Linux Security Modules (LSM) [70] hooks in the kernel to implement its policy. The SELinux architecture is based on the Generalized Framework for Access Control (GFAC) proposed by Abrams [1] and LaPadula [39] and supports multiple security models. In SELinux, the policy server makes access control decisions and the object managers are responsible for enforcing access control decisions. It provides a policy description language for expressing various types of policies. The three available standard policies for SELinux are all based on the approach of Type Enforcement while additionally supplying RBAC and Multilevel Security (MLS) mechanisms. The Type Enforcement implementation of SELinux does not distinguish between domains and types and domain depicts a type concerning a subject. In order to enable standard Linux program code to be mapped into a Type Enforcement environment without making any changes necessary, the SELinux policy language supplies a method to automatically initiate domain transitions. In relation to RBAC, where access privileges are not directly applied to user accounts, but rather to roles [24, 26], the user's extent of access privileges is defined by the roles she is permitted to utilize. SELinux supports the concepts of roles and users but is not intended for enforcing policies at the level of individual applications. A role is mapped to the Type Enforcement as a set of domains. In respect to MLS, where information is processed at different security levels, access permissions are given to users with different security clearances and needs-to-know requirements, to prevent users from obtaining access to information for which they lack authorization. Policy description and configuration in SELinux is non-trivial because of the relationships between multiple models of SELinux and consequently it is a little challenging to use [72]. Our work complements the efforts of SELinux in that it provides access control for isolated applications in user-space.

The Medusa DS9 security system [47] is another security extension for Linux kernels. Medusa DS9 consists of a patch to the Linux kernel (monitor) and a user-space daemon (security decision center) called Constable that acts as an authorization server that is invoked by the kernel to authorize operations. Medusa provides a language for specifying the policies and the authorization server is implemented as an interpreter for this configuration language. Medusa claims to be able to implement different security models at user-space level. The user-space implementation of the authorization server allows to port Medusa to new Linux kernel versions. This gives Medusa an advantage over alternative approaches that require custom-built kernel with very specific patching requirements. The virtual spaces (VS) model, used in Medusa, replaces the access control matrix which is hard to implement in a real operating system. The objects and subjects are separated into a finite number of domains, named virtual spaces. Each object can be a member of any number of virtual spaces at one time. Each subject is assigned a set of abilities, one for each access type. Ability is a set of virtual spaces. Access is granted, when the object and the subject share at least one common virtual space for the given access type. Note that the term virtual space has no direct association with application containers or related isolated environments such as chrooted jails. Similar to Medusa [47], we propose to incorporate the access control modeling and decision control in user-space with robust and expressive persistence layer which allows high interoperability and porting of the framework to any general-purpose Linux kernels without a requirement for custom kernel patching [38]. However, Medusa does not offer support for managing Linux capabilities and does not have a mechanism for regulated inter-component interaction across isolated (containerized) environments.

The Rule Set Based Access Control (RSBAC) [53] attempts to bring more advanced access control model to Linux based server systems. RSBAC is an open source security extension for current Linux kernels. The kernel-based patch provides high level of security to the Linux kernel and operating environment. In RSBAC model, the access control system of the Linux kernel is divided into the Access Control Enforcement Facility (AEF), Access Control Decision Facility (ADF), and the Access Control Information (ACI) module. ADF implements the system's mandatory security policies and a metapolicy to decide whether processes' requests satisfy those security policies. AEF uses the ADF decisions to enforce the operations of system call functions. All RSBAC framework components are hard-linked into the custom-built Linux kernel. RSBAC supports divergent security policies implemented as modules in a single framework. However, the framework does not have a mature representation format to provide a unified way of modeling and expressing the policies for all the diverse policy modules that the framework claims to support. This limits its wide-spread adaptability. In contrast to RSBAC, our work provides domain-specific expressive policy formulation framework and is implemented in user-space that allows it to be deployed on any Linux server system.

The Grsecurity package [30] is a composition of Linux kernel patches combined with a small set of control programs. The package aims to harden known vulnerabilities in the Linux system while paying special attention to privilege escalation and root exploits. Grsecurity provides a MAC mechanism based on ACL and RBAC capabilities combined with trusted path execution, that allows to limit the right of program executions to certain specified file names. The set of patches provides protection mechanisms for file systems, executables and networks. Grsecurity can harden the chroot environment against certain known attacks, prevent unprivileged users from reading kernel information and is able to limit and isolate the operating system's process view. In short, Grsecurity hardens the Linux operating system and its proprietary mechanisms while restraining system entities like users and processes. It does this by placing additional logic on the Linux kernel and also alters the kernel's own mechanisms to comply with the desired behavior. Grsecurity does not follow any formal model of security and access control, but emerged as a composition of countermeasures against several known weaknesses, vulnerabilities, or concrete attacks. Therefore, it lacks a general systematic approach or comprehensive formal model. Consequently, analysis of the security properties of the various mechanisms is non-trivial despite recent attempts to develop techniques for its formal analysis [15].

The decomposition of complex, legacy, monolithic applications into fine-grained, least-privilege memory compartments is supported by Bittau et al. [13] that provides programming primitives to allow the creation of compartments with default-deny semantics. Specifically, Bittau et al. offer a flexible memory tagging scheme that grants a compartment memory privileges at a memory-tag granularity for the related memory objects. Contrarily to Bittau et al., in our work we deal with information flow control at the granularity of stand-alone OS processes. Efforts at compartmentalization have also been recently observed in UNIX systems with Capsicum framework [63, 64] that represents a lightweight operating system capability and sandbox framework planned for inclusion in FreeBSD. It supports compartmentalization of monolithic UNIX applications into logical applications. By adding capability primitives to standard UNIX API, it gives application developers a path to satisfying the requirement of least-privilege. Privilege separation also referred to as compartmentalization is addressed by introducing capabilities and capabilities mode. However, these capabilities should not be confused with Linux capabilities, which are coarse-grained privileges that are not associated with objects and cannot be transferred across processes. Capsicum capabilities are an extension of UNIX file descriptors and reflect rights on specific objects such as files and sockets. In contrast to our work, that adds a purely user-space management layer on top of the existing kernel level support for capabilities, Capsicum requires application modifications to exploit these new security functionalities. Moreover, extensive kernel changes are needed in Linux and FreeBSD to provide support for Capsicum capabilities. However, our access control framework also permits the management of compartmentalization for a single monolithic application through the notion of policy classes and regulation of control and data flow interaction between such isolated compartments implemented in the form of containerized OS processes.

The application-level access control is emphasized in Decentralized Information Flow Control (DIFC) [51]. DIFC allows application writers to control how data flows between the pieces of an application and the outside world. As applied to privacy, DIFC allows untrusted software to compute with private data while trusted security code controls the release of that data. As applied to integrity, DIFC allows trusted code to protect untrusted software from unexpected malicious inputs. In either case, only bugs in the trusted code, which tend to be small and isolated, can lead to security violations. Current DIFC systems that run on commodity hardware can be broadly categorized into two types: language-level and operating system-level DIFC [38, 56]. Language level solutions provide no guarantees against security violations on system resources, like files and sockets. Operating system solutions can mediate accesses to system resources, but are inefficient at monitoring the flow of information through fine-grained program data structures [56]. DIFC efforts like Flume [38] and Laminar [56] generally employ an interposition layer that replaces system calls with interprocess communication to the reference monitor, which enforces data flow policies and performs safe operations on the application's behalf. Regardless of the level of implementation (Language level, OS level or both as in the case of Laminar) the security and privacy guarantees come at a price – application code has to be modified and performance overheads are incurred on the modified binaries. Moreover, the complexities of rewriting parts of the application code to use the DIFC security guarantees are not trivial and require extensive API and domain knowledge [56]. These challenges, despite the provided benefits, limits the widespread applicability of this approach. Our solution allows to divide the information flow between service components into data and control planes that are regulated through the user-space reference monitor. Therefore, no modification to OS kernel is required. The rewrite of existing applications for utilization of data flow may not be necessary, since a separate flow requesting application that leverages our TSL can handle such a task and deliver the replica of a data object to unmodified application [7].

Application-defined decentralized access control (DCAC) for Linux has been recently proposed by Xu et al. [72] that allows ordinary users to perform administrative operations enabling isolation and privilege separation for applications. In DCAC, applications control their privileges with a mechanism implemented and enforced by the operating system, but without centralized policy enforcement and administration. DCAC is configurable on a per-user basis only [72]. DCAC is based on the concept of attributes, which can represent different types of principals including users, groups, applications, and application components. The attributes can be arranged in a hierarchy, where the parent attribute has a superset of the privileges of the child attribute. The regular users can manage the principals by regulating the hierarchy without the need for a system administrator. The process carries an attribute set which is inherited by process control events. Each object has an access control list that specifies the rules for each access mode based on conditions of its attributes. If the attributes of a process match these conditions, it is permitted to access the object in the specified mode. The authors also describe the notion of attribute gateways that provide controlled privilege escalation. The objective of DCAC is decentralization with facilitation of data sharing between users in a multi-user environment. Our work is designed for a different deployment domain – provision of access control framework for isolated applications where access control has to be managed and enforced by the centralized user-space reference monitor at the granularity of individual applications using expressive high-level policy language without a need to modify OS kernel.

In the realm of enterprise computing applications running on top of Microsoft Windows Server infrastructure, the aim is to provide data services (DSs) to its users. Examples of such services are email, workflow management, and calendar management. NIST Policy Machine (PM) [23, 25] was proposed so that a single access control framework can control and manage the individual capabilities of the different DSs. Each DS operates in its own environment which has its unique rules for specifying and analyzing access control. The PM tries to provide an enterprise operating environment for multi-user base in which policies can be specified and enforced in a uniform manner. The PM follows the attribute-based access control model and can express a wide range of policies that arise in enterprise applications and also provides the mechanism for enforcing such policies. Our research efforts are similar to NIST PM [25] since it offers the policy management and mediation of data services through a centralized reference monitor. However, our access control goals are different. We do not attempt to model user-level policies as done by NIST PM. Our framework, on the other hand, provides the mechanism exclusively for controlled inter-application collaboration and coordination of localized service components across Linux-based isolated runtime environments that also regulates access to system resources based on the principle of least privilege. Note that, the importance of such a mechanism, that is not currently present in NIST PM is acknowledged by its researchers [25].

In the mobile devices environment, Android Intents [19] offers message passing infrastructure for sandboxed applications; this is similar in objectives to our tuple space communication paradigm proposed for the enforcement of regulated inter-application communication for isolated service components using our model of communicative policy classes. Under the Android security model, each application runs in its own process with a low-privilege user ID (UID), and applications can only access their own files by default. That is similar to our deployment scheme. Our notion of capabilities policy classes is similar to Android permissions that are also based on the principle of least privilege. Permissions are labels, attached to application to declare which sensitive resources it wants to access. However, Android permissions are granted at the user's discretion [2]. Our server-oriented centralized framework deterministically enforces capabilities and information flow accesses between isolated service components without consent of such components based on the concept of policy classes. Despite their default isolation, Android applications can optionally communicate via message passing. However, communication can become an attack vector since the Intent messages can be vulnerable to passive eavesdropping or active denial of service attacks [19]. We eliminate such a possibility in our proposed communication architecture due to the virtue of tuple space communication that offers connectionless inter-application communication as discussed in Chapter 4. Malicious applications cannot infer on or intercept the inter-application traffic of other services deployed on the same server instance because communication is performed via isolated tuple spaces on a file system. Moreover, message spoofing is also precluded by our architecture since the enforcement of message passing is conducted via the centralized LPM reference monitor that regulates the delivery of messages according to its policies store.

Our work also bears resemblance to the Law-Governed Interactions (LGI) proposed by Minsky et al. [48, 49], which allows an open group of distributed active entities to interact with each other under a specified policy called the law of the group. The inter-component communication in our work is proposed in the same manner via the tuple space using the Tuple Space Controller integrated in our centralized LPM reference monitor that has complete control over inter-component interaction [21, 49].

The tuple space model as a type of shared memory, originally introduced by Linda [29] has been widely adapted for parallel programming tasks [16, 67], support for language-level coordination [62], multi-agent systems [17, 21] and distributed systems [46, 48, 49] in general. Several commercial implementations of tuple space paradigm have also been developed in the past, targeting highly parallel and High-Performance Computing (HPC) applications with enhanced support for tuple persistence, distribution across network nodes and matching capabilities [16]. We have adapted the original Linda model to serve the requirements of secure inter-component communication within a single-node OS with dedicated file system-level space per component. In comparison to traditional tuple spaces that allow potentially thousands of tuples per single space, our search complexity is minimal since only at most two tuples are allowed to be present in a given tuple space. That is a deliberate restriction imposed by the necessity of providing basic DoS protection and resource preservation when dealing with concurrent transfers of large data objects made possible through our LPM middleware. As covered in Chapter 4, the original paradigm has a number of resource-oriented limitations [16] and does not offer security guarantees. For that matter, many researchers [17, 48, 62, 73] have conducted adaptation of the original tuple space model to fit the domain-specific requirements. The LighTS tuple space framework [5] is somewhat similar to our work in a sense that it also provides localized variant of a tuple space per application with a possibility of persistence. However, it has adapted the original operations on Linda tuple space for use in context-aware applications. LighTS offers support for aggregated content matching of tuple objects and other advanced functionality such as matches on value ranges and support for uncertain matches. Our adaptation allows coordination and collaboration between isolated service components based on precise content matching on a set of tuple fields. Our model allows a mixed mode of information transfer between service components – tuples can contain actual language-level objects or could be used to replicate larger data objects such as large ASCII file objects. Note, that no restriction on types of replicated objects exists in our TSL implementation – aside from ASCII objects, a complete byte-level replication is entirely possible. Therefore, data objects, such as images, could be potentially replicated between service components. We also enable dual planes of inter-component communication – components can communicate using a control plane, data plane or both. To the best of our knowledge, we offer the first persistent tuple space implementation that facilitates the regulated inter-component communication without a need for applications to share a common memory address space or requirements for address space mapping mechanisms [7, 11].

Chapter 3

Component-Oriented Access Control Framework

This chapter provides a detailed overview of the developed access control framework. We first present a real-world motivating example where a single service can consist of several components, each deployed in isolation [7, 8].

3.1 Motivating Example

Consider a service deployment scenario illustrated in Figure 3.1 that is taken from a real-world telecom service provider [52]. A Linux server has three applications, namely, *Squid Web Cache Server*, *Squid Log Analyzer*, and *HTTP Web Server*, deployed in three separate isolated environments (chrooted jail directories), each under a distinct unprivileged user identifier (UID). Combined, all three applications represent individual components of a single service – ISP web caching that caches Internet HTTP traffic of a large customer base to minimize the utilization of ISP's Internet backbone. *Squid Web Cache Server* component generates daily operational cache logs in its respective runtime environment. *Squid Log Analyzer* component needs to perform data analytics on those operational log files on a daily basis. It then creates analytical results in the form of HTML files that need to be accessible by the *HTTP Web Server* component to be available through the web browser for administrative personnel. Each component may also need access to system resources. For example, *Squid Web Cache Server* component needs access to network socket I/O resources and some advanced networking capabilities of the Linux kernel [42] in order to operate.

As the example above demonstrates each service component has customized requirements for access to various OS resources and also specific needs with respect to communication with other components. We implement a new object-oriented access control framework based on the notion of policy classes that regulates access to OS resources and also permits regulated and fine-grained inter-component communication [8].

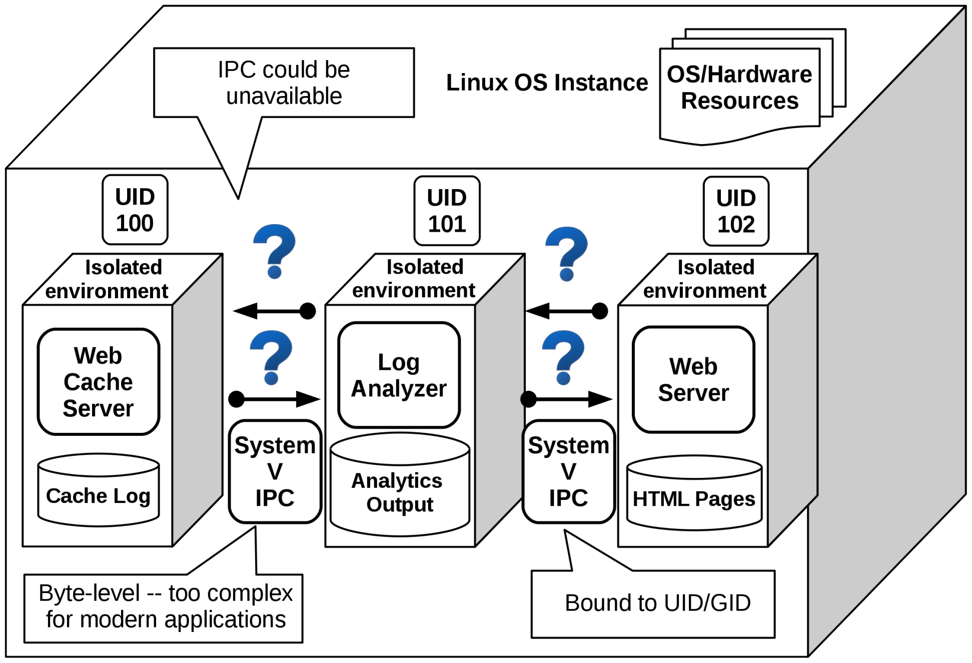


Figure 3.1: Problems of Controlled Sharing illustrated in Web Cache Service Example

3.2 Component Formalization

We begin by providing a definition of a component and then give the formalisms for the notion of policy classes that forms the basis of our access control framework.

Definition 3.2.1 [Component:] *A* component *is an application process in the Linux environment executing with non super-user privileges (UID $\neq$ 0) and having access to program executable resources such as memory, network resources, CPU cycles, and files in its directory structure.*

Let $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ where $\mathcal{C}$ is the set of components and c_j denotes a component where $1 \leq j \leq n$. Note that, two components c_i and c_j where $1 \leq i, j \leq n$ may have the same UID if they are executing in the same isolated containerized environment. In such a case, the two components will have access to the same set of resources [10].

In the following sections, we describe how to confer fine-grained access control to a component with regards to accessing system resources through the policies of the component's capabilities class. We then describe how components belonging to isolated containers can coordinate their execution and get access to each others' resources through the policies specified in the component's communicative policy class.

3.3 Capabilities Class

The individual containerized components of a service often need regulated access to OS resources. In the Linux environments, the application runtime access control to the underlying OS resources has been traditionally regulated by root privileges which provides all permissions on system and user resources. The applications regulated by root privileges run with a special user identifier (UID = 0) that allows them to bypass access control checks. However, giving root permissions to an application violates the principle of least privilege and can be misused. Subsequently, in Linux kernels starting from version 2.1, the root privilege was partitioned into disjoint capabilities [41]. Instead of providing “root" or “non-root" access, capabilities provide more fine-grained access control – full root permissions no longer need to be granted to processes accessing some OS resources. For example, a web server daemon needs to listen to port 80. Instead of giving this daemon all root permissions, we can set a capability on the web server binary, like *CAP_NET_BIND_SERVICE* using which it can open up port 80 (and 443 for HTTPS) and listen to it, like intended. The new emerging concept of Linux application containers such as Docker service [22] and CoreOS [20] heavily leverages the Linux capabilities model. Despite the incorporation of capabilities in mainstream Linux and application containers, capabilities management in user-space is challenging [32] and is only addressed in our work [8].

This is partly because the amendment of kernel-space capabilities does not provide persistent storage of state change in regard to capabilities being removed or added to a particular application. Consequently, such a deficiency leads to inability to track, identify and manage individual capabilities and their sets for a large number of applications and service components in a timely manner [32].

In service provider context, the only capabilities, realistically required by the containerized services deployed on the same OS server instance are the capabilities associated with access to

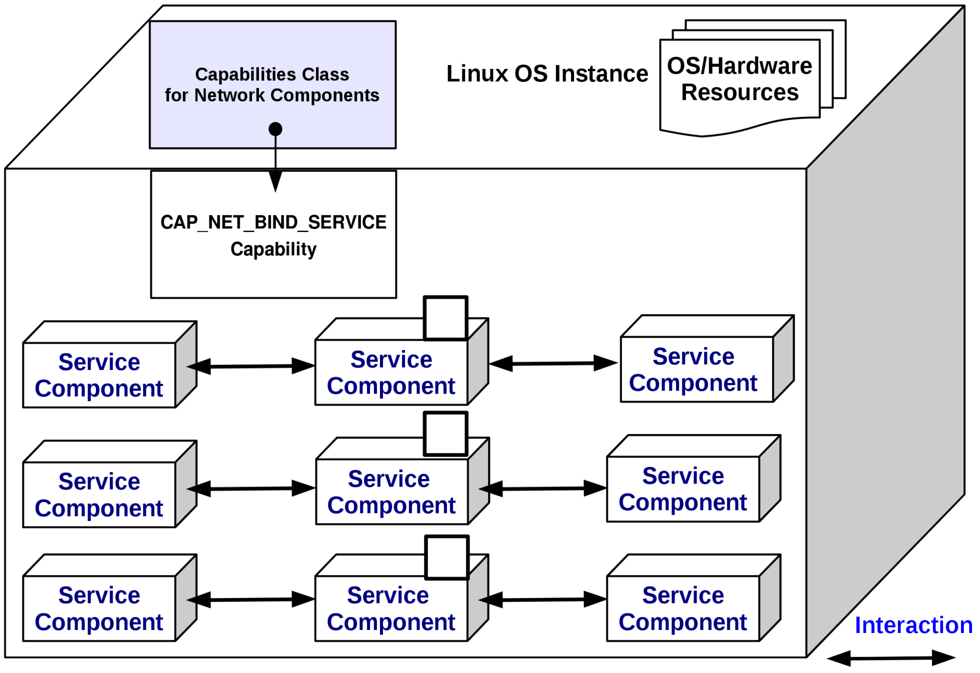

Capabilities Class for Network Components
Linux OS Instance
OS/Hardware Resources
CAP_NET_BIND_SERVICE Capability
Service Component
Interaction

Figure 3.2: Capabilities Class for Components Accessing the Network

OS network communication resources. That is because some customer facing service components, such as web server component might need access to this type of OS resources. The rest of Linux capabilities mostly represent highly specialized super-user capabilities that are of no interest to general-purpose containerized service deployments and could be even dangerous in the multi-service settings. For instance, CAP_CHOWN capability gives unnecessary capability to allow making changes to the file UIDs and GIDs. Another capability, CAP_DAC_OVERRIDE allows to bypass file read, write and execute permission checks that poses imminent security threat to the OS environment. Another dangerous CAP_SYS_PTRACE capability allows a process to look up information about another process, read and write the memory of that process, and attach to (or trace) that process in order to debug it, or analyze its behavior. This gives total control over the process being traced. Therefore, individual service components should be conferred with specific capability(s) based on the principle of least privilege.

We introduce the notion of a *capabilities class* that is associated with a set of Linux capabilities. Each capabilities class can have one or more service components. The components in such a class have all the capabilities associated with this class and can therefore access the same set of OS resources as illustrated in Figure 3.2. Each service component can belong to at most one capabilities class, but a class can have multiple components. Note that, two distinct capabilities classes will be associated with different sets of Linux capabilities.

We now describe how to manage Linux capabilities using the notion of a *capabilities class* that we describe below.

Definition 3.3.1 [Capabilities Policy Class:] *A* capabilities policy class *is one whose members are components that have access to the same Linux capabilities. Each capabilities policy class has an attribute called* capability *that lists the Linux capabilities associated with the class.*

Let $\mathcal{CP} = \{cp_1, cp_2, \ldots, cp_m\}$ be the set of capabilities policy classes. Let $cp_i(cap)$ be the set of Linux capabilities associated with class cp_i. No two distinct capabilities classes can have the same set of Linux capabilities, that is, for $i \neq j$, $cp_i(cap) \neq cp_j(cap)$. Each capabilities class cp_i can have one or more components denoted as $cp_i = \{c_p, c_q, c_r, \ldots, c_s\}$. Components c_p and c_q belonging to the same capability class cp_i can access the set of OS resources as defined by $cp_i(cap)$. Each component can belong to at most one capabilities class, that is, $c_s \in cp_i \Rightarrow c_s \notin cp_j$ where $i \neq j$. If component c_k does not need access to any Linux capabilities, it is not a member of any cp_j, denoted as $c_k \notin cp_j$ where $1 \leq j \leq m$.

As a part of our unified framework, capabilities class supports a set of management operations on it [8]. The following high-level operations are supported by our capabilities class model:

1. *create* a capabilities policy class,

2. *add/remove* capabilities to/from a policy class,

3. *show* capabilities in a policy class,

4. *add/remove* components to/from a policy class, and

5. *show/count* components in a policy class.

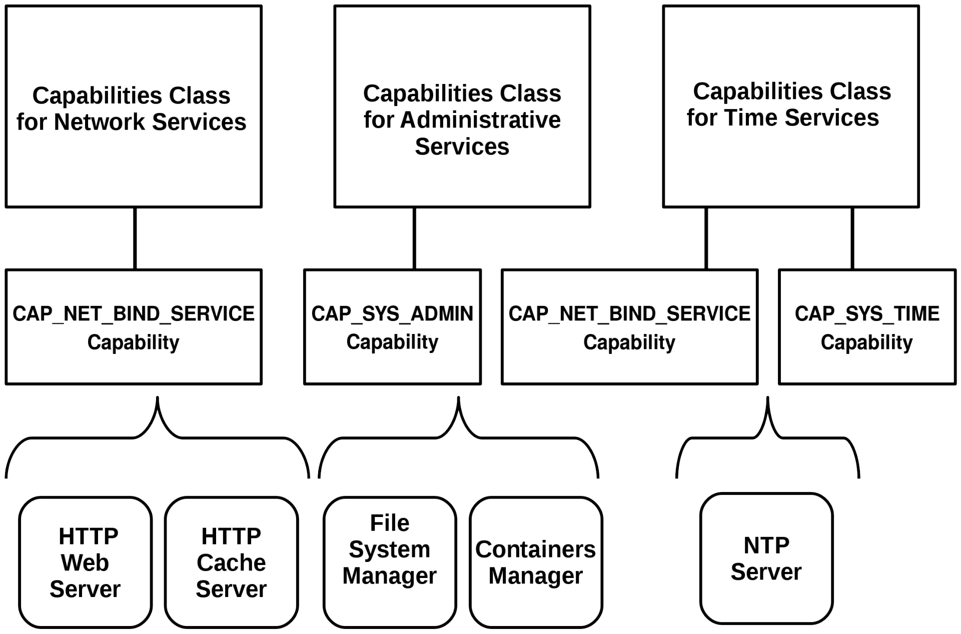

Capabilities Class for Network Services
Capabilities Class for Administrative Services
Capabilities Class for Time Services
CAP_NET_BIND_SERVICE Capability
CAP_SYS_ADMIN Capability
CAP_NET_BIND_SERVICE Capability
CAP_SYS_TIME Capability
HTTP Web Server
HTTP Cache Server
File System Manager
Containers Manager
NTP Server

Figure 3.3: Capabilities Policy Classes

Note, that the necessity to associate a single service component with at most one capabilities class is driven by the fact that a component in such a policy class inherits all the capabilities associated with such a class and can therefore access the same set of OS resources [8]. Since a single capabilities policy class can have multiple components, we can place all the service components into a single class depending on the service category [9]. For instance, all components that require access to specific network socket I/O can be placed into a single capabilities class that grants them such a capability. Some service categories may require access to more capabilities than others. For instance, a web server may belong to a separate class that delegates a single CAP_NET_BIND_SERVICE capability to gain access permission to network sockets while a Network Time Protocol (NTP) server requires both CAP_NET_BIND_SERVICE capability as well as CAP_SYS_TIME capability to change system time on the machine. In such cases, we place the two components into different capabilities classes based on the principle of least privilege.

Such a separation is further supported by the fact that the capabilities enforcement is expressed through the notion of capabilities bits assigned to a binary on the file system [8, 44] – OS kernel cannot assign overlapping sets of capabilities to a single binary without overwriting some of them. Thus, each component has to belong to a distinct capabilities class in accordance with service categorization (classification) [10]. Such an enforcement is depicted in Figure 3.3.

3.4 Communicative Class

In order to address the requirements of the regulated communication between isolated service components, we introduce the notion of a *communicative class* that consists of a group of applications (service components) that reside in different isolated environments and need to collaborate and/or coordinate with each other in order to provide a service offering [8].

Definition 3.4.1 [Communicative Policy Class:] *A* communicative policy class *is one whose members are components belonging to isolated environments that need to communicate to provide some service.*

Let $\mathcal{CM} = \{cm_1, cm_2, \dots cm_t\}$ be the set of communicative policy classes. Each communicative policy class cm_i consists of two or more components that together offer some higher level service; this is denoted as $cm_i = \{c_b, c_d, c_f, \dots, c_g\}$. Each component c_b cannot be a part of multiple communicative classes, that is $c_b \in cm_i \Rightarrow c_b \notin cm_j$ where $i \neq j$. If component c_d does not need to coordinate or collaborate with other components in different isolated environments, then it may not belong to any communicative class, denoted as $c_d \notin cm_i$ where $1 \leq i \leq t$.

Our notion of communicative class is different from the conventional notion of UNIX groups. In the conventional groups, the privileges assigned to a group are applied uniformly to all members of that group. In this case, we allow controlled sharing of private data objects among members of the communicative class via object *replication*. Such a sharing can be very fine-grained and it may be *unidirectional* – an isolated component can request a replica of a data object belonging to another isolated component but not the other way around.

Some service components may require *bidirectional* access requests where both components can request replicas of respective data objects. Such types of possible information flow are depicted

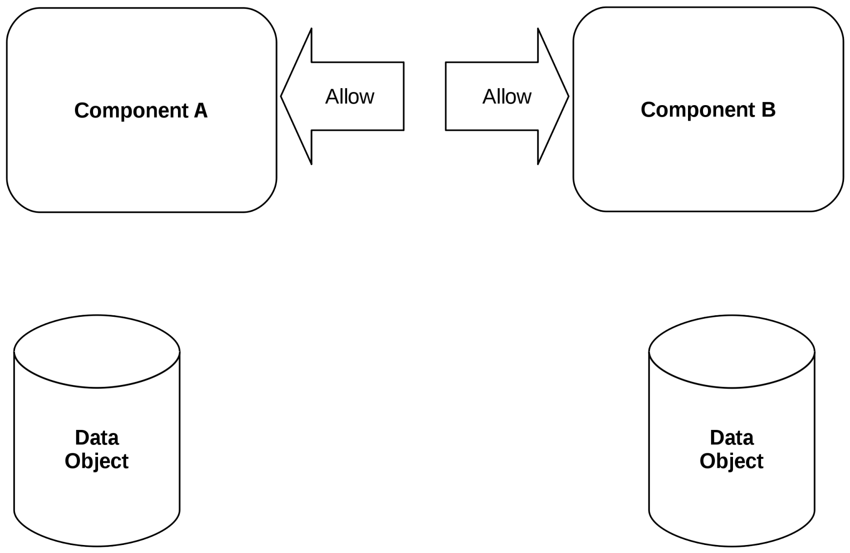


Figure 3.4: Bidirectional Data Flow Control between Isolated Service Components

in Figures 3.4, 3.5 and 3.6 where 'Allow' arrow denotes the granted request for a replicated data object in the direction of an arrow, while 'Disallow' one signifies the forbidden request.

Implementing such rules may be non-trivial as isolated environments are non-traversable due to isolation properties. This necessitates proposing alternative communication constructs. The access control policies of a communicative class specify how the individual components in such a class can request a replica of mutual data objects. Note that, the individual components in a class may have different Linux capabilities – therefore, they may belong to the same communicative class but to different capabilities classes. Only components within the same communicative class can communicate and therefore communication across different communicative classes is forbidden. Such a regulation is well-suited for multiple services hosted on a single server instance. The assignment of individual service to a separate class facilitates the fine-grained specification of communication policies between various isolated service components [7, 8].

A more complicated flow communication pattern is depicted in Figure 3.7 where a single unidirectional data pipeline is depicted. Such a flow pattern is expressed via a *composition* of binary communication policies between individual pairs of components. In this case, such a composition

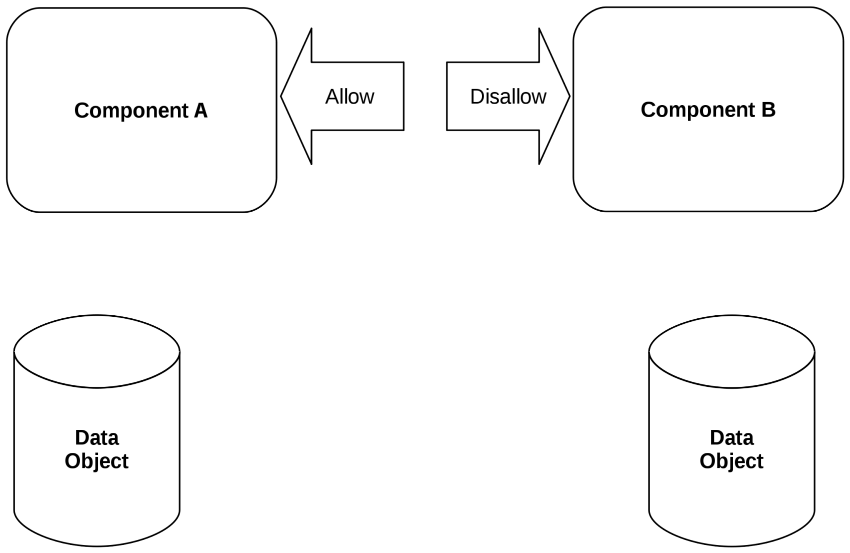


Figure 3.5: Unidirectional Data Flow Control between Isolated Service Components

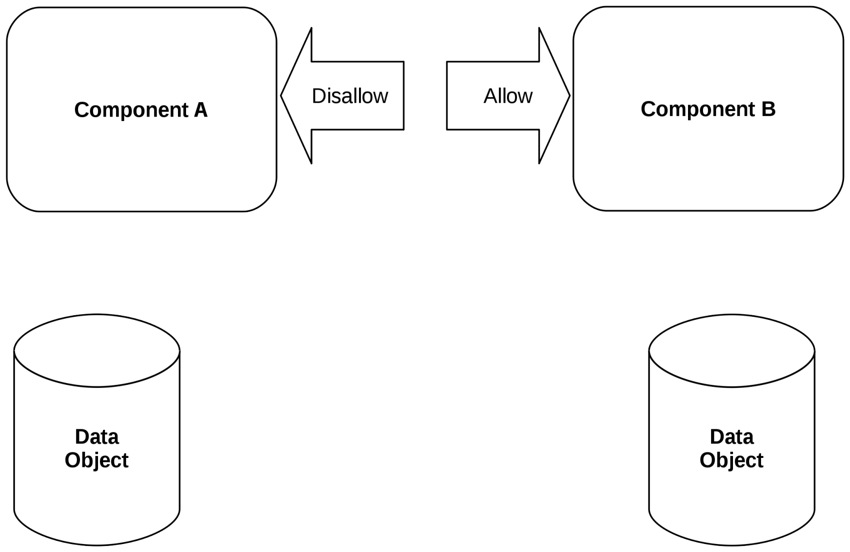


Figure 3.6: Unidirectional Data Flow Control between Isolated Service Components

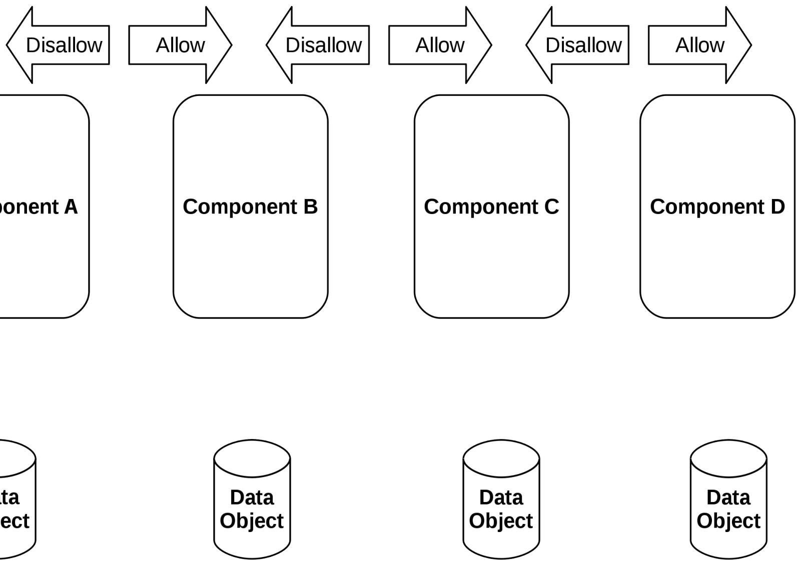


Figure 3.7: Unidirectional Pipelined Data Flow Control between Isolated Service Components

allows each subsequent component to obtain a replica from a counter-component of such a multi-component service but not the other way around. For instance, Component B is granted a replica of an object that belongs to Component A. However, Component C is not granted with such an access policy but can request a replica of an object that belongs to Component B. Consequently, a fully bidirectional version of such a pipelined pattern is depicted in Figure 3.8.

The construct of communicative class is designed to support the following communication patterns between the components in a single class.

Coordination – often components acting as a single service do not require direct access to mutual data objects or their replicas but rather need an exchange of messages to perform coordinated invocation or maintain collective state [8]. However, some services may need coordination in support of certain data flows. For instance, in the context of introduced web caching service, a Web Cache Server component may experience excessive load from users' HTTP requests and consequently generate large cache log objects. The size of such objects has to be kept at threshold that allows efficient and fast processing that spares system resources such as CPU and RAM. Therefore, the objects are rotated by the Web Cache Server component at

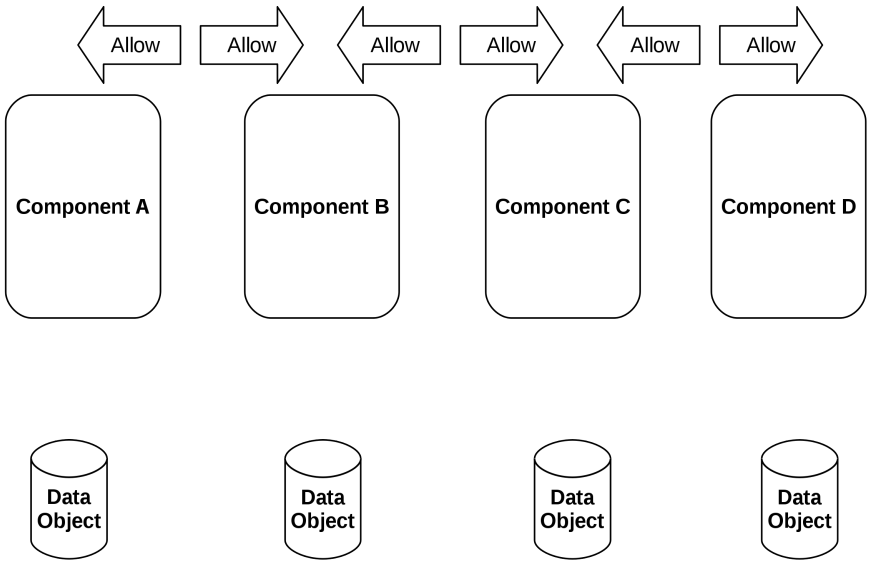


Figure 3.8: Bidirectional Pipelined Data Flow Control between Isolated Service Components

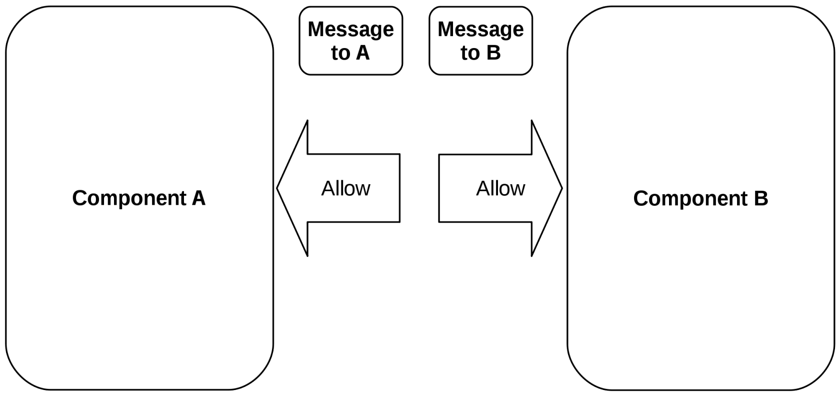


Figure 3.9: Coordination Flow Control between Isolated Service Components

predetermined size. Such a rotation under load may generate a number of log objects on a daily basis that all need to be processed by the Log Analyzer component. Therefore, an exchange of coordination messages between two components may allow them to convey information about a specific set of objects that are required for processing to generate a daily analytical report on HTTP traffic. Such a coordination flow is depicted in Figure 3.9. Coordination across mutually isolated environments is problematic. However, if components belong to a single communicative class, it enables the exchange of coordination messages without reliance on usual UNIX IPC mechanisms that may be unavailable under security constrained conditions [8].

Collaboration – components acting as a single service may need to access mutual data or runtime file objects to collaborate and perform joint or codependent measurements or calculations as illustrated in the description of the web caching service. Empowering a component with the ability to obtain a replica of a data object that belongs to another component in the same communicative class makes such a collaboration possible.

As a part of our unified framework, communicative class supports a set of management operations on it [8]. The following high-level operations are supported by our communicative policy class model:

1. *create* a communicative policy class,
2. *add/remove* components to/from a communicative policy class,
3. *show/count* components in a communicative policy class,
4. *add/remove* explicit permission for a component to request a replica of a file system data object(s) to/from a communicative policy class, and
5. *enable/disable* component coordination with other component(s) in a communicative policy class.

A communicative policy class can be classified as a *coordinative class* or a *collaborative class* depending on whether it needs to coordinate or share data with other components. Coordinative class contains a set of coordination policies and a collaborative class contains a set of collaboration policies. Let $\mathcal{CR}$ and $\mathcal{CL}$ be the set of coordinative classes and collaborative classes respectively. Note that, the set of communicative classes comprise the set of coordinative class and the set of collaborative class, that is, $\mathcal{CM} = \mathcal{CR} \cup \mathcal{CL}$. Moreover, if a communicative class cm_i supports both coordination and collaboration, then $cm_i \in \mathcal{CR} \cap \mathcal{CL}$.

We now give the details of our access control policies in coordinative and collaborative policy classes. The access control rules for the coordinative policy class is defined by a partial function AF_{cr} that takes in two components as arguments and returns either 1 or 0 signifying whether the coordination is allowed or prohibited. $AF_{cr}(c_i, c_j) = 1$ signifies that components c_i and c_j are allowed to coordinate with each other otherwise such coordination is prohibited. Note that, this function is commutative. As per the rules of the coordinative policy class, the two components c_i and c_j can coordinate only if they belong to some coordinative policy class cr_p. This is denoted as $AF_{cr}(c_i, c_j) = 1$ only if $c_i, c_j \in cr_p$ where $cr_p \in \mathcal{CR}$.

Next, we describe the access control rules for the collaborative policies. The collaborative policies are more complex. The components must be part of the same collaborative class and also there must be an explicit permission that gives a component access to replicas' of specific objects in the collaborative class. Each private object o_r is owned by some component which is given by the function $own(o_r)$. $own(o_r) = c_m$ denotes that o_r is the private object of component c_m. Let $\mathcal{P}$ be the set of explicit permissions, denoted as $\mathcal{P} = \{p_1, p_2, \ldots, p_s\}$, that grants components access to private object's replica belonging to other components. Each permission $p_i \in \mathcal{P}$ is a triple of the form $< c_m, c_n, o_k >$ which denotes that component c_m has permission to read replica of object o_k that is owned by component c_n. We are now ready to describe the access control function AF_{cl}. This is a partial function that takes in two arguments, namely, a component and a data object and returns 1 or 0 signifying whether such access is allowed or prohibited. $AF_{cl}(c_k, o_n) = 1$ signifies that c_k is allowed to read a replica of o_n otherwise such read is prohibited. The component c_k can read object o_n, owned by component c_m, only if c_k and c_m belong to some collaborative policy class cl_q and there is a permission that allows c_k access to o_n that is owned by c_m. Formally, $AF_{cl}(c_k, o_n) = 1$ only if $< c_k, c_m, o_n > \in \mathcal{P}$ where $own(o_n) = c_m$, and $c_k, c_m, \in cl_q$ where $cl_q \in \mathcal{CL}$.

3.4.1 Supplemental Use Case

As noted, in the context of isolation within a single OS that hosts multiple services the various service components may not have access to a shared location for the purpose of communication and data sharing. We provide an additional example that will help to illustrate the introduced collaboration and coordination workflows.

The e-greetings service is composed of three service components each running in a separate isolated environment. The service sends personalized greeting e-cards to the e-mail addresses of the recipients. Components have only the minimum privileges needed to accomplish their tasks and are deployed under separate unprivileged UIDs. Due to isolation properties, those applications cannot write data objects to a shared storage area of the server OS such as /var directory to simplify their interaction.

First component is the HTTP web server that provides the web interface for the service users to fill out the greetings card form where a user can choose a specific card image and specify the recipient e-mail address. Four key attributes of the individual greetings order are written into a separate order text file – greetings message, sender's and recipient's e-mail addresses and the absolute URL to the location of the postcard image file. Second component is the e-mail assembling agent (mailman) that processes the order file to prepare the greetings e-mail message. Third component of the service is the actual Mail Transfer Agent (MTA) that is responsible for sending the individual e-card e-mail message to the greetings recipient. Thus, the mailman component requires controlled access to the order file prepared by the web server component to create the e-mail message and the mail sending component, in turn, requires controlled access to individual mail message object [8].

One possible deployment scheme for the described e-greetings service might add the layer of coordination logic to the interaction between the components. For example, if the web component of the service experiences excessive load from user requests and generates massive amounts of order files in a single directory, it can optionally send the coordination message to the mailman component to indicate that the request interval for the order files has to be decreased from the default 30 seconds to 5 seconds. Optionally, additional coordinative information may be incorporated in the message such as availability of alternative web server for load-balancing purposes. If the mailman component is capable of receiving and processing such coordination messages, it can, in turn, adjust its e-mail message preparation intervals. Consequently, it can coordinate with the mail sending component to indicate faster request times for the assembled e-mail message objects [8].

3.5 Overview of the Policies Formulation

The access control policies for the introduced framework are managed via the Domain Specific Policy Language (DSPL) within LPM (a part of the Parser Layer depicted in Figure 6.1) that allows the formulation of various component-oriented policies in a human, rather than machine-friendly form [3, 8]. The language allows to perform Create / Read / Update / Delete (CRUD) functionality on individual policies and create both capability as well as communicative policy classes in the Policies Store (PS). In this section, we demonstrate that meaningful component-oriented policies can be expressed completely in a form simple and concise enough to be administered at a reasonable cost. For implementation details of the Parser Layer we direct the interested reader to our GitHub repository for LPM [6].

3.5.1 Capabilities Classes Management

The following samples of policies illustrate how capabilities classes can be managed in practice via the CLI provided to the administrative personnel:

- `[ COUNT_CAPABILITIES_CLASSES ]` – show how many Capabilities Classes exist in the PS

- `[ SHOW_CAPABILITIES_CLASSES ]` – show individual Capabilities Classes in the PS
- `[ CREATE_CAPABILITIES_CLASS 1 application_service_with_ID_1_class ]` – create Capabilities Class for a specific application service
- `[ SHOW_CAPABILITIES_CLASS_CAPABILITIES 1 ]` – show all Linux Capabilities associated with a particular Capabilities Class
- `[ ADD_CAPABILITIES_CLASS_CAPABILITY 1 CAP_DAC_OVERRIDE ]` – add CAP_DAC_OVERRIDE Linux Capability to a particular Capabilities Class
- `[ REMOVE_CAPABILITIES_CLASS_CAPABILITY 1 CAP_CHOWN ]` – remove CAP_CHOWN Linux Capability from a particular Capabilities Class
- `[ SHOW_CAPABILITIES ]` – show all available Linux Capabilities
- `[ COUNT_CAPABILITIES_CLASS_COMPONENTS 1 ]` – show how many service components are associated with a particular Capabilities Class
- `[ SHOW_CAPABILITIES_CLASS_COMPONENTS 1 ]` – show all service components associated with a particular Capabilities Class
- `[ MOVE_COMPONENT_TO_CAPABILITIES_CLASS /opt/containers/service-100/bin/log-analyzer 1 ]` – move a service component to a particular Capabilities Class

3.5.2 Communicative Classes Management

The following samples of policies illustrate how communicative classes can be managed in practice via the CLI provided to the administrative personnel:

- `[ COUNT_COMMUNICATIVE_CLASSES ]` – show how many Communicative Classes exist in the PS

- `[ SHOW_COMMUNICATIVE_CLASSES ]` – show individual Communicative Classes in the PS
- `[ CREATE_COMMUNICATIVE_CLASS 1 web_caching_service_class ]` – create Communicative Class for a specific application service
- `[ SHOW_COMMUNICATIVE_CLASS_COMPONENTS 1 ]` – show all service components associated with a particular Communicative Class
- `[ COUNT_COMMUNICATIVE_CLASS_COMPONENTS 1 ]` – show how many service components are associated with a particular Communicative Class
- `[ MOVE_COMPONENT_TO_COMMUNICATIVE_CLASS /opt/containers/service-100/bin/log-analyzer 1 ]` – move a service component to a particular Communicative Class
- `[ SHOW_COMMUNICATIVE_CLASS_COLLABORATION_POLICIES 1 ]` – show all collaboration records associated with a particular Communicative Class
- `[ SHOW_COMMUNICATIVE_CLASS_COORDINATION_POLICIES 1 ]` – show all coordination records associated with a particular Communicative Class
- `[ ADD_COMMUNICATIVE_CLASS_COLLABORATION_POLICY 1 component_pathID object_path ]` – add a collaborative policy to a particular Communicative Class
- `[ ADD_COMMUNICATIVE_CLASS_COORDINATION_POLICY 1 component_pathID_1 component_pathID_2 ]` – add a coordinative policy to a particular Communicative Class
- `[ REMOVE_COMMUNICATIVE_CLASS_COLLABORATION_POLICY 1 component_pathID object_path ]` – remove a collaborative policy from a particular Communicative Class
- `[ REMOVE_COMMUNICATIVE_CLASS_COORDINATION_POLICY 1 component_pathID_1 component_pathID_2 ]` – remove a coordinative policy from a particular Communicative Class

Chapter 4

Communication Architecture

As discussed in Chapter 3, communication between various service components is regulated via the concept of communicative policy classes. Such an access control abstraction has to be properly supported at the level of LPM reference monitor that needs to possess the necessary communication primitives to enforce such a regulation. We now provide details on our enforcement architecture for communicative class model that is one of the main contributions of this work.

4.1 Tuple Space Paradigm

In order to address the problems of interprocess communication across isolated environments we proposed an alternative approach that can be classified as a special case of *generative communication* paradigm introduced by Linda programming model [29]. In this approach, processes communicate *indirectly* by placing tuples in a *tuple space*, from which other processes can read or remove them. Tuples do not have an address but rather are accessible by matching on content therefore being a type of content-addressable associative memory [48]. Such a tuple space via which processes can communicate is depicted in Figure 4.1.

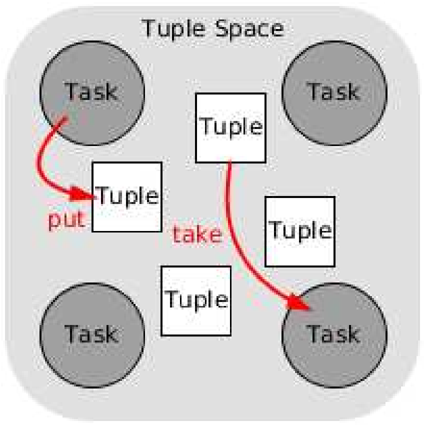


Figure 4.1: Tuple Space

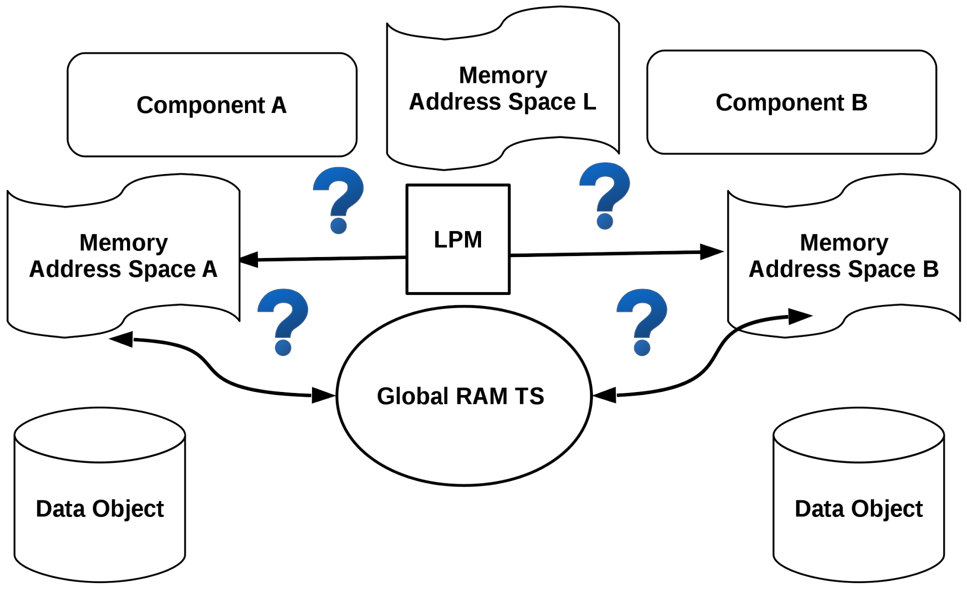


Figure 4.2: Tuple Space Limitations - 1

This programming model allows decoupled interaction between processes separated in time and space: communicating processes need not know each other's identity, nor have a dedicated connection established between them [62]. In comparison to general-purpose message-passing that provides a rather low-level programming abstraction for building distributed systems and enabling inter-component interaction, Linda, instead, provides a simple coordination model with higher level of abstraction that makes it very intuitive and easy to use [16].

4.1.1 Paradigm Limitations

The lack of any protection mechanism in the basic model [48, 62] makes the single global shared tuple space unsuitable for interaction and coordination among untrusted components. There is also the danger of possible tuple collisions – as the number of tuples that belong to a large set of divergent components in a tuple space increases, there is an increasing chance of accidental matching of a tuple that was requested by another component. Moreover, the traditional in-memory implementation of tuple space, oriented at language-level interaction [66], makes it unsuitable in

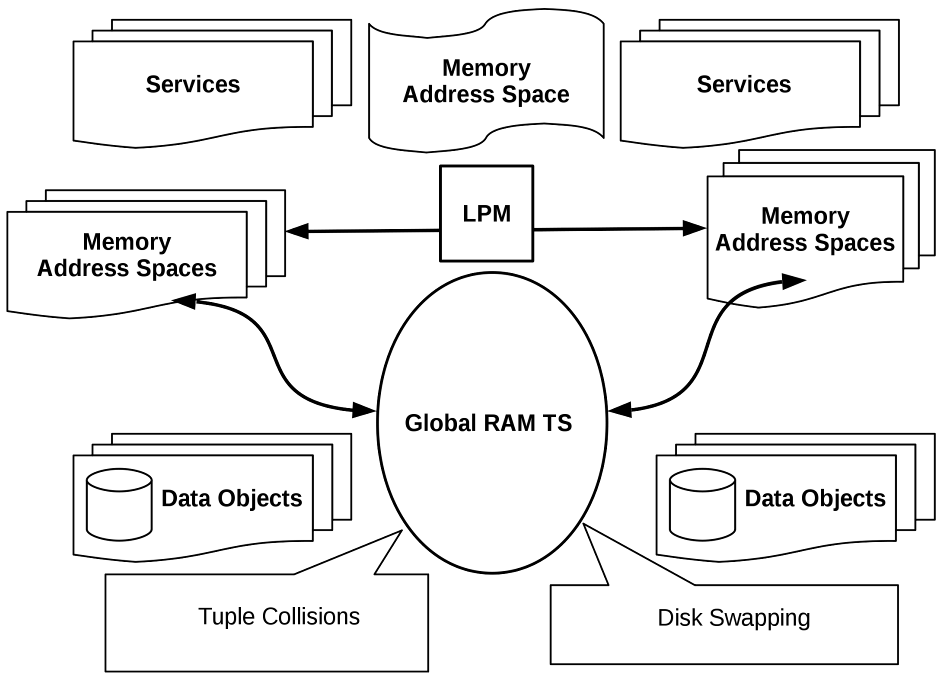


Figure 4.3: Tuple Space Limitations - 2

our current work due to a wide array of possible security attacks and memory utilization overheads. Solutions based on main memory could be subjected to heavy disk swapping with simultaneous transfers of large data objects. That essentially eliminates the advantages of using purely memory resident tuple spaces with hardware that has limited RAM capacity [7]. Therefore, we adapt the tuple space model that will satisfy our requirements for secure and reliable communication between service components within a single communicative policy class [8]. Note that, in this adaptation the content-based nature of retrieval from a tuple space will necessitate content-based access control approaches [48].

Another problem identified with the RAM-based tuple spaces is that it is suitable mainly for a single application with multiple threads that share the same memory address space or applications that rely on some form of shared memory support [11]. In such a simplified deployment scenario, a global tuple space is easily accessible by consumer and producer threads within a single application. However, in the context of our current work we deal with separate service components that do not share the same address space in memory which makes such a solution unsuitable [17].

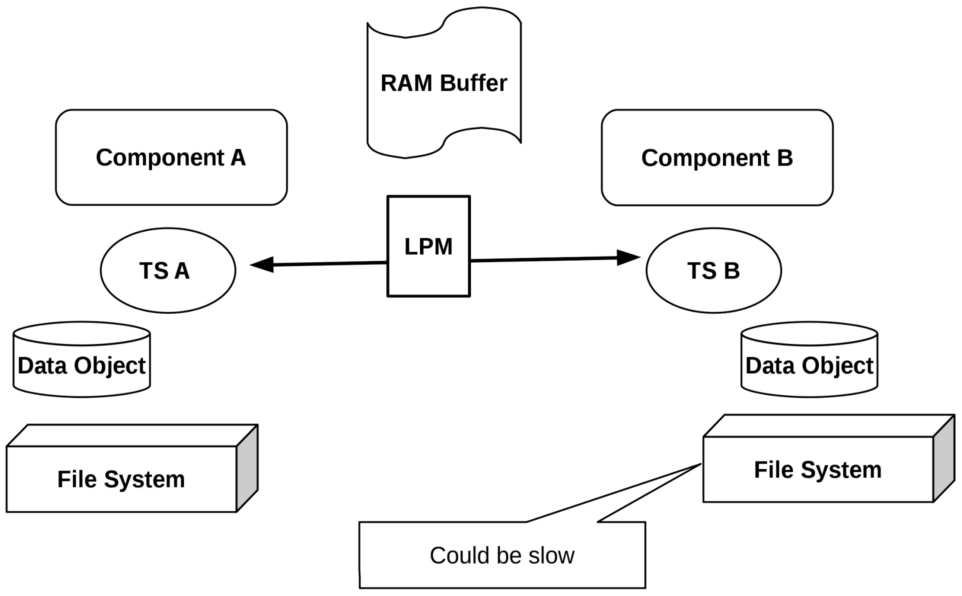


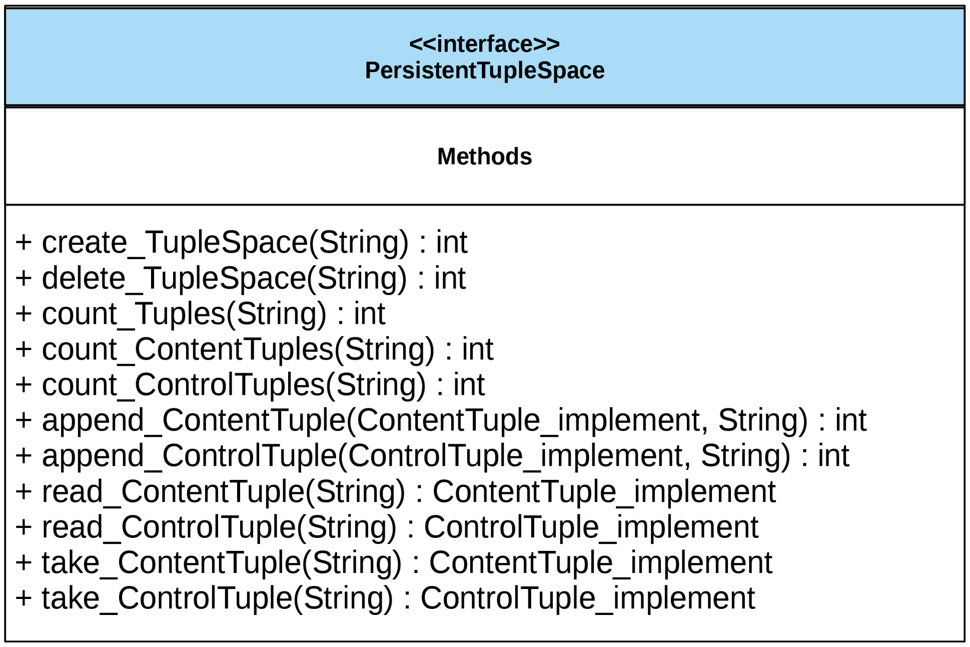
Figure 4.4: Tuple Space on a File System

For instance, two isolated service components written in Java cannot access mutual tuple spaces because each component is deployed in a separate Java Virtual Machine (JVM) instance [65]. The discussed limitations are depicted in Figures 4.2 and 4.3.

4.1.2 Paradigm Adaptation

We propose a tuple space *calculus* that is compliant with the originally introduced base model [29] but is applied on dedicated tuple spaces of individual service components instead of a global space. Our *tuple space calculus* comprises the following operations:

1. *create tuple space* operation,
2. *delete tuple space* operation – Deletes tuple space only if it is empty,
3. *read* operation – Returns the value of individual tuple without affecting the contents of a tuple space,
4. *append* operation – Adds a tuple without affecting existing tuples in a tuple space, and



```
<<interface>>
PersistentTupleSpace

Methods

+ create_TupleSpace(String) : int
+ delete_TupleSpace(String) : int
+ count_Tuples(String) : int
+ count_ContentTuples(String) : int
+ count_ControlTuples(String) : int
+ append_ContentTuple(ContentTuple_implement, String) : int
+ append_ControlTuple(ControlTuple_implement, String) : int
+ read_ContentTuple(String) : ContentTuple_implement
+ read_ControlTuple(String) : ControlTuple_implement
+ take_ContentTuple(String) : ContentTuple_implement
+ take_ControlTuple(String) : ControlTuple_implement
```

Figure 4.5: Interface of a Persistent Tuple Space

5. *take* operation – Returns a tuple while removing it from a tuple space.

We adhere to the *immutability* property – tuples are immutable and components can either append or remove tuples in a tuple space without changing contents of individual tuples.

A component is allowed to perform all the described operations in its tuple space while LPM is restricted to read and append operations only. Note, that the take operation is the only manner in which tuples get deleted from a tuple space because the delete tuple space operation is allowed only on an empty tuple space [8].

Tuple space is implemented as an abstraction in the form of a file system directory with its calculus performed via Tuple Space Library (TSL) employed by the service components and the LPM reference monitor through its Tuple Space Controller (TSC) – a module of LPM which is allowed a limited access to a component's tuple space. The adaptation is shown in Figure 4.4. Therefore, this part of the proposed unified framework is not transparent and the components may need to be modified in order to utilize the tuple space communication. However, in certain cases that may not be necessary. For instance, if components require only limited collaboration, such

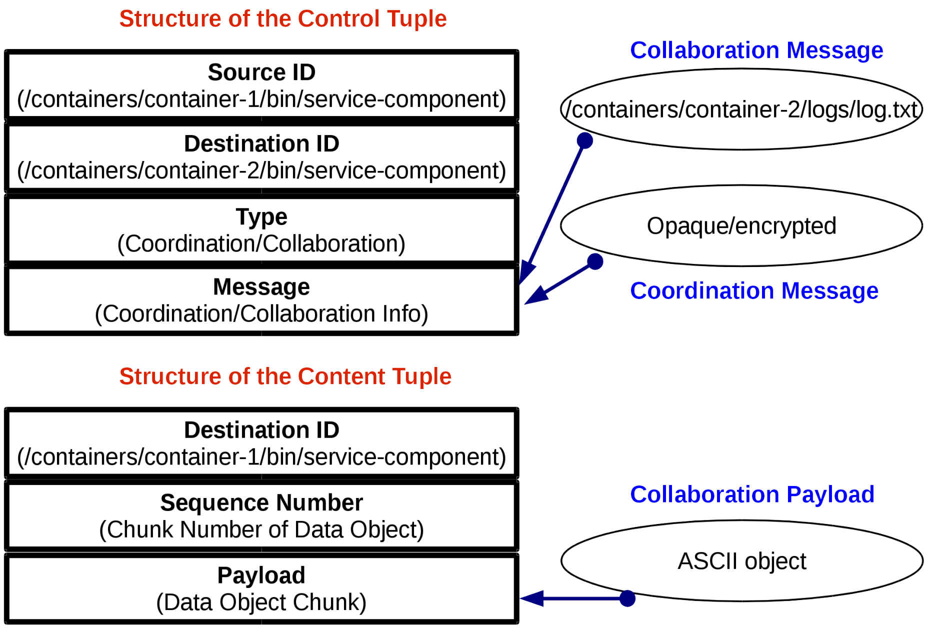

Structure of the Control Tuple
Source ID
(/containers/container-1/bin/service-component)
Destination ID
(/containers/container-2/bin/service-component)
Type
(Coordination/Collaboration)
Message
(Coordination/Collaboration Info)
Collaboration Message
/containers/container-2/logs/log.txt
Opaque/encrypted
Coordination Message
Structure of the Content Tuple
Destination ID
(/containers/container-1/bin/service-component)
Sequence Number
(Chunk Number of Data Object)
Payload
(Data Object Chunk)
Collaboration Payload
ASCII object

Figure 4.6: Tuples Structure

as periodic requests for replicas of data objects (the case for daily logs), a separate data requester component that employs TSL can handle such a task without the need to modify the existing component such as a log analyzer [8]. The discussed operations on such a persistent tuple space on a file system are shown in Figure 4.5.

The LPM plays a mediating role in the communication between service components. The communication takes place through two types of tuples: *control tuples* and *content tuples*. Control tuples can carry messages for coordination or requests for sharing. Content tuples are the mechanism by which data gets shared across service components. The LPM periodically checks for control tuples in the tuple spaces for components registered in its database. We have two different types of communication between a pair of service components. The first case is where the two components do not share any data but must communicate with each other in order to coordinate activities or computation. The second case is where a component shares its data with another one. Note, that in our calculus, at most one control tuple and one content tuple could be appended into a tuple space at any given time [7, 8].

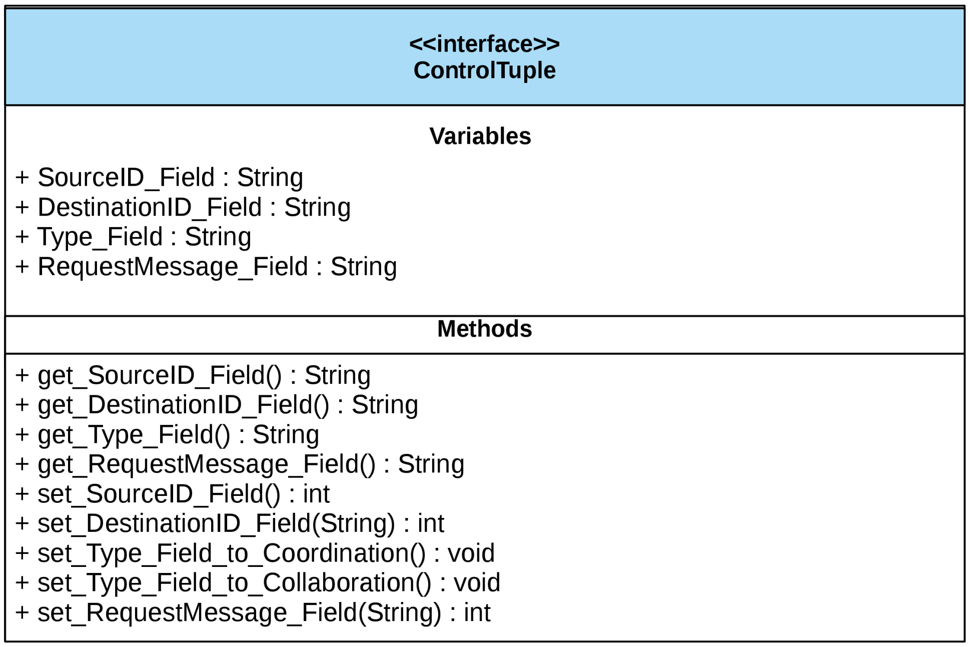

<<interface>>
ControlTuple
Variables
+ SourceID_Field : String
+ DestinationID_Field : String
+ Type_Field : String
+ RequestMessage_Field : String
Methods
+ get_SourceID_Field() : String
+ get_DestinationID_Field() : String
+ get_Type_Field() : String
+ get_RequestMessage_Field() : String
+ set_SourceID_Field() : int
+ set_DestinationID_Field(String) : int
+ set_Type_Field_to_Coordination() : void
+ set_Type_Field_to_Collaboration() : void
+ set_RequestMessage_Field(String) : int

Figure 4.7: Interface of a Control Tuple

The structure of the tuples is shown in Figure 4.6. Control tuples are placed by a service component into its tuple space for the purpose of coordination or for requesting data from other components.

A control tuple has the following fields:

1. *Source ID* – This indicates an absolute path of the service component that acts as the identifier of the communication initiator.

2. *Destination ID* – This indicates an absolute path of the component that acts as the identifier of the communication recipient.

3. *Type* – This indicates whether it is a collaborative or coordinative communication.

4. *Message* – This contains the collaborative/coordinative information. For collaboration it is the request for an absolute path of data object. Coordination message may be opaque as other entities may be oblivious of this inter-component communication. It may even be encrypted to ensure the security and privacy of inter-component coordination efforts. XML or JSON

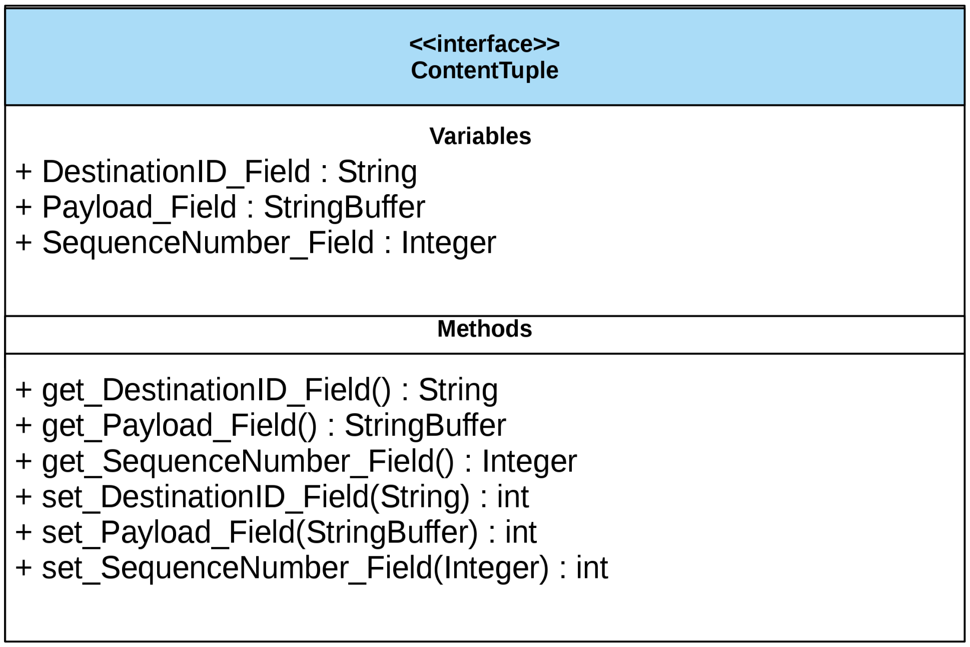

<<interface>>
ContentTuple
Variables
+ DestinationID_Field : String
+ Payload_Field : StringBuffer
+ SequenceNumber_Field : Integer
Methods
+ get_DestinationID_Field() : String
+ get_Payload_Field() : StringBuffer
+ get_SequenceNumber_Field() : Integer
+ set_DestinationID_Field(String) : int
+ set_Payload_Field(StringBuffer) : int
+ set_SequenceNumber_Field(Integer) : int

Figure 4.8: Interface of a Content Tuple

are possible formats that can be used for the representation of coordination messages. LPM merely shuttles the coordination tuples between respective components' spaces and is not aware of their semantics [8].

The interface for basic operations on manipulating various fields of a control tuple are depicted in Figure 4.7.

Content tuples are used for sharing data objects across components and they have the following fields:

1. *Destination ID* – This indicates the ID of recipient component that is an absolute path of a component.

2. *Sequence Number* – This indicates the sequence number of a data object chunk that is transported. ASCII objects in the form of chunks are the primary target of inter-component collaboration.

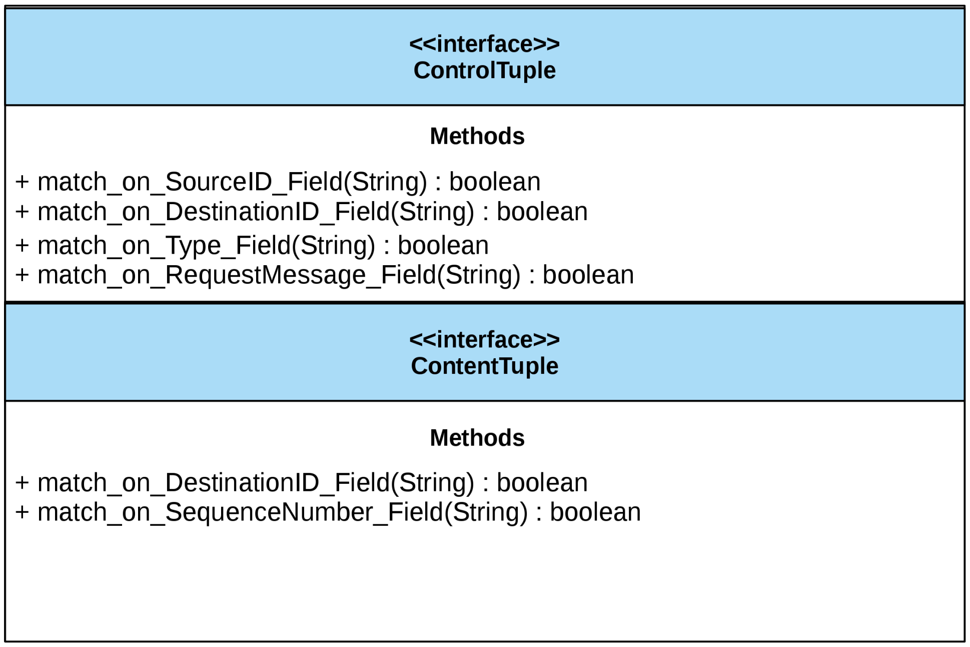

<<interface>>
ControlTuple
Methods
+ match_on_SourceID_Field(String) : boolean
+ match_on_DestinationID_Field(String) : boolean
+ match_on_Type_Field(String) : boolean
+ match_on_RequestMessage_Field(String) : boolean
<<interface>>
ContentTuple
Methods
+ match_on_DestinationID_Field(String) : boolean
+ match_on_SequenceNumber_Field(String) : boolean

Figure 4.9: Template Matching Operations on Tuples

3. *Payload* – This contains the chunk of a data object. Content tuples are placed by the LPM reference monitor into corresponding tuple space of the requesting component that needs to receive content. Note that content tuples are designed for collaboration only. Coordination is performed exclusively through control tuples [8].

   The interface for basic operations on manipulating various fields of a content tuple are depicted in Figure 4.8.

Aside from basic set and get operations, individual tuples have support for template matching on main fields. Such operations are widely used in practice during access control checks prior to enforcement. They may also provide rudimentary support for higher-level operations that could be incorporated in the future based on service requirements [5]. The corresponding methods have boolean output to easily detect whether the content of individual field matches the stated template. Such matching operations are depicted in Figure 4.9.

Containerized service components are often not aware of whether they are deployed in an isolated runtime environment, such as a chrooted jail or not. Therefore, tuple fields, such as

Source/Destination IDs and object paths that technically require the absolute path to the object on the file system can be substituted with the isolated environment ID, such as a container ID. This permits the service deployment with individual components that are only aware of immediate containerized path locations or corresponding components' service identifiers. For instance, the containerized identifier, such as */100/opt/bin/service-component-2* can be mapped to a system-wide path of */opt/containers/container-100/opt/bin/service-component-2* by the LPM reference monitor with a proper support for such a composite service mapping [7].

Chapter 5

Tuple Space Transactions

In this chapter we provide the details on sample transactional flow involved in tuple space operations, necessary to carry out collaborative and coordinative types of communication between isolated service components. Such a flow, termed as *Tuple Space Transactions* (TST) could be categorized as a form of regulated OS-level interaction [14] between such components. Since loosely coupled processes cannot communicate directly due to isolation properties, the flow is conducted indirectly via the Tuple Space Controller (TSC) [7].

5.1 Coordinative Transaction

Coordinative communication between two components is depicted in Figure 5.1. Intrinsically, coordination is bidirectional, since both endpoints need to obtain coordinative messages. Both components need to create the corresponding tuple spaces in the isolated runtime environments. In the first phase, Component 1 delivers a message to Component 2.

- **[Step 1:]** Component 1 appends a control tuple (see the structure of tuples in Figure 4.6) to its tuple space *TS 1*. This control tuple (denoted as message A) has to be subsequently delivered to Component 2;
- **[Step 2:]** TSC reads the control tuple from *TS 1*;
- **[Step 3:]** Component 1 retracts the control tuple via the take operation;
- **[Step 4:]** TSC appends the control tuple into tuple space *TS 2* of Component 2;
- **[Step 5:]** Component 2 takes the appended control tuple (message A from Component 1) from its tuple space *TS 2*.

In the next phase of coordinative communication, Component 2 has to deliver its coordination message to Component 1. Such a message could contain independently new coordinative infor-

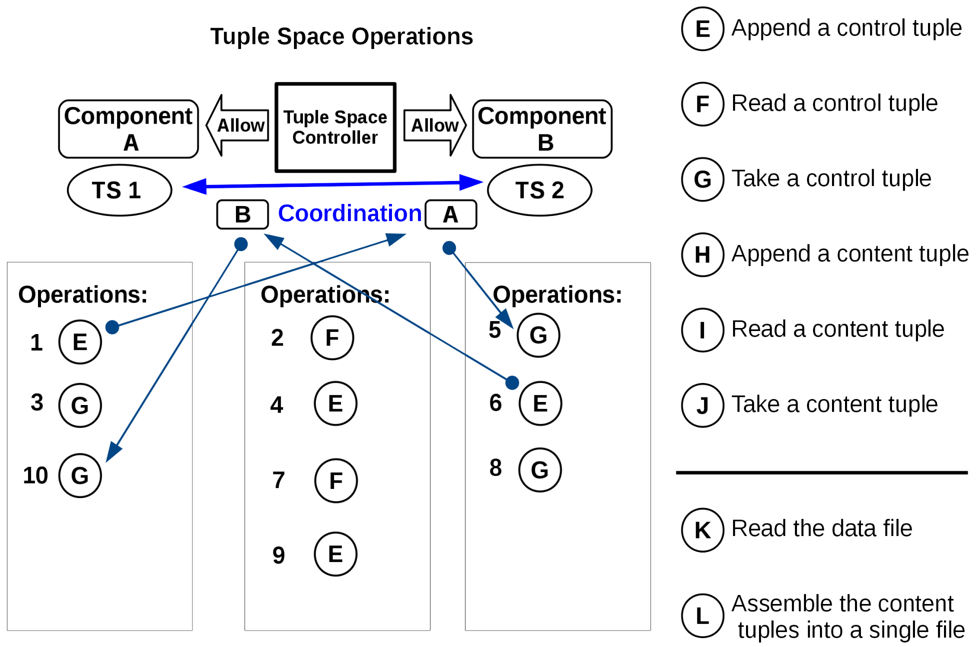


Figure 5.1: Coordination through Tuple Spaces

mation or serve as the acknowledgement for the control tuple that has just been received. Such a decision is service-specific. However, we require that coordinative transactional flow is terminated through such a confirmative control tuple from Component 2. The steps in the second phase are described next.

- **[Step 6:]** Component 2 appends a control tuple to its tuple space *TS 2*. This control tuple (denoted as message B) has to be subsequently delivered to Component 1;
- **[Step 7:]** TSC reads the control tuple from *TS 2*;
- **[Step 8:]** Component 2 retracts the control tuple via the take operation;
- **[Step 9:]** TSC appends the control tuple into tuple space *TS 1* of Component 1;
- **[Step 10:]** Component 1 takes the appended control tuple (message B from Component 2) from its tuple space *TS 1*. This step completes the coordinative transaction.

Note that the coordination messages could be of any type. Therefore, our communication architecture allows full transparency in inter-component exchange and does not require proprietary formats. Most common formats that could be incorporated into the message field of a control tuple are XML, JSON or ASCII strings. Such a choice is service-dependent. Moreover, the service components could utilize the serialization libraries such as XStream [71], to represent class objects in the form of XML messages. In this case, isolated components that use our TSL library can perform complete object-based transport within a single service solely through provided tuple space communication [7].

5.2 Collaborative Transaction

Collaborative communication is depicted in Figure 5.2. Intrinsically, collaboration is unidirectional, since the workflow is only directed from a single requester to TSC and back in the form of content tuples [7]. In contrast to a control tuple, a content tuple only has a Destination ID field, as depicted in Figure 4.6. However, at the level of service logic, collaboration flow could conceptually be bidirectional. Both endpoints could obtain replicas of mutual data objects through TSC, if such a replication is explicitly permitted in the policies store of a reference monitor. Such a scenario of symmetric collaboration is depicted in Figure 5.2. The steps of collaborative transaction, on the left, are shown below.

- **[Step 1:]** Component 1 appends a control tuple to its tuple space *TS 1* with indication of request for data object that is owned by Component 2;
- **[Step 2:]** TSC reads the control tuple from *TS 1*;
- **[Step 3:]** TSC reads the requested data object on the file system. Note that this step is not a part of the actual transactional flow, but represents the internal operations of TSL;
- **[Step 4:]** TSC appends the replica of a data object, fragmented in three content tuples, into tuple space *TS 1*, one tuple at a time. Note that TSC can append the next content tuple only after the current one is taken from a tuple space. The step shows four actual tuples – TSC

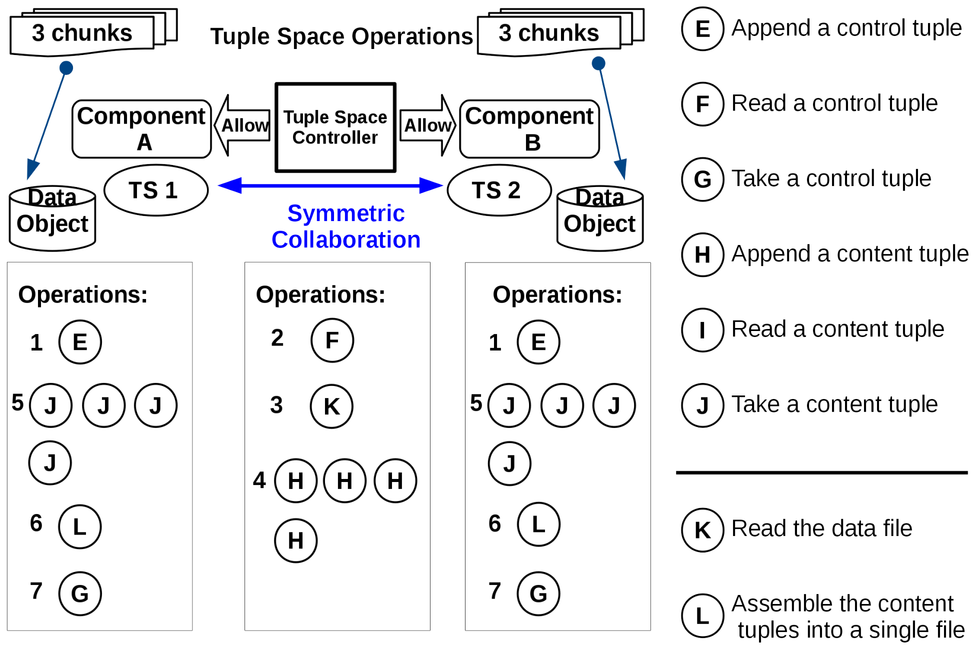
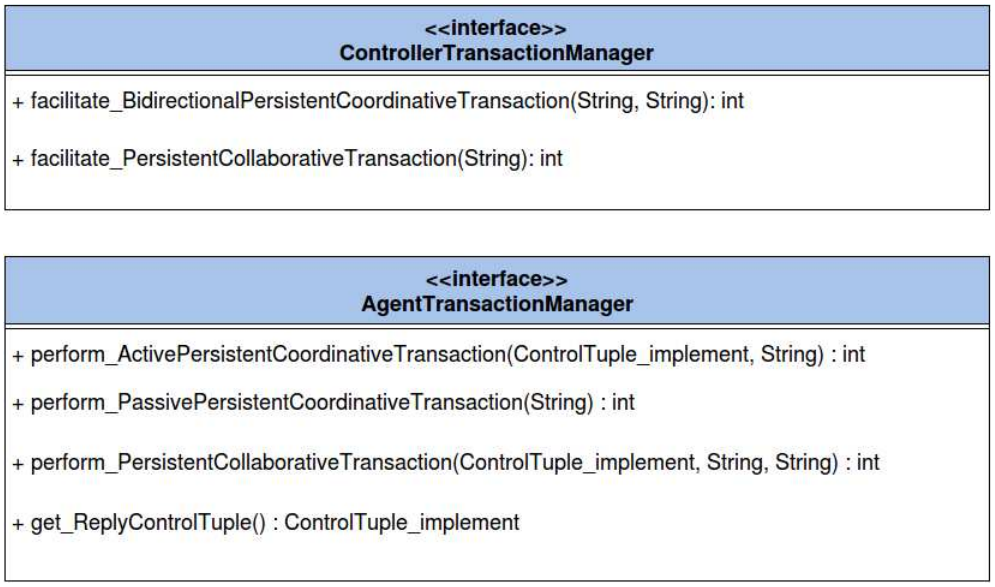


Figure 5.2: Collaboration through Tuple Spaces

has to append a special End of Flow (EOF) content tuple to indicate the end of data flow. Such a tuple has the Payload field set to empty string and Sequence Number field set to -1 to indicate the EOF condition;

- **[Step 5:]** Component 1 takes appended content tuples, one tuple at a time;
- **[Step 6:]** Component 1 assembles the appended content tuples into a replica of the requested data object. Note that this step is not a part of the actual transactional flow, but represents the internal operations of TSL;
- **[Step 7:]** Component 1 takes a control tuple from its tuple space *TS 1*. This step completes the collaborative transaction.

The flow of second collaborative transaction, on the right, is identical. The communication starts with the creation of a tuple space and ends with its deletion after the transactional flow completes.



```
<<interface>>
ControllerTransactionManager

+ facilitate_BidirectionalPersistentCoordinativeTransaction(String, String): int

+ facilitate_PersistentCollaborativeTransaction(String): int
```

```
<<interface>>
AgentTransactionManager

+ perform_ActivePersistentCoordinativeTransaction(ControlTuple_implement, String) : int
+ perform_PassivePersistentCoordinativeTransaction(String) : int

+ perform_PersistentCollaborativeTransaction(ControlTuple_implement, String, String) : int

+ get_ReplyControlTuple() : ControlTuple_implement
```

Figure 5.3: API of Tuple Space Transactions

5.3 Transactional API

The complexity for both types of communication is hidden from the components. TSL provides public Application Programming Interface (API) methods without exposing internal operations of tuple space calculus [6].

The main API methods for tuple space transactions are depicted in Figure 5.3. TSC executes the implementation of the *ControllerTransactionManager* class while the service component executes the implementation of the *AgentTransactionManager* class within the TSL library. *Controller-TransactionManager* implementation has the following public methods:

- *facilitate_BidirectionalPersistentCoordinativeTransaction()* – performs the exchange of control tuples between corresponding tuple spaces of service components. The implementation of this method uses the private *facilitate_UnidirectionalPersistentCoordinativeTransaction()* method to append control tuples to individual tuple spaces involved in the coordination.

- *facilitate_PersistentCollaborativeTransaction()* – performs the replication of a data object requested in the collaborative request issued by the component.

AgentTransactionManager implementation has the following public methods:

- *perform_ActivePersistentCoordinativeTransaction()* – initiates a start of coordinative transaction by appending the initial control tuple in its own tuple space.
- *perform_PassivePersistentCoordinativeTransaction()* – initiates the ending of coordinative transaction by waiting for a control tuple from the counterpart component.
- *perform_PersistentCollaborativeTransaction()* – initiates and completes the collaborative transaction by assembling the replica at component's end.
- *get_ReplyControlTuple()* – obtains the control tuple that has been appended by the TSC in its tuple space from the counterpart component.

Therefore, a single coordinative transaction (control flow) depicted in Figure 5.1 that utilizes a set of operations of tuple space calculus is actualized through the invocation of the *facilitate_BidirectionalPersistentCoordinativeTransaction()* method within ControllerTransactionManager instance in TSC. It is coupled with the invocation of *perform_ActivePersistentCoordinativeTransaction()* method within AgentTransactionManager implementation on a side of the component that initiates coordination. It is then followed by the invocation of *perform_Passive- PersistentCoordinative-Transaction()* method within AgentTransactionManager implementation on a side of the component that receives initial coordination message.

Consequently, a single collaborative transaction (data flow) depicted in Figure 5.2 that utilizes a set of operations of tuple space calculus is implemented through the invocation of the *facilitate_PersistentCollaborativeTransaction()* method within *ControllerTransactionManager* instance in TSC. It is coupled with the invocation of *perform_PersistentCollaborativeTransaction()* method within *AgentTransactionManager* implementation on a side of the component that requests a replica of a data object. Note, that the actual replication through content tuples is implemented within TSL Utilities package. *ControllerTransactionManager* invokes the *fragment_ObjectReplica()* method of this package in its execution of *facilitate_PersistentCollaborativeTransaction()* method. Consequently, *AgentTransactionManager* invokes *assemble_ObjectReplica()* method of the same package in its execution of *perform_PersistentCollaborativeTransaction()* method.

5.4 Formal Properties of Tuple Space Calculus

In this section we elaborate on possible executions of introduced *tuple space calculus* that forms the basis of Tuple Space Transactions (TST). In Chapter 4 we have already briefly defined a set of such operations. Let us now formally define all of them individually in the form of *calculus functions* that take a single TS parameter as input:

1. *Cr_TS(TS)* – Create Tuple Space (TS) operation,
2. *D_TS(TS)* – Delete TS operation
3. *R_crt(TS)* – Read operation on a control tuple
4. *R_cnt(TS)* – Read operation on a content tuple
5. *A_crt(TS)* – Append operation on a control tuple
6. *A_cnt(TS)* – Append operation on a content tuple
7. *T_crt(TS)* – Take operation on a control tuple
8. *T_cnt(TS)* – Take operation on a content tuple
9. *C_crt(TS)* – Count operation that outputs a number of control tuples in a given TS
10. *C_cnt(TS)* – Count operation that outputs a number of content tuples in a given TS
11. *C(TS)* – Count operation that outputs a number of both control and content tuples in a given TS

Note, that in our calculus, at most one control tuple and one content tuple could be appended into a tuple space at any given time [7, 8]. Therefore, $C\_crt(TS) \leq 1$, $C\_cnt(TS) \leq 1$ and $C(TS) \leq 2$ for any given TS in a given model. Let us now formally define a tuple space within a given file system structure that may be accessed by TSC and service component in the context of our model.

Definition 5.4.1 [Tuple Space:] *A* tuple space *is one owned by the component whose members are tuples that can be operated upon via a set of predefined calculus functions such that* $C\_crt(TS) \leq 1$*,* $C\_cnt(TS) \leq 1$ *and* $C(TS) \leq 2$

Let us now define a notion of transactions.

Definition 5.4.2 [Collaborative Transaction:] *A* collaborative transaction *is one conducted between TSC and service component through a sequence of calculus functions that delivers a copy of an object to a service component through its tuple space.*

Definition 5.4.3 [Coordinative Transaction:] *A* coordinative transaction *is one conducted between a pair of service components via TSC through a sequence of calculus functions that exchanges control messages between such components through their respective tuple spaces.*

As previously stated, TSC is restricted to a subset of *Read* and *Append* operations only. Note, that the *Read* operation and its corresponding calculus functions are equivalent to a copy operation. *Read* and *Count* operations are the only operations on TS that do not conflict with the rest of calculus functions. Therefore, these are the only two operations that can be *interleaved* within a given TST without a violation of the consistency of its execution. In fact, the actual implementation of TSL library extensively uses *Count* to validate the stated properties of TS before the execution of every calculus function within a given transactional sequence.

Note, that only a single TST may be executed within a given TS – every transaction starts with a creation of a TS and terminates with its removal. Such a *transactional singularity* is ensured by the single control tuple property per given TS. Therefore, the interleaving of multiple transactions within a single TS is precluded in our variant of a tuple space abstraction.

We can now define a single TST in the form of a serial *sequence* S where each operation is serially followed by the next one within such a transaction. Note, that operations executed by TSC are marked with $_c$ while operations by service component are given with $_a$ notation. A single collaborative transaction for a data object that can be replicated in three content tuples, $C\_cnt(TS) = 3$, can now be defined with the following serial sequence:

Example 5.1 Collaborative Sequence Example

$S1 = \{Cr\_TS(TS)_a; A\_crt(TS)_a; R\_crt(TS)_c; A\_cnt(TS)_c; T\_cnt(TS)_a; A\_cnt(TS)_c; T\_cnt(TS)_a; A\_cnt(TS)_c; T\_cnt(TS)_a; A\_cnt(TS)_c; T\_cnt(TS)_a; T\_crt(TS)_a; D\_TS(TS)_a; \}$

Note, that a sequence of four $A\_cnt(TS)$ and $T\_cnt(TS)$ operations is required to complete the transaction. That is because TSC appends additional EOF content tuple to indicate the completion of replication. The execution of sequence $S1$ is equivalent to the flow depicted in Figure 5.2. Note, that a collaborative sequence that involves a smaller/larger data object will only differ in the number of $A\_cnt(TS)$ and $T\_cnt(TS)$ operations to complete its execution.

A single coordinative transaction can now be defined with the following set of serial sequences between a pair of coordinating components:

Example 5.2 Coordinative Sequence Set Example

$S2 = \{Cr\_TS(TS1)_a1; A\_crt(TS1)_a1; R\_crt(TS1)_c; T\_crt(TS1)_a1; A\_crt(TS2)_c; T\_crt(TS2)_a2; \}$

$S3 = \{Cr\_TS(TS2)_a2; A\_crt(TS2)_a2; R\_crt(TS2)_c; T\_crt(TS2)_a2; A\_crt(TS1)_c; T\_crt(TS1)_a1; \}$

$S4 = \{D\_TS(TS1)_a1; D\_TS(TS2)_a2; \}$

Therefore, a complete bidirectional coordinative transaction is $\{S2; S3; S4; \}$. Note, that for clarity purposes, we now indicate the corresponding TS parameter for every involved operation. The execution of a set of sequences $\{S2; S3; S4; \}$ is equivalent to the flow depicted in Figure 5.1. Note, that both sequences $S2$ and $S3$ are equivalent and differ only in the TS input parameter for respective calculus functions. We can observe that in each of the sequences, TSC performs a copy of a tuple via the *Read* operation without the violation of tuple's immutability property.

At the same time, $S4$ can only be executed *after* both $S2$ and $S3$ complete. Therefore, a *temporal* execution dependency for $S4$ has to be observed for successful completion of transactional flow. That is because *Delete* operations on a given set of tuple spaces cannot be interleaved within earlier sequences due to bidirectional property of a coordinative transaction that requires the delivery of control tuples to a pair of service components. Specifically, such an interleaving will break the successful execution because the confirmatory control tuple will not be appended to a given TS due to its premature removal.

Chapter 6

System and Policies Storage Architecture

This chapter provides information on underlying system architecture of our component-oriented access control framework.

6.1 System Architecture

LPM acts as a centralized enforcement point and reference monitor (RM) [8, 38] for the services deployed on a single OS server instance. The unified framework uses the embedded SQLite [61] database library to store and manage policy classes abstractions and their policy records. The usage of embedded database facility eliminates the dependency on a separate database server that is prone to potential availability downtimes and security breaches. The LPM implemented in Java Standard Edition (SE) is deployed under unprivileged UID with elevated privileges using Linux capabilities within the same OS outside the containerized environments such as chrooted jails as a form of OS-level container. Figure 6.1 illustrates the components of the LPM. These are described below.

- **[User Interface Layer:]** This layer provides operator with Command-Line Interface (CLI) to issue commands to manage the framework.
- **[Parser Layer:]** This layer parses and validates the user input from the CLI shell and then forwards the parsed input to the underlying layers for execution. Its main functionality lies in internal DSPL policy language that is used for representing the access control policies in user-friendly form.
- **[Enforcer Layer:]** This layer enforces the capabilities on the given application using Linux LibCap [44] library and grants/denies access to OS resources depending on the capabilities class associated with the component. The layer also integrates a TSC [48] that is responsible for tuple space operations for the enforcement of collaboration and coordination of service

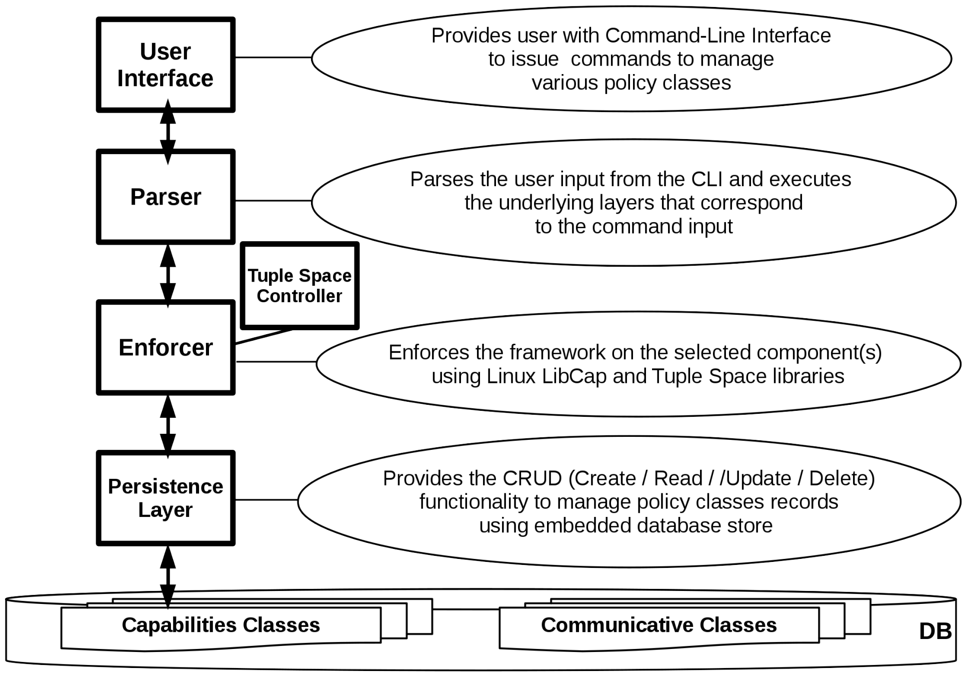

User Interface
Provides user with Command-Line Interface to issue commands to manage various policy classes
Parser
Parses the user input from the CLI and executes the underlying layers that correspond to the command input
Tuple Space Controller
Enforcer
Enforces the framework on the selected component(s) using Linux LibCap and Tuple Space libraries
Persistence Layer
Provides the CRUD (Create / Read / /Update / Delete) functionality to manage policy classes records using embedded database store
Capabilities Classes
Communicative Classes
DB

Figure 6.1: Architecture of Lightweight Policy Machine

components in a single communicative class. Note, that due to functional complexity of TSC, we provide only its reference implementation that uses our TSL library.

- **[Persistence Layer:]** This layer provides the Create / Read / Update / Delete (CRUD) functionality to manage access control policy records using embedded database facilities. This layer is discussed in detail in Section 6.2.

System implementation features tens of classes and packages associated with corresponding layers, with overall volume of current production code exceeding 12 000 lines, not counting the unit tests per individual class [6]. Specifically, the reference monitor is implemented using the modular architecture where individual layers are treated as a composition of multiple system components. Such an architecture facilitates transparent testing of functionality for individual components with a fine degree of granularity.

We believe that the provision of detailed implementation details for all the packages, classes and their class variables and methods associated with corresponding functionality layers of LPM is beyond the scope of this dissertation. Therefore, for such information we direct the interested reader to our GitHub repository for LPM [6].

6.2 Overview of the Policies Store

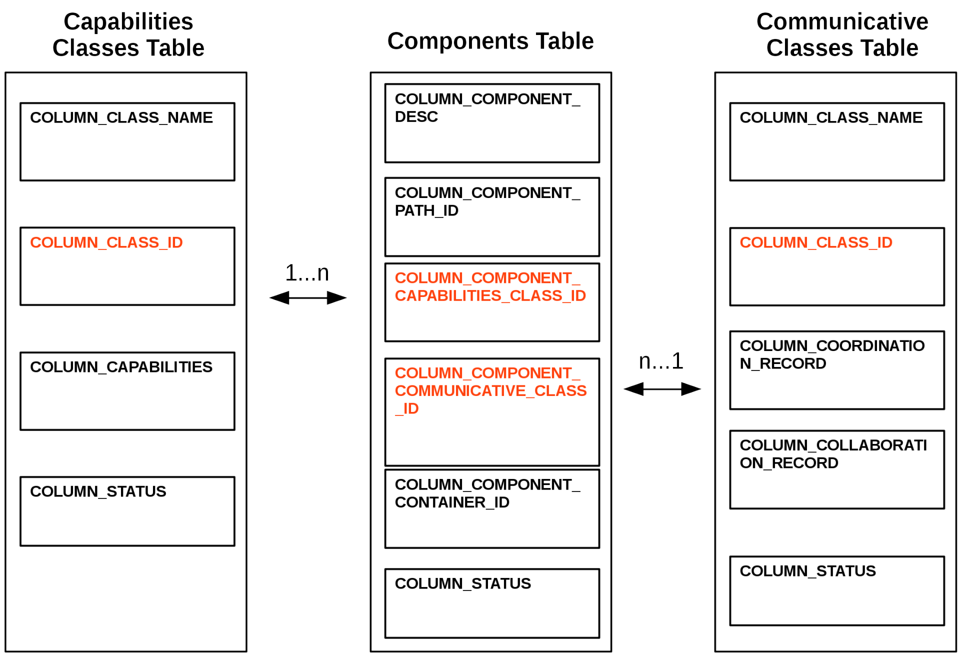

Capabilities Classes Table
COLUMN_CLASS_NAME
COLUMN_CLASS_ID
COLUMN_CAPABILITIES
COLUMN_STATUS
1...n
Components Table
COLUMN_COMPONENT_DESC
COLUMN_COMPONENT_PATH_ID
COLUMN_COMPONENT_CAPABILITIES_CLASS_ID
COLUMN_COMPONENT_COMMUNICATIVE_CLASS_ID
COLUMN_COMPONENT_CONTAINER_ID
COLUMN_STATUS
n...1
Communicative Classes Table
COLUMN_CLASS_NAME
COLUMN_CLASS_ID
COLUMN_COORDINATION_RECORD
COLUMN_COLLABORATION_RECORD
COLUMN_STATUS

Figure 6.2: Database Schema of Policies Store

Persistence Layer of our LPM reference monitor provides the Create / Read / Update / Delete (CRUD) functionality to manage policies using embedded database facilities. For implementation details of the Persistence Layer that is rather complex, we direct the interested reader to our GitHub repository for LPM [6]. We now provide detailed description of the schema, the business logic, the relationship between main tables and the possible access control records involved in our component-oriented access control model. The schema of the embedded database (referred to as the Policies Store) for storing framework's access control policies appears in Figure 6.2. It shows the schema for three tables, namely, *Components Table*, *Capabilities Classes Table*, and

Communicative Classes Table. We begin by describing the attributes of the *Components Table* that is described below.

```
[COLUMN_COMPONENT_PATH_ID, COLUMN_COMPONENT_DESC,
COLUMN_COMPONENT_ID, COLUMN_COMPONENT_CAPABILITIES_CLASS_ID,
COLUMN_COMPONENT_COMMUNICATIVE_CLASS_ID,
COLUMN_COMPONENT_CONTAINER_ID,
COLUMN_COMPONENT_TUPLE_SPACE_PATH, COLUMN_STATUS, UPDATE_COLUMN]
```

The description of individual columns is as follows:

- `COLUMN_COMPONENT_PATH_ID` – absolute path to the location of the service component is currently used as the component ID.
- `COLUMN_COMPONENT_DESC` – description of the service component and its functionality.
- `COLUMN_COMPONENT_ID` – reserved for future use - may be used for alternative ID mechanisms.
- `COLUMN_COMPONENT_COMMUNICATIVE_CLASS_ID` – a reference to the ID of the Communicative Class in Communicative Classes Table in which the component may belong. This is a one-to-one relationship because a component may belong only to one such class.
- `COLUMN_COMPONENT_CAPABILITIES_CLASS_ID` – a reference to the ID of the Capabilities Class in Capabilities Classes Table in which the component may belong. This is a one-to-one relationship because a component may belong only to one such class.
- `COLUMN_COMPONENT_CONTAINER_ID` – reserved for future use with various container technologies.
- `COLUMN_COMPONENT_TUPLE_SPACE_PATH` – reserved for possible use. It indicates the component's tuple space location. Technically this should be determined automatically based on the component's path_ID – TS should be located immediately at the 1st level of component's root directory.

- `COLUMN_STATUS` – not currently used but may be employed for specification of whether a corresponding row in the database table is active or not from the access control standpoint. Such a necessity may arise in the future when a record may still be kept in PS without a need to physically delete it. Therefore, changing its status from active to inactive may be beneficial for such an optimization of PS operation.

- `UPDATE_COLUMN` – used by Persistence Layer to indicate which field to update when calling *write_ComponentsTableRecord()* method that belongs to Persistence Layer [6].

The schema of the *Communicative Classes Table* is given below.

```
[COLUMN_CLASS_ID, COLUMN_CLASS_NAME, COLUMN_COORDINATION_RECORD,
COLUMN_COLLABORATION_RECORD, COLUMN_STATUS, UPDATE_COLUMN]
```

The description of individual columns is as follows:

- `COLUMN_CLASS_ID` – unique Communicative Class ID. This is a one-to-many relationship because several components in Components Table may belong to one such class.

- `COLUMN_CLASS_NAME` – Name of a Class.

- `COLUMN_COORDINATION_RECORD` – a pair: `COMPONENT_1_PATH_ID ; COMPONENT_2_PATH_ID`. If such a record exists, that signifies a permission for coordination.

- `COLUMN_COLLABORATION_RECORD` – a pair: `COMPONENT_PATH_ID ; Data_Object_Absolute_Path`. If such a record exists, that signifies a permission for collaboration.

- `COLUMN_STATUS` – not currently used but may be employed for specifying whether a corresponding row in the database table is active or not from the access control standpoint. Such a necessity may arise in the future when a record may still be kept in PS without a need to physically delete it. Therefore, changing its status from active to inactive may be beneficial for such an optimization of PS operation.

- `UPDATE_COLUMN` – used by internal Persistence Layer to indicate which field to update when calling *write_CommunicativeClassesTableRecord()* method that belongs to Persistence Layer [6].

We now give some examples of coordination and collaboration records.

Example 6.1 Coordination Record Example 1

[/s/missouri/a/nobackup/services/containers/container-1/bin/component-publisher ;

/s/missouri/a/nobackup/services/containers/container-2/bin/component-generator]

Example 6.2 Coordination Record Example 2

[/s/missouri/a/nobackup/services/containers/container-2/bin/component-generator ;

/s/missouri/a/nobackup/services/containers/container-1/bin/component-publisher]

Example 6.3 Collaboration Record Example 1

[/s/missouri/a/nobackup/services/containers/container-1/bin/component-publisher ;

/s/missouri/a/nobackup/services/containers/container-2/data-logs/secure.log]

Example 6.4 Collaboration Record Example 2

[/s/missouri/a/nobackup/services/containers/container-2/bin/component-generator ;

/s/missouri/a/nobackup/services/containers/container-1/data-logs/secure.log]

Note, that both coordination records shown in Examples 6.1 and 6.2 should exist in the table to allow bidirectional coordination between two components via TSC. Any record input is checked for validity before insertion – corresponding component IDs and objects paths should exist in the system. Such conditions are validated by the corresponding implementation logic in Persistence Layer. It also ensures that there are no duplicate rows in corresponding tables and all key constraints are satisfied.

The schema of the *Capabilities Classes Table* is given below.

```
[COLUMN_CLASS_ID, COLUMN_CLASS_NAME, COLUMN_CAPABILITIES,
COLUMN_STATUS, UPDATE_COLUMN]
```

The description of individual columns is as follows:

- `COLUMN_CLASS_ID` – unique Capabilities Class ID. This is a one-to-many relationship because several components in Components Table may belong to one such class.
- `COLUMN_CLASS_NAME` – Name of a Class.
- `COLUMN_CAPABILITIES` – a list of one or more Linux Capabilities assigned to a class.
- `COLUMN_STATUS` – not currently used but could be employed for specification of whether a corresponding row in the database table is active or not from the access control standpoint. Such a necessity may arise in the future when a record may still be kept in PS without a need to physically delete it. Therefore, changing its status from active to inactive may be beneficial for such an optimization of PS operation.
- `UPDATE_COLUMN` – used by internal Persistence Layer to indicate which field to update when calling *write_CapabilitiesClassesTableRecord()* method that belongs to Persistence Layer [6].

Chapter 7

Evaluation and Experimental Results

The deployment of component-oriented access control framework in the real-world settings requires a thorough performance evaluation. The model for capabilities classes does not incur any significant performance overheads for the unified framework. This is because its enforcement is based on the calls to the LibCap library [44] that essentially updates the file system capabilities metadata information for a process [70]. Such operations do not incur the performance overheads because library mediations do not require extra disk I/O aside from the I/O load of the base system [3, 4]. There is also no additional memory utilization required aside from the RAM consumption by the LPM reference monitor itself.

However, the situation is quite different for the model of communicative classes. The enforcement is based on the tuple space paradigm that is known to be quite resource intensive [48]. The performance overheads for a memory-resident global shared tuple space are well known and include memory consumption overheads, efficiency problems with tuple matching at high speeds and search complexity with a large number of divergent tuples present in a single space continuum [62]. Those properties essentially pose a limit on a number of tuple objects in a given tuple space [29, 48, 62]. Taking that into consideration as discussed in Chapter 4, the design of our tuple space implementation is reliant on the alternative strategy of the persistent file system-based solution with personal tuple space per component [7].

7.1 Correctness of Access Control Enforcement

In this section we report the evaluation of correctness that is associated with enforcement of access control policies for the proposed framework. It is important to verify such policies before we measure various aspects of our framework that are related to its performance. Correctness of such access control checks is crucial because without validated enforcement the framework policies could be bypassed by the malicious or compromised service components.

Table 7.1: Correctness of Enforcement Operations - 1

Operation	Policy Class	Output	Enforcement Description
Add capability to a class	Capabilities	Enforced	This adds a given Linux capability for all the components associated with a class to components' binaries in a file system.
Remove capability from a class	Capabilities	Enforced	This removes a given Linux capability for all the components associated with a class from components' binaries in a file system.
Move component to a class	Capabilities	Enforced	This adds all the capabilities listed in a class to a given component's binary in a file system. Consequently, this removes all the capabilities associated with a different class from a previous assignment, if any, from a given component's binary.

The verified operations of the framework in a context of corresponding policy classes are given in Table 7.1 and Table 7.2. The 'Enforcement Description' column in each table provides a necessary explanation of the enforcement outcome per individual operation.

The access control checks are conducted at the level of Enforcer Layer that in turn invokes the Persistence Layer to query the Policies Store for the existence of corresponding policy records. The enforcement has been empirically verified through unit tests at the level of business logic for the proposed framework. The operations involved in enforcement of Linux capabilities on components' binaries have been verified using getcap() [44] utility.

The validations that correspond to Capabilities Classes in Table 7.1 are available at: *https://github.com/kirillbelyaev/tinypm/blob/LPM2/src/test/java/BL_CapabilitiesClasses_UnitTests.java*.

The validations that correspond to Communicative Classes in Table 7.2 are available at: *https://github.com/kirillbelyaev/tinypm/blob/LPM2/src/test/java/BL_CommunicativeClasses_UnitTests.java*

Table 7.2: Correctness of Enforcement Operations - 2

Operation	Policy Class	Output	Enforcement Description
Add collaboration policy to a class	Communicative	Enforced	This adds an explicit permission for a specified component to request a replica of a given data object from a specified location in a file system.
Remove collaboration policy from a class	Communicative	Enforced	This removes an explicit permission for a specified component to request a replica of a given data object from a specified location in a file system.
Add coordination policy to a class	Communicative	Enforced	This adds an explicit permission for a given pair of components to coordinate via individual tuple spaces in a file system.
Remove coordination policy from a class	Communicative	Enforced	This removes an explicit permission for a given pair of components to coordinate via individual tuple spaces in a file system.
Move component to a class	Communicative	Enforced	This provides a mapping of a given component to a specified class. All the operations are checked for such a mapping before execution. Therefore, both components have to be mapped to the same class in order to communicate via collaboration or coordination.

7.2 Performance of Tuple Space Transactions

The initial prototype of the TSL implemented in Java SE is publicly available through the LPM's GitHub repository [6]. The specification of the machine involved in the benchmarking is depicted in Table 7.3. Internal JVM memory utilization and time information has been obtained using JVM's internal Runtime and System packages. We do not provide the benchmarking results for coordinative transaction. Despite its implementation complexity, such a transaction involves only exchange of two control tuples and therefore does not incur any significant performance overheads in terms of CPU and RAM utilization. The cumulative RAM consumption for concurrent coordinative transaction with TSC and service component each executing in separate JVM threads has been observed at merely 44 KB. The actual unit test for coordination is available at: *https://github.com/kirillbelyaev/tinypm/blob/LPM2/src/test/java/TSLib_UnitTests _Coordination.java*.

Table 7.3: Node Specifications

Attribute	Info
CPU	Intel(R) Xeon (R) X3450 @ 2.67 GHz; Cores: 8
Disk	SATA: 3Gb/s; RPM: 10 000; Model: WDC; Sector size: 512 bytes
File System	EXT4-fs; Block size: 4096 bytes; Size: 53GB; Use: 1%
RAM	8 GB
OS	Fedora 23, Linux kernel 4.4.9-300
Java VM	OpenJDK 64-Bit Server SE 8.0_92

For collaboration, the payload of individual content tuple is set at 1 MB. Therefore, for instance, it takes 64 content tuples to replicate a 64 MB data object. Six sizes of data objects have been chosen – 64, 128, 256, 512, 1024 and 2048 MB objects respectively. Collaborative transactional flow, as discussed in Chapter 4, is performed on the EXT4 file system, where the requesting service component creates a tuple space in its isolated directory structure and assembles the content tuples appended by the TSC into a replica in its isolated environment outside the tuple space directory.

Replication performance for sequential collaboration is depicted in Figure 7.1. The create_Object Replica() method in Utilities package of the TSL library is a reference method that sequentially

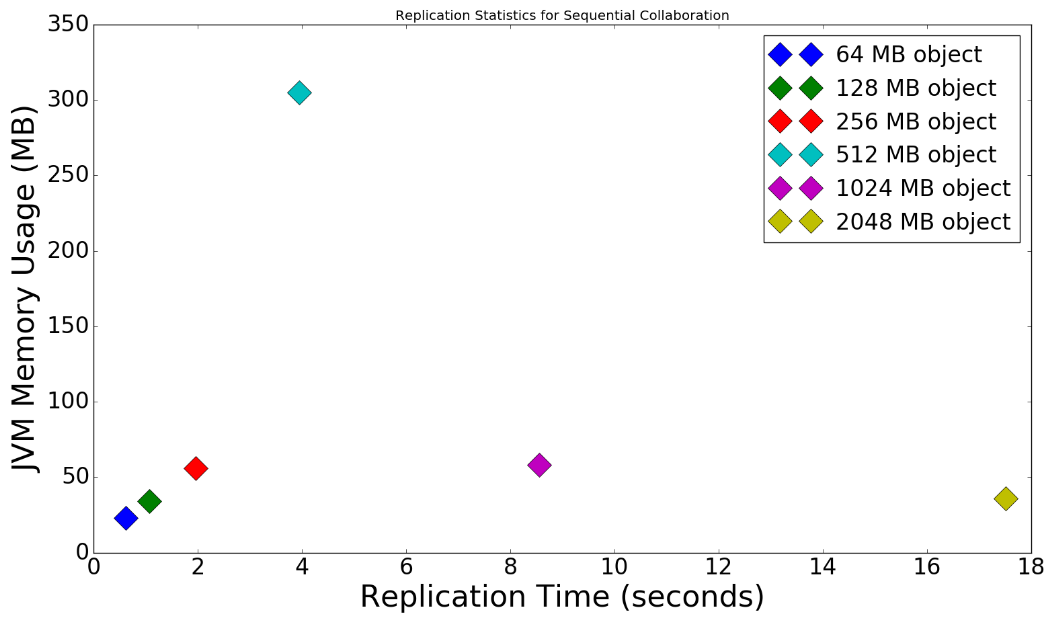


Figure 7.1: Replication Performance for Sequential Collaboration

executes the collaborative transaction conducted between TSC and the service component within a single thread of execution. We can observe that the replication time progressively doubles with an increase of the object size. On average, it takes 0.625 seconds to replicate a 64 MB object, 1.065 seconds a 128 MB object, 1.955 seconds a 256 MB object, 3.950 seconds a 512 MB object, 8.550 seconds a 1024 MB object and 17.505 seconds to replicate a 2048 MB object.

Java Virtual Machine (JVM) memory utilization during sequential collaboration has been observed to be negligible. That is largely due to the usage of Java NIO library [34] in our Utilities package that is designed to provide efficient access to the low-level I/O operations of modern operating systems. On average, memory usage is 23 MB for replication of a 64 MB object, 34 MB for a 128 MB object, 56 MB for a 256 MB object, 305 MB for a 512 MB object (an outlier, repeatedly observed with this object size that might be specific to the garbage collector for this particular JVM), 58 MB for a 1024 MB objects, and 36 MB for replication of a 2048 MB object.

Note, that since the measured JVM memory utilization takes into account the processing of both TSC and requester components within a single thread of execution, the actual JVM utilization will be roughly twice lower for two endpoints in the collaborative transaction when endpoints

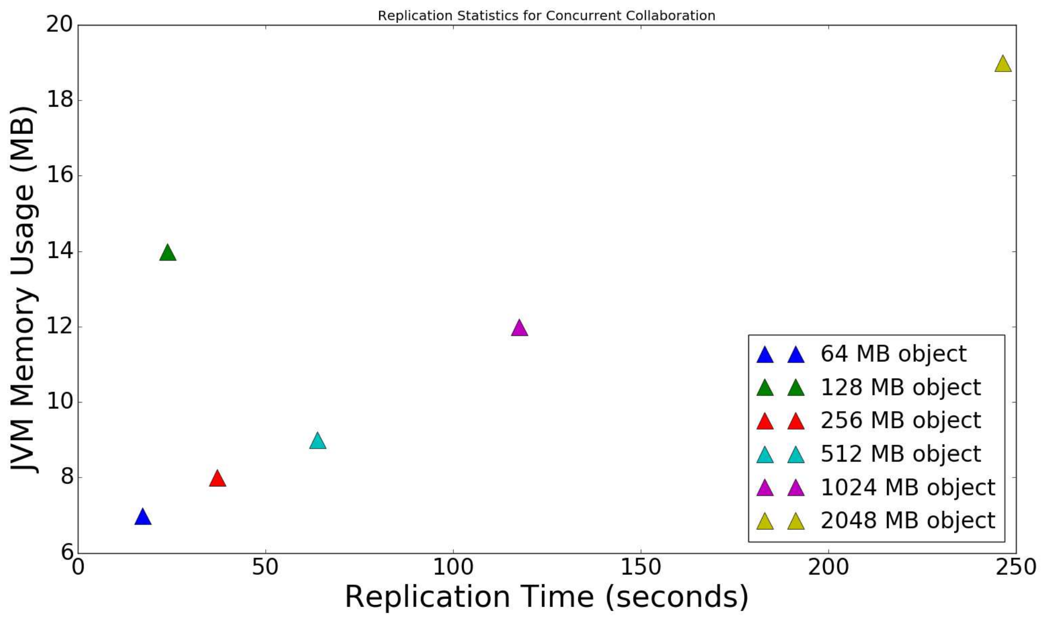


Figure 7.2: Replication Performance for Concurrent Collaboration

execute in separate JVMs. This shows the practical feasibility of our collaborative implementation even for replication of large data objects. According to obtained results, we can anticipate that TSC can handle a large number of concurrent collaborative transactions without consuming significant amounts of physical RAM.

We observed partially full utilization of a single CPU core during replication of the largest data object (2048 MB). The actual unit test for sequential collaboration is available at: *https://github.com/kirillbelyaev/tinypm/blob/LPM2/src/test/java/TSLib_Utilities_UnitTests.java*.

In real-world settings TSC and service component execute concurrently in separate threads, and in fact in different JVMs. Replication performance for concurrent collaboration is depicted in Figure 7.2, where TSC and service component execute as concurrent threads in a single JVM. In such settings, TCS thread performs a short sleep in its section of TSL library after every append operation to allow the service component thread to take a content tuple from its tuple space. That results in a longer replication time compared to sequential execution depicted in Figure 7.1.

Due to concurrent execution, two CPU cores have been partially utilized by the JVM during concurrent collaboration. The obtained results show that replication time is sufficient for non-critical, non-real-time services where medium-size data objects need to be replicated across service components. Further decrease in replication time is possible through the usage of faster storage media, such as Solid-State Drives (SSDs) and Non-Volatile Memory (NVM) [18].

Again, we can observe that the replication time progressively doubles with an increase of the object size. On average, it takes 17.152 seconds to replicate a 64 MB object, 23.8 seconds a 128 MB object, 37.1 seconds a 256 MB object, 63.8 seconds a 512 MB object, 117.5 seconds a 1024 MB object and 246.505 seconds to replicate a 2048 MB object.

In line with sequential collaboration, JVM memory utilization during concurrent collaboration also has been observed to be negligible. On average, memory usage is 7 MB for replication of a 64 MB object, 14 MB for a 128 MB object (an outlier, repeatedly observed with this object size that might be specific to the garbage collector for this particular JVM that is not related to the outlier depicted in Figure 7.1 during sequential collaboration), 8 MB for a 256 MB object, 9 MB for a 512 MB object, 12 MB for a 1024 MB objects, and 19 MB for replication of a 2048 MB object. In fact, the utilization is much lower then in case of sequential collaboration.

Again, when executed in separate JVMs, the memory footprint for every endpoint in the transactional flow will be further diminished. Therefore, TSC memory usage during real-life operations for handling multi-component collaborative transactions is expected to be minimal. Note, that due to preliminary nature of conducted transactional benchmarks, the focus is on functionality, rather then availability. Therefore, no actual saturation of storage media has been attempted. The actual unit test for concurrent collaboration is available at: ***https://github.com/kirillbelyaev/tinypm/blob/LPM2/src/test/java/TSLib_UnitTests_Collaboration.java***.

Note, that due to basic DoS protection properties of collaborative transaction that requires a consumption of every content tuple before a new one is appended to a given tuple space, the speed of object replication in our solution is theoretically slower in comparison to existing IPC mechanisms that do not address DoS issues. The preliminary comparison of replicating a 1 GB data object via the existing UNIX pipe mechanism shows a 6 to 8 times decrease in object replication time in comparison with our sequential collaborative transaction. However, pipe-based replication is purely based on unidirectional byte flow. In our solution, a complex sequence of tuple space operations on the file system is performed by endpoints involved in the transactional flow. Such transactional logic that is performed at the granularity of individual tuples inherently requires synchronization and additional computational overheads including access control checks that results in longer execution time.

7.3 Load Simulation of Tuple Space Controller

The focus of our access control framework is on provision of flexible types of information flow between infrequently communicating (not real-time) service components. Therefore, as already noted, it is not intended for High-Performance Computing (HPC) services and large data object transfers between components. Nevertheless, it is important to provide an accurate estimate of the resource consumption incurred by the TSC that is responsible for the information flow transport. In this part, we provide the actual memory allocation (Resident Set Size (RSS) info obtained through ps and htop utilities) – the non-swapped physical memory that a replication task (a single collaborative transaction) has used. In contrast to the previously reported JVM memory allocation that is subject to arbitrary fluctuations related to garbage collection mechanism, RSS provides the real memory allocation estimates on the actual hardware. The TSC load simulation has been conducted on the enhanced server hardware. The specification of the machine is depicted in Table 7.4.

Table 7.4: Server Node Specifications

Attribute	Info
CPU	Intel(R) Xeon (R) E5520 @ 2.27 GHz; Cores: 8
Disk	SAS: 3 Gb/s; RPM: 15 000; Model: Fujitsu; Sector size: 512 bytes
File System	EXT4-fs; Block size: 4096 bytes; Size: 40GB; Use: 1%
RAM	24 GB
OS	Fedora 24, Linux kernel 4.7.5-200
Java VM	OpenJDK 64-Bit Server SE 8.0_102

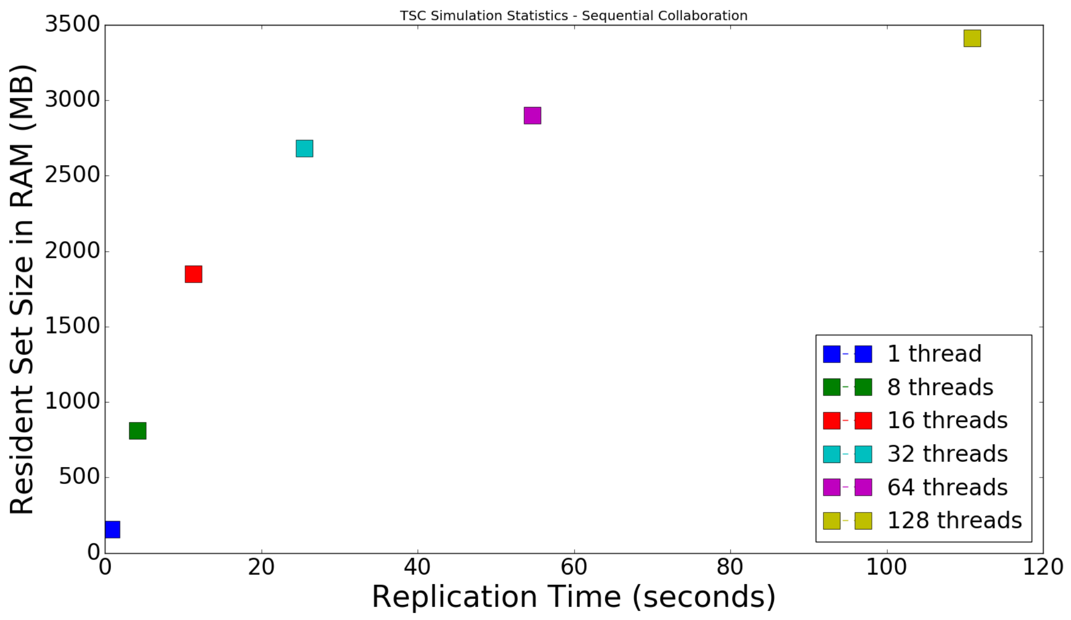


Figure 7.3: Tuple Space Controller Simulation Information for a 64 MB Object

We have simulated the tentative resource consumption by the TSC conducting concurrent sequential collaborative transactions for a large number of service components – 1, 8, 16, 32, 64 and 128 concurrent transactions respectively, each executing in a separate JVM thread. Note, that this necessitates a creation and corresponding removal of persistent tuple spaces per individual transaction. Therefore, for instance 128 transactions will create and then consequently remove 128 such tuple spaces after completion of transactional execution.

Two object sizes have been used for replication – 64 MB for the lower bound and 2048 MB for the upper bound of the load assessment. No external processes unrelated to benchmarking were present during the simulation load on the system. Note, that every individual experiment has been repeated for up to 10 times to verify the consistency of performance indicators. The actual unit tests for TSC load simulation are available at: *https://github.com/kirillbelyaev/tinypm/blob/LPM2/src/test/java/TSLib_TSC_SequentialCollaboration_UnitTests.java*.

We observed that the load has been evenly distributed on all CPU cores by the JVM and OS SMP facility and initially utilizes 100% on every core, gradually decreasing, as every thread executing collaborative transaction passes the peak work section. Our main focus is on RAM con-

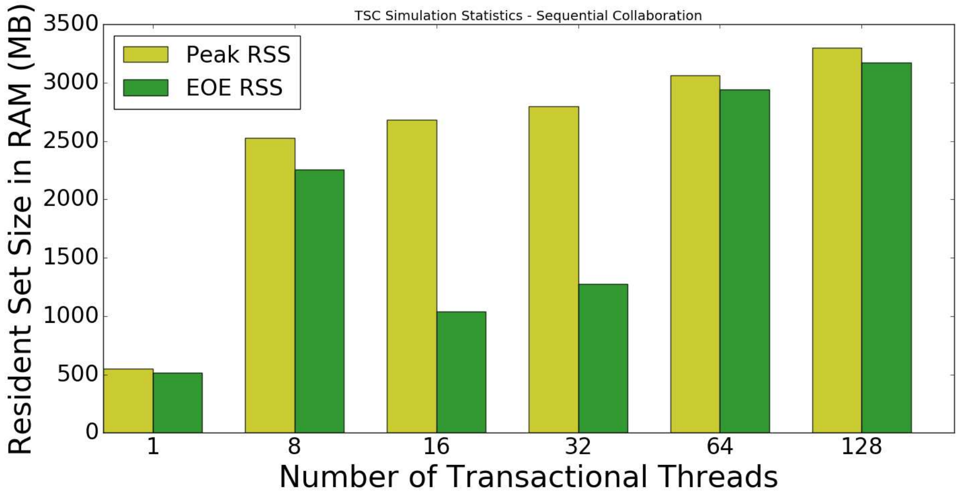


Figure 7.4: Tuple Space Controller Simulation Information for a 2048 MB Object

sumption since it is critically important to estimate its usage by potentially tens of concurrent transactions performed by TSC.

The replication time and RSS info associated with a 64 MB data object is depicted in Figure 7.3. For a single transaction, executing within a single JVM thread, the RSS is 159 MB with replication time at 0.78 s. For 128 concurrent transactions, executing in independent JVM threads, the cumulative upper bound for RSS is 3415 MB with last transaction to complete replication at the threshold of 110.858 s. The replication time nearly doubles with the proportional increase of the number of objects that need to be replicated. However, the memory usage does not generally double with load increase. The upper bound of 3415 MB represents only 14 to 15% of the available system memory depicted in Table 7.4. Therefore, the simulation shows that TSC could be rather memory-efficient on a larger scale with replication of small data objects for a large number of requesting service components.

The RSS info, associated with replication of a 2048 MB data object is depicted in Figure 7.4. Due to limitations of disk partition size we have not been able to run the 32, 64 and 128 thread experiments until completion to record the final replication time. The experiments terminate at the point of partition space saturation. However, two key indicators have been observed: peak RSS shows the highest memory allocation observed for a repeated number of simulations; End of Execution (EOE) RSS shows the highest memory allocation observed at transaction termination time for a repeated number of simulations.

For a single transaction, executing within a single JVM thread, the peak RSS was observed to be 552 MB with EOE RSS at 515 MB. We can see a consistent increase in peak RSS with progressive increase in the number of transactions. For 128 concurrent transactions, executing in independent JVM threads, the peak RSS reaches 3298 MB with EOE RSS at 3170 MB. However, we do not see the drastic difference in memory consumption between 8 and 128 transactional threads. In fact, peak RSS for 8 concurrent threads is 2525 MB – a mere 700 MB difference observed with peak RSS for 128 threads. Such results, once again, empirically confirm that actual TSC implementation could be rather memory efficient at a larger scale, occupying only a fraction of available system RAM at peak load times on modern server hardware.

The important fact to observe is that the peak RSS with the same number of concurrent transactions has been nearly identical for two different data object sizes depicted in Figure 7.4 and Figure 7.3. In fact, the RSS is slightly larger for 128 transactions associated with a 64 MB data object. This shows that object size does not have a strong impact on the real memory usage of individual collaborative transaction conducted through our TSL implementation. As already mentioned, this is largely due to the use of Java NIO library [34] in our TSL implementation. It also shows that modern Java platforms could often times provide a viable alternative to compiled languages such as C++ for complex and secure enterprise-grade middleware implementations [68].

For the sake of completeness, we provide the actual replication time before disk saturation observed for the TSC load simulation associated with a 2048 data object depicted in Figure 7.4. For a single transaction executing within a single JVM thread the average replication time is 22.406 s. For 8 threads the average replication time is 199.545 s. For 16 threads the average replication time to concurrently complete the replication of 16 2048 MB data objects is reported to be 431.72 s. Note, that reported replication time might be nearly irrelevant in real-world settings where dozens of concurrently running service components could add additional I/O load on the server storage hardware with service-specific file system activity. That could significantly increase the completion time of a single collaborative transaction for a set of large data objects.

7.4 Performance of Enhanced Tuple Space Transactions

In this section we report the performance of the enhanced implementation of TSTs. The enhancement avoids potential race conditions within *append* operation between TSC and service component in a single TS. This is achieved via writing both types of tuples in the form of hidden objects. This property is achieved via appending '.' to the name of an object on a given file system.

Only after tuple object is fully written in TS it is exposed via rename operation. We have also increased the sleep period on a side of the TSC (in fragment_ObjectReplica() method within Utilities package of TSL library) for the collaborative transaction that results in longer replication time in comparison to original version. However, the replication time is now consistent through multiple test runs. Such an enhancement offers increased reliability in the event of multiple concurrent collaborative transactions that replicate large data objects on the server instance.

Experiments have been conducted on the machine with the same server parameters depicted in Table 7.4. Replication time and RSS information for various sizes of a data object for such an enhancement of collaborative transaction is depicted in Figure 7.5. Here, TSC and service component again execute as concurrent threads in a single JVM. Note, that we do not report the statistics for a 2 GB object since in real service settings large data objects will be stored/rotated at 1 GB granularity to avoid long replication times. We also do not measure internal JVM memory utilization because it does not reflect the actual physical memory usage and is not consistent through repeated experiments.

The RSS info has been recorded to be approximately the same for all data object sizes. We take into account the final End of Execution (EOE) RSS that shows the highest static RSS in physical RAM allocated by JVM at the termination time of collaborative transaction. We do not take into

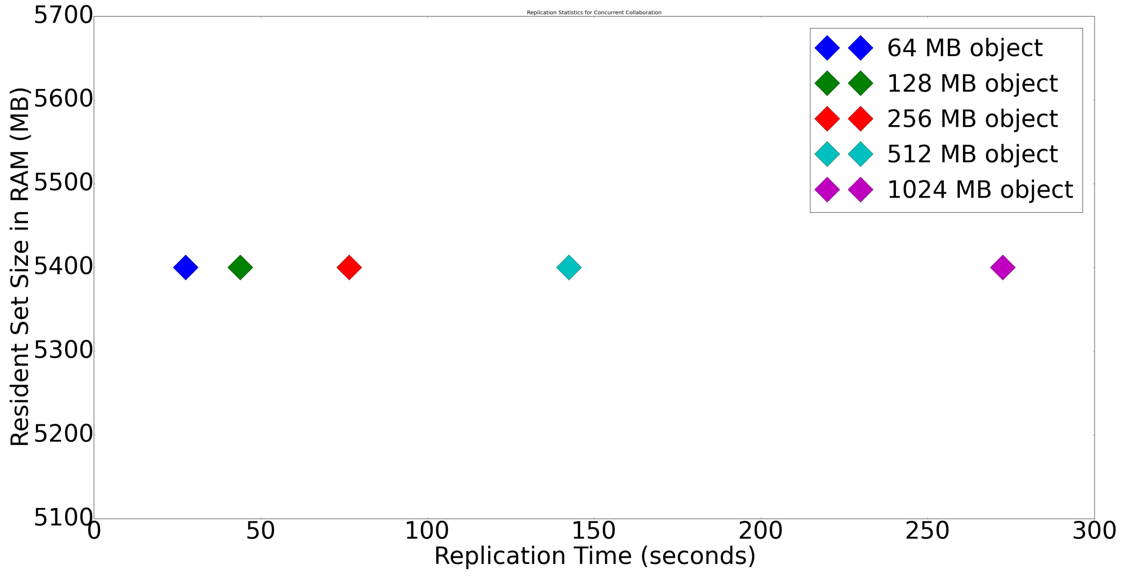


Figure 7.5: Replication Performance for Concurrent Collaboration

account the fluctuating spikes in memory usage during the runtime of individual transaction that last only several milliseconds.

The highest value of EOE RSS through repeated experiments was recorded under the boundary of 5400 MB. In line with previous experiments it reinforces the fact that object size does not have a significant correlation with the physical memory usage of individual collaborative transaction conducted through our TSL implementation.

Note that, as previously stated, in real-world settings TSC and service component execute in separate JVMs. Therefore, the memory footprint for every endpoint in the transactional flow will be further diminished – roughly a two-fold decrease. Again, we can observe that the replication time progressively doubles with an increase in the object size. On average, it takes 27.524 s to replicate a 64 MB object, 43.9 s for a 128 MB object, 76.534 s for a 256 MB object, 142.4 s for a 512 MB object and 272.505 s to replicate a 1024 MB data object.

Chapter 8

Discussion on Framework Properties

This chapter provides discussion on security and system properties of our access control framework.

8.1 Security Aspects

In this section, we discuss the security of our component-oriented access control framework. Each component executes in an isolated runtime environment. Such a ’jail’ directory is assigned a distinct unprivileged UID to utilize existing Linux DAC mechanism. A chroot on UNIX/Linux OS is an operation that changes the apparent root directory for the current running process and its children. A program that is run in such a modified environment cannot name and therefore normally cannot access files outside the designated directory tree. The isolated runtime is populated with all required program files, configuration files, device nodes and shared libraries that are required for the successful service component execution. Note that, the shared libraries (if present) have to be properly audited before inclusion in the runtime environment [8].

Our framework aims to give each component only those capabilities that it needs and only those inter-component interactions that are required for offering services. We assume that the LPM which is our reference monitor and all parts of its Trusted Computing Base (TCB) such as TSL and LibCap libraries are trusted. We also assume that the capabilities policies and communicative policies have been written correctly. The LPM is responsible for executing these policies and putting components in their respective policy classes. Thus, a component can access only those system resources that is permitted by the capabilities policy class to which it has been assigned.

As indicated in Section 2, the policies of our component-oriented access control framework are centrally enforced via a user-space reference monitor. Therefore, individual service components may not be aware of specific policies that are being enforced upon them. For instance, a network component may not be aware that a specific OS-level Linux capability has been enforced upon its binary since our centralized enforcement would be completely transparent to such a component.

The enforcement of interaction across components is also completely transparent to them. Service components are not aware of a reference monitor's mediation and therefore the regulation of communication between them is transparent. For that matter, in contrast to DIFC mechanisms where applications may be able to dynamically declare and change access policies depending on the privileges delegated to them [38], our centralized enforcement paradigm does not provide such a freedom for individual components. That is because policies for individual service components in our settings are predefined within a lifecycle of a particular service and are not subject to change. Moreover, the mechanism of excessive trust delegation to untrusted components may be potentially detrimental in multi-tenant settings where a single OS may host multiple services that belong to different legal entities.

In our design, the communication across components takes place through their individual tuple spaces. Thus, we must protect the confidentiality, integrity, and availability of the tuple space of each component. Each component in an isolated runtime environment has a directory structure within the file system in which it can create its own tuple space. The communicative policies bind a component to its tuple space location. Only the individual component can perform all the operations, namely, *create tuple space*, *delete tuple space*, *read*, *append*, and *take*. The LPM reference monitor can only perform *read* and *append* operations on the tuple space. Thus, no one other than the component itself can remove anything from its tuple space. A service component cannot access the tuple space of its peers – this protects the confidentiality and integrity of the tuple space and the data contained in it. A service component cannot also cause denial-of-service on another component's tuple space. This is because even if a component requests a large sized data object, this data is decomposed into fixed size chunks and only one chunk at a time is loaded into and transferred from the tuple space.

Coordination has also been designed with basic security in mind. Coordinative transaction requires a second (confirmative) control tuple to be delivered to the initiating service component to complete the transactional flow. In mission-critical scenarios, a reference monitor cannot be fully trusted and no assumption should be made by the initiating component that its original coordinative message has been successfully delivered to destination. Therefore, a confirmation in the form of a second control tuple from the destination should be delivered to source. Note, that a compromised reference monitor can append a forged control tuple mimicking the confirmation. However, such a possibility is prevented in our framework, since LPM follows strict tuple space calculus [8]. Furthermore, such a forgery could be detected and precluded at the level of service logic where individual service components can deploy a higher level security protocol [62] that operates on fully encrypted message fields of individual control tuples.

Note, that although regulated communication flow is aimed at closing all known covert channels between unrelated services and their components, in practice that is hard to achieve. Some covert channels (such as covert timing channels) can be reduced but not eliminated, particularly for systems connected to the network [38, 74]. Components that are delegated a capability of network access can potentially inter-communicate via network channels and additional measures such as IP firewall rules should be used to preclude such a possibility.

A malicious entity cannot impersonate as an honest component and compromise the confidentiality and integrity of the data of any component. The components and their tuple space in the directory structure are binded in the policy store. Consequently, even if a malicious entity poses as an honest component, it will not be able to access the tuple space that is in the directory structure of the honest component. If a component is dishonest or has a trojan horse, it will get access only to those resources that are allowed by the policy classes that contain it.

One of possible secure deployments is shown in Figure 8.1 where user-space LPM RM is deployed under unprivileged UID with elevated privileges using Linux capabilities within the main OS while service components are placed in OS-level containers such as chrooted jails. LPM has limited file system access to tuple spaces of individual components and read-only access to data directories where components may generate respective data objects. Note, that individual service components may have layered file system structure [69] within their respective OS containers

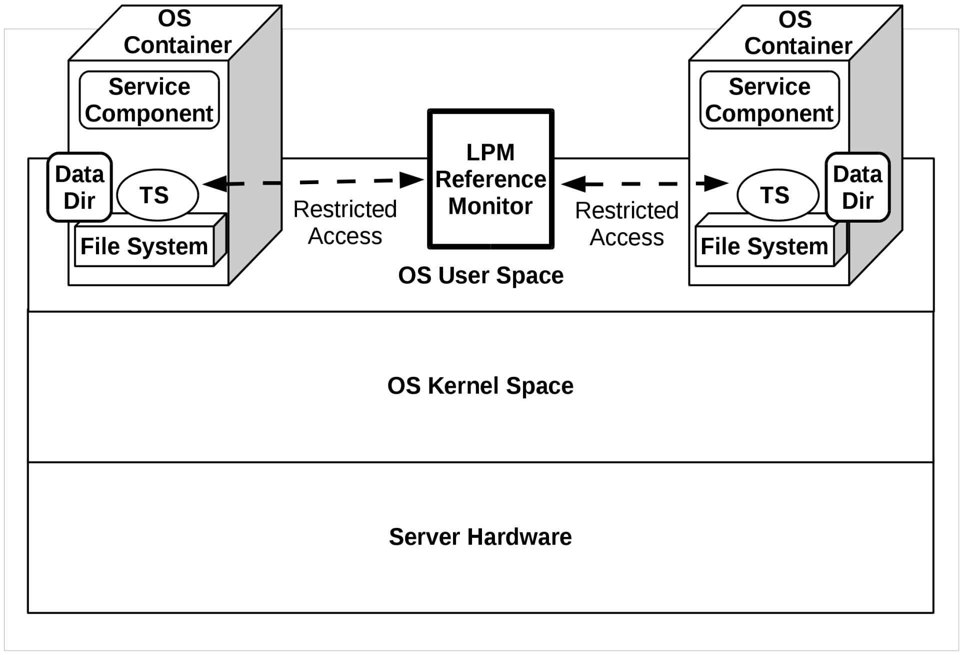


Figure 8.1: Secure Deployment

where permanent storage of data objects is located in a directory hierarchy that is completely inaccessible by the reference monitor.

8.2 User-Space Design

Our framework is supported through the user-space LPM reference monitor which becomes the main differentiator in contrast to most existing works (covered in Chapter 2) that are mainly based on kernel-space solution such as SELinux [57] and DTE [3]. In contrast to such kernel-level solutions, LPM provides access control facilities that are mainly oriented towards containerized services that operate in user space under unprivileged UIDs and may need incremental access to OS and hardware resources mediated through kernel. The isolated components of such services that require specific elevated privileges are given them through the notion of LPM's capabilities classes abstraction (discussed in Chapter 3) without a need to incorporate direct kernel-level support into the reference monitor. At the same time, secure information flow between such components does not require kernel-space support because components exchange business logic flows through customized tuple space abstraction in user-space. In fact, mediation of intensive data flows through kernel-level IPC has two inherent problems. First, its performance is architecturally limited by the cost of invoking the kernel and mediating cross-address space interaction for two communicating components. Inherently, the cost of context switching from kernel space to user space for large data transfers can be detrimental to overall system performance [11]. Second, service components execute in user space and therefore benefit from IPC mechanism implemented through user-level libraries [11] such as our TSL implementation that avoids complexities of message passing that requires additional synchronization primitives [8]. For instance, user-space IPC daemons such as D-Bus [33] (discussed in Chapter 4) have been specifically designed to leverage such user-space advantages. Moreover, having the kernel copy application-level data and coordination messages between address spaces of interacting service components is neither necessary nor sufficient to guarantee safety [11, 28, 38].

One of the main advantages of the user-space design for the reference monitor are portability, ease of implementation and in some sense correctness. LPM does not destabilize the kernel [8, 38]. At the same time, our user-space reference monitor may benefit from tighter integration with OS kernel through user-level interface such as Linux Security Modules [70] (LSM) hooks or related mediation layers. For instance, LPM already utilizes the capabilities management through calls to user-space LibCap [44] library that has direct interface to kernel. Note, that enforcement of more fine-grained, execution-specific security policies for individual isolated service components is possible through existing kernel-space access control solutions such as SELinux that may be used in combination with our user-space reference monitor [9].

For instance, user-space reference monitor such as Flume [38] runs a small component in the kernel that is accessed via LSM to implement a layer of system call interposition for regulation of IPC flows. CamFlow [54], another similar DIFC solution with controlled data sharing for cloud services also uses a kernel module, CamFlow-LSM that is implemented in the form of LSM for OS-level IFC enforcement.

Chapter 9

Conclusion and Future Work

In this chapter we summarize the contributions (Section 9.1) of this dissertation and provide an overview of plans for future related work (Section 9.2).

9.1 Contributions

The contribution of this dissertation includes (1) a state-of-the-art survey on access control models in the scope of our work, (2) a capabilities classes model for components that manages access to system resources using Linux capabilities, (3) a communicative classes model for components that manages inter-component interaction, (4) a tuple space paradigm for enforcement of policies for communicative classes, (5) a research prototype that provides a reference implementation of the unified access control framework, and (6) an evaluation of the proposed framework.

For the state-of-the-art survey, we conducted a systematic literature review on access control related to systems research in the context of identified set of problems. The systematic review compared relevant works in access and information flow control and identified their limitations through a systematic evaluation.

We proposed a component-oriented access control framework for secure deployment of services within a single Linux OS instance. The framework consists of two models and is based on a concept of policy objects, referred to as policy classes.

Specifically, as part of the framework, we proposed a model for capabilities classes. Each capability class consists of policies that are associated with a set of Linux capabilities [44]. The capabilities classes differ from each other on the basis of capabilities they possess. Each service component is placed in at most one capabilities class. The OS resources that the component can access depends on the Linux capabilities associated with that class. Our implementation of the model provides a way by which Linux capabilities can be administered to the services executing in isolated environments.

We also proposed a model for communicative classes. Communicative classes are needed for communication of components that belong to different isolated environments. Each communicative class consists of policies for inter-component interaction. Our implementation of the model provides a way by which such communication can be administered to a set of services.

We also implemented a mechanism necessary for the enforcement of communication policies within such communicative classes. We adapted the generative communication paradigm introduced by Linda programming model [29] which uses the concept of tuple spaces for process communication. We enhanced the original paradigm to address the problems of security and operational limitations and provided enforcement rules such that only components belonging to the same communicative class can communicate using this approach. We introduced and implemented the well-formed Tuple Space Transactions that provide fine-grained regulation between data and control flows for interacting service components. The introduced notion of such transactions also incorporates the basic protection against denial-of-service attacks by malicious service components.

We developed a research prototype that provides a reference implementation for the proposed unified framework in the form of LPM middleware. Our LPM allows the formulation, management and enforcement of access policies to OS resources for individual service components and it also allows regulated inter-component communication across isolated environments. LPM is resident in user-space and it acts as a form of a reference monitor [38] that allows high interoperability and usage of the framework on any general-purpose Linux OS without a requirement for custom kernel patching [8, 38].

We have evaluated the reference implementation of the proposed framework. Specifically, we have assessed the performance of enforcement mechanism for inter-component interaction based on the adaptation of tuple space paradigm. Contrarily to the enforcement based on Linux capabilities, tuple space paradigm is known to be quite resource intensive. We conducted systematic benchmarks with a large number of concurrent tuple space transactions involved in replication of large data objects. The evaluation results suggest that our adaptation of the tuple space paradigm can be utilized for regulated inter-component interaction by a multitude of server-side multi-component services without significant consumption of computational resources. As part of the evaluation, we also assessed the correctness for enforcement of access control policies for both types of policy classes.

9.2 Future Work

The potential to deploy a multitude of services within a single OS offers a higher degree of control over service workflows on modern hardware. Such deployments are becoming increasingly common for cloud service providers. As part of future work, we plan to extend the proposed access control framework for distributed settings [58] where service components may be located at separate server nodes for the purpose of hardware optimization for specific workloads [74]. From the perspective of access control such a distribution will require the design of an aggregation layer that provides the transparency property for the management of distributed service components and enforcement of corresponding component-oriented access policies.

Another future direction of this research is to investigate the applicability of proposed component-oriented framework for provision of security guarantees to monolithic applications through the mechanism of compartmentalization. Many monolithic server applications such as HTTP application, web and cache servers, as well as many other system services are often designed as a single stand-alone process executable. Such a legacy design is far from being secure and reliable. However, such applications may be decomposed into a set of application compartments – independent modules that are based on functionality [13]. Each module may execute a part of the application logic in a separate process and could be isolated from other modules in the system. For that matter, it may be worthwhile to investigate the applicability of proposed models and technologies for coordination and potential exchange of data flows across such modules. Moreover, it will be worthwhile to see how the failure of one module impacts other modules and what types of coordination constructs are needed to make the application secure and resilient.

Another interesting direction of relevant future research is the design of protocols for the provision of encrypted inter-component coordination flow that operate on top of tuple space transactions and do not rely on key distribution and exchange via the potentially vulnerable reference monitor. Such protocols may reduce a potential risk of decrypting the critical control flows between coordinating service components by the reference monitor. In such an event a monitor may be instructed to decrypt the flows based on the availability of encryption/decryption keys in the isolated environment of the deployed component. It then may potentially alter the original content of individual coordination messages that may cause the disruption of established service workflow. However, if appropriate protocols are designed to address such an issue, the impact of a breach may be minimized. Therefore, the possibility of such a protective enhancement should be adequately investigated in the future.

Bibliography

[1] M.D. Abrams, K.W. Eggers, LJ LaPadula, and IM Olson. Generalized Framework for Access Control: An Informal Description. In *Proc. of NCSC*, pages 135–143, 1990.

[2] Alessandro Armando, Roberto Carbone, Gabriele Costa, and Alessio Merlo. Android Permissions Unleashed. In *Proc. of IEEE CSF*, pages 320–333, 2015.

[3] Lee Badger, Daniel F Sterne, David L Sherman, Kenneth M Walker, and Sheila Haghighat. Practical Domain and Type Enforcement for UNIX. In *Proc. of IEEE S & P*, pages 66–77, 1995.

[4] Lee Badger, Daniel F Sterne, David L Sherman, Kenneth M Walker, and Sheila A Haghighat. A Domain and Type Enforcement UNIX Prototype. *Computing Systems*, 9(1):47–83, 1996.

[5] Davide Balzarotti, Paolo Costa, and Gian Pietro Picco. The LighTS Tuple Space Framework and its Customization for Context-Aware Applications. *WAIS*, 5(2):215–231, 2007.

[6] Kirill Belyaev. Lightweight Policy Machine (LPM) for Linux – Component-Oriented Access Control for "containerized" service environments. https://github.com/kirillbelyaev/tinypm/tree/LPM2, 2016. accessed 18-September-2016.

[7] Kirill Belyaev and Indrakshi Ray. Component-Oriented Access Control for Deployment of Application Services in Containerized Environments. In *Proc. of CANS*, pages 383–399, 2016.

[8] Kirill Belyaev and Indrakshi Ray. Towards Access Control for Isolated Applications. In *Proc. of SECRYPT*, pages 171–182, 2016.

[9] Kirill Belyaev and Indrakshi Ray. Component-Oriented Access Control – Application Servers Meet Tuple Spaces for the Masses. *FGCS*, page to appear, 2017.

[10] Kirill Belyaev and Indrakshi Ray. On the formalization, design, and implementation of component-oriented access control in lightweight virtualized server environments. *C&S*, 71:15–35, 2017.

[11] Brian N Bershad, Thomas E Anderson, Edward D Lazowska, and Henry M Levy. User-level Interprocess Communication for Shared Memory Multiprocessors. *ACM TOCS*, 9(2):175–198, 1991.

[12] Matthew A. Bishop. *The Art and Science of Computer Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[13] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proc. of USENIX NSDI*, pages 309–322, 2008.

[14] Kevin Boos, Emilio Del Vecchio, and Lin Zhong. A Characterization of State Spill in Modern Operating Systems. In *EuroSys*, pages 389–404, 2017.

[15] Michele Bugliesi, Stefano Calzavara, Riccardo Focardi, and Marco Squarcina. Gran: Model checking grsecurity RBAC policies. In *IEEE CSF*, pages 126–138, 2012.

[16] Vitaly Buravlev, Rocco De Nicola, and Claudio Antares Mezzina. Tuple Spaces Implementations and Their Efficiency. In *Proc. of COORDINATION*, pages 51–66, 2016.

[17] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. XML Dataspaces for Mobile Agent Coordination. In *Proc. of ACM SAC*, pages 181–188, 2000.

[18] Xianzhang Chen, Edwin H-M Sha, Qingfeng Zhuge, Weiwen Jiang, Junxi Chen, Jun Chen, and Jun Xu. A Unified Framework for Designing High Performance In-Memory and Hybrid Memory File Systems. *JSA*, 68:51–64, 2016.

[19] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-Application Communication in Android. In *Proc. of ACM MobiSys*, pages 239–252, 2011.

[20] CoreOS Developers. What is CoreOS? https://coreos.com/docs/, 2016. accessed 18-September-2016.

[21] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. Coordination and Access Control in Open Distributed Agent Systems: The TuCSoN Approach. In *Proc. of COORDINATION*, pages 99–114. Springer, 2000.

[22] Docker Developers. What is Docker? https://www.docker.com/what-docker/, 2016. accessed 18-September-2016.

[23] David Ferraiolo, Vijayalakshmi Atluri, and Serban Gavrila. The Policy Machine: A Novel Architecture and Framework for Access Control Policy Specification and Enforcement. *JSA*, 57(4):412–424, 2011.

[24] David Ferraiolo, Janet Cugini, and D Richard Kuhn. Role-Based Access Control (RBAC): Features and Motivations. In *Proc. of ACSAC*, pages 241–248, 1995.

[25] David Ferraiolo, Serban Gavrila, and Wayne Jansen. On the Unification of Access Control and Data Services. In *Proc. of IEEE IRI*, pages 450–457, 2014.

[26] David F Ferraiolo and D Richard Kuhn. Role-Based Access Control. *Proc. of NCSC*, pages 554–563, 1992.

[27] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM TISSEC*, 4(3):224–274, 2001.

[28] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*, 2004.

[29] David Gelernter. Generative Communication in Linda. *ACM TOPLAS*, 7(1):80–112, 1985.

[30] GrSecurity Developers. What is GrSecurity? https://grsecurity.net, 2016. accessed 18-September-2016.

[31] Serge Hallyn and Phil Kearns. Domain and Type Enforcement for Linux. In *Proc. of ALS*, pages 247–260, 2000.

[32] Serge E Hallyn and Andrew G Morgan. Linux Capabilities: Making them Work. In *Proc. of Linux Symp.*, pages 163–172, 2008.

[33] Inc. Havoc Pennington, Red Hat. D-Bus Specification. https://dbus.freedesktop.org/doc/dbus-specification.html, 2016. accessed 18-September-2016.

[34] Java NIO Developers. Java Non-blocking I/O Library. http://en.wikipedia.org/wiki/Non-blocking_I/O_(Java), 2016. accessed 25-October-2016.

[35] Michael K Johnson and Erik W Troan. *Linux Application Development*. Addison-Wesley Professional, 2004.

[36] Poul-Henning Kamp and Robert Watson. Jails: Confining the Omnipotent Root. In *Proc. of SANE*, pages 1–15, 2000.

[37] Poul-Henning Kamp and Robert Watson. Building Systems to be Shared, Securely. *ACM Queue*, 2(5):42–51, 2004.

[38] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *Proc. of SOSP*, pages 321–334, 2007.

[39] Leonard LaPadula. Rule-Set Modeling of Trusted Computer System. In Abrams M., Jajodia S., and Podell H., editors, *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press, 1995.

[40] Linux Containers Developers. What are Linux Containers? https://linuxcontainers.org/lxc/introduction/, 2016. accessed 18-September-2016.

[41] Linux Developers. Linux Programmer's Manual. http://man7.org/linux/man-pages/man7/capabilities.7.html, 2016. accessed 18-September-2016.

[42] Linux Kernel Developers. Transparent Proxy Support. https://www.kernel.org/doc/Documentation/networking/tproxy.txt, 2017. accessed 10-March-2017.

[43] Linux Programmer"s Manual. Kernel Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html, 2016. accessed 18-September-2016.

[44] Linux Programmer's Manual. LIBCAP Manual. http://man7.org/linux/man-pages/man3/libcap.3.html, 2016. accessed 18-September-2016.

[45] Peter Loscocco. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proc. of USENIX ATC, FREENIX Track*, pages 29–42, 2001.

[46] Roberto Lucchi and Gianluigi Zavattaro. WSSecSpaces: a secure data-driven coordination service for web services applications. In *Proc. of ACM SAC*, pages 487–491, 2004.

[47] Medusa Developers. Medusa DS9 Security System. http://medusa.terminus.sk/English/project.shtml, 2016. accessed 18-September-2016.

[48] Naftaly H Minsky, Yaron M Minsky, and Victoria Ungureanu. Making Tuple Spaces Safe for Heterogeneous Distributed Systems. In *Proc. of ACM SAC*, pages 218–226, 2000.

[49] Naftaly H Minsky and Victoria Ungureanu. Unified support for heterogeneous security policies in distributed systems. In *Proc. of USENIX SS*, pages 131–142, 1998.

[50] Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *Advanced Linux Programming*. NRP, 2001.

[51] Andrew C Myers and Barbara Liskov. Protecting Privacy using the Decentralized Label Model. *ACM TOSEM*, 9(4):410–442, 2000.

[52] n–Logic Ltd. n-Logic Web Caching Service Provider. http://n-logic.weebly.com/, 2016. accessed 18-September-2016.

[53] Amon Ott and Simone Fischer-Hübner. The Rule Set Based Access Control (RSBAC) Framework for Linux. In *Proc. of ILK*, 2001.

[54] Thomas FJ-M Pasquier, Jatinder Singh, David Eyers, and Jean Bacon. CamFlow: Managed data-sharing for cloud services. *IEEE TCC*, 5(3):472–484, 2017.

[55] Daniel Price and Andrew Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *LISA*, volume 4, pages 241–254, 2004.

[56] Indrajit Roy, Donald E Porter, Michael D Bond, Kathryn S Mckinley, and Emmett Witchel. Laminar: Practical Fine-Grained Decentralized Information Flow Control. *ACM SIGPLAN Notices*, 44(6):63–74, 2009.

[57] SELinux Developers. Security Enhanced Linux. http://selinuxproject.org, 2016. accessed 18-September-2016.

[58] Jatinder Singh, Jean Bacon, and David Eyers. Policy enforcement within emerging distributed, event-based systems. In *Proc. of ACM DEBS*, pages 246–255, 2014.

[59] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *ACM SIGOPS OSR*, volume 41, pages 275–287. ACM, 2007.

[60] R Spencer, S Smalley, P Loscocco, M Hibler, D Andersen, and J Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proc. of USENIX SS*, pages 123–139, 1999.

[61] SQLite Developers. SQLite. https://www.sqlite.org/, 2016. accessed 18-September-2016.

[62] Jan Vitek, Ciarán Bryce, and Manuel Oriol. Coordinating Processes with Secure Spaces. *Sci. Comput. Program.*, 46(1-2):163–193, 2003.

[63] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical Capabilities for UNIX. In *Proc. of USENIX SS*, volume 46, 2010.

[64] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. A Taste of Capsicum: Practical Capabilities for UNIX. *Communications of the ACM*, 55(3):97–104, 2012.

[65] George Wells. Interprocess Communication in Java. In *PDPTA*, pages 407–413, 2009.

[66] George C Wells. New and improved: Linda in Java. *Science of Computer Programming*, 59(1):82–96, 2006.

[67] George C Wells, Alan G Chalmers, and Peter G Clayton. Linda Implementations in Java for Concurrent Systems. *CCPE*, 16(10):1005–1022, 2004.

[68] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proc. of SOSP*, pages 230–243, 2001.

[69] Charles P Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P Quigley, Erez Zadok, and Mohammad Nayyer Zubair. Versatility and Unix Semantics in Namespace Unification. *ACM TOS*, 2(1):74–105, 2006.

[70] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proc. of USENIX SS*, pages 17–31, 2002.

[71] XStream Developers. XStream Serialization Library. http://x-stream.github.io/, 2016. accessed 18-September-2016.

[72] Yuanzhong Xu, Alan M. Dunn, Owen S. Hofmann, Michael Z. Lee, Syed Akbar Mehdi, and Emmett Witchel. Application-Defined Decentralized Access Control. In *Proc. of USENIX ATC*, pages 395–408, 2014.

[73] Jia Yu and Rajkumar Buyya. A Novel Architecture for Realizing Grid Workflow using Tuple Spaces. In *Proc. of Intl. Workshop on Grid Computing*, pages 119–128, 2004.

[74] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing Distributed Systems with Information Flow Control. In *Proc. of USENIX NSDI*, pages 293–308, 2008.