Component-Oriented Access Control for Deployment of Application Services in Containerized Environments

Kirill Belyaev(✉) and Indrakshi Ray

Computer Science Department, Colorado State University, Fort Collins, USA
kirill.belyaev@outlook.com, Indrakshi.Ray@colostate.edu

Abstract. With the advancements in multi-core CPU architectures, it is now possible for a server operating system (OS) such as Linux to handle a large number of concurrent application services on a single server instance. Individual service components of such services may run in different isolated environments, such as chrooted jails or application containers, and may need controlled access to system resources and the ability to collaborate and coordinate with each other in a regulated and secure manner. In an earlier work, we motivated the need for an access control framework that is based on the principle of least privilege for formulation, management, and enforcement of policies that allows controlled access to system resources and also permits controlled collaboration and coordination for service components deployed in disjoint containerized environments under a single OS instance. The current work provides a more in-depth treatment of secure inter-component communication in such environments. We show the policies needed for such communication and demonstrate how they can be enforced through a Linux Policy Machine that acts as the centralized reference monitor. The inter-component interaction occurs through the persistent layer using a tuple space abstraction. We implemented a tuple space library that provides operations on the tuple space. We present preliminary experimental results of its implementation that discuss the resource usage and performance.

Keywords: Access control · Data and application security · Denial of service protection · Distributed systems security · Security architectures

1 Introduction

The advancements in contemporary multi-core CPU architectures have greatly improved the ability of modern server operating systems (OS) such as Linux to deploy a large number of concurrent application services on a single server instance. The emergence of application containers [7], introduction of support

This work was supported by a grant from NIST under award no. 70NANB15H264, 60NANB16D249 and 60NANB16D250.

for kernel namespaces [11] allows a set of loosely coupled service components to be executed in isolation from each other and also from the main operating system. This enables application service providers to lower their total cost of ownership by deploying large numbers of application services on a single server instance and possibly minimize horizontal scaling of applications across multiple nodes. Executing the individual service components in isolated containers has its benefits. If a single containerized application runtime is compromised by the attacker, the attack surface is limited in its scope to a single component. This theoretically limits the possibility of disrupting the entire data service. Moreover, such an approach also simplifies the management and provision of service components.

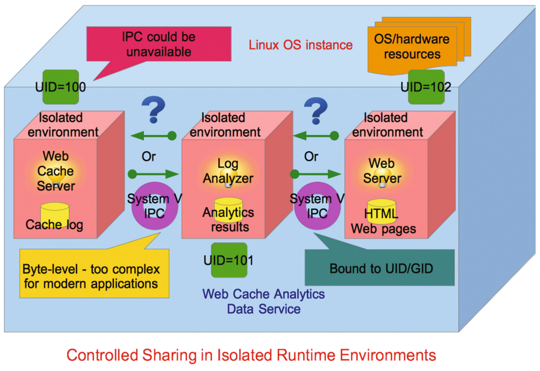

Fig. 1. Problems of controlled sharing

In this model, the individual isolated service components may need to coordinate and collaborate to provide the service and the various service components may not have access to a common centralized database management system or a key-value store for the purpose of communication. Moreover, many services may not rely on database storage in the first place. For instance, consider a real-world service deployment scenario illustrated in Fig. 1. 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. In such a case, there is a need to access and share data objects across the applications in disjoint containerized environments. 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. Usual Inter-Process Communication (IPC) primitives such as message queues, memory-mapped files and shared memory may cause unauthorized access or illegal information flow and therefore could be disabled in targeted deployments [3].

In conventional UNIX or Linux OS, applications can be deployed in isolated (containerized) environments, such as chrooted jails. Such isolated environments limit the access of the applications beyond some designated directory tree and have the potential to offer enhanced security and performance. However, no mechanism is provided for controlled communication and sharing of data objects between isolated applications across such environments. A new type of access control targeting multi-service deployments on contemporary server hardware has been recently introduced through Linux Policy Machine (LPM) [3] framework. The framework proposed such a component-oriented access control for isolated service components that controls access to OS resources, and regulates the inter-component communication under a single service. LPM is a user-space reference monitor that allows the formulation and enforcement of component-oriented policies.

The regulated communication between isolated service components relies on the access control that is based on the adaptation of generative communication paradigm introduced by Linda programming model [8] which uses the concept of tuple spaces for process communication. However, the traditional tuple spaces lack any security features and also have operational limitations [3]. Most implementations are limited to tuple space communication within a single memory address space of an application and do not offer simple ways to interact between separate component processes with independent runtime environments. Moreover, main memory based solutions 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. In our current work we enhance the original paradigm and provide the initial experimental results of the developed Tuple Space Library (TSL) that relies on the persistent storage approach [5] and provides personal tuple space per service component for security reasons. The communication between applications is mediated through a Tuple Space Controller (TSC) – the component of the LPM reference monitor which is allowed limited access to an application's tuple space. The component-oriented access control allows a regulated way of coordinating and collaborating among components of a single service through tuple spaces. We also present the formal model for expressing component-oriented policies.

The rest of the paper is organized as follows. Section 2 provides the overview of component-oriented access control targeting isolated runtime environments. Section 3 describes the architecture for inter-component communication. Section 4 demonstrates the preliminary experimental results for the developed TSL library. Section 5 covers some related works. Section 6 concludes the paper.

2 Communicative Access Control

In order to address the requirements of the regulated communication between isolated service components, we introduce the notion of a *communicative policy 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. Our notion of communicative policy 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 policy class via object replication. Such a sharing can be very fine-grained and it may be *unidirectional* – an isolated application can request a replica of data object belonging to another isolated application but not the other way around.

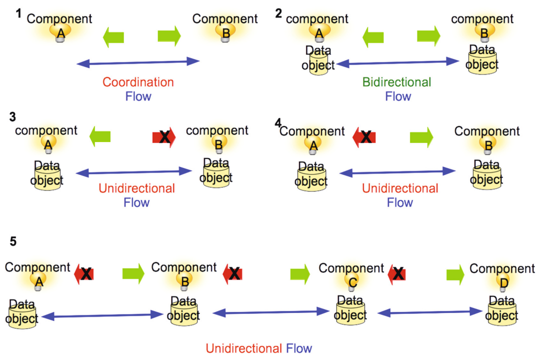

Fig. 2. Flow control of isolated service components (Color figure online)

Some applications may require *bidirectional* access requests where both applications can request replicas of respective data objects. Such types of possible information flow are depicted in Fig. 2 where green arrow denotes the granted request for a replicated data object in the direction of an arrow, while red one with a cross 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 policy class specify how the individual applications in such a class can request a replica of mutual data objects. Only applications within the same communicative class can communicate and therefore communication across different communicative policy classes is forbidden. Such a regulation is well-suited for multiple data services hosted on a single server instance. The assignment of individual data service to a separate policy class facilitates the fine-grained specification of communication policies between various isolated service components.

The construct of communicative policy class is designed to support the following communication patterns among the applications in a single class. (i) *coordination* – often applications 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 [3]. Coordination across mutually isolated environments is problematic. However, if applications belong to a single communicative policy class, it enables the exchange of coordination messages without reliance on usual UNIX IPC mechanisms that may be unavailable under security constrained conditions. (ii) *collaboration* – components acting as a single data 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 an application with the ability to obtain a replica of a data object that belongs to another application in the same communicative policy class makes such collaboration possible.

Based on the described communication patterns between service components, a single communicative policy class can be classified as a *coordinative policy class* if it contains a set of coordination policies. Consequently, it can also be classified as a *collaborative policy class* if it contains a set of collaboration policies.

3 Communications Architecture

We now discuss the enforcement architecture for communicative policy class model.

3.1 IPC Constraints

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 [12]. 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 data service components also prefer information-oriented communication at the level of objects [4]. The necessity of adequate authentication primitives to prove application identity may also be non-trivial. Moreover, as illustrated in Sect. 2, 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 collaboration or coordination, which violates our principle of least privilege [3].

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-application exchange on a single server instance that hosts a large number of isolated application services. In other words, IPC may be disabled on the level of OS kernel. 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 UID/GID access control associations that does not provide fine-grained control at the level of individual applications. Therefore kernel-space IPC mechanisms do not offer regulated way of inter-application interaction. The usage of system-wide user-space IPC frameworks such as D-Bus [9] may also be problematic due to security reasons and absence of flexible access control mechanisms despite its ability to transport arbitrary byte strings (but not file objects) [3].

3.2 Tuple Space Paradigm

In order to address the complexities introduced in Sect. 3.1, we applied an alternative approach that can be classified as a special case of *generative communication* paradigm introduced by Linda programming model [8]. 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 [12]. 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 [15]. However, the lack of any protection mechanism in the basic model [12,15] makes the single global shared tuple space unsuitable for interaction and coordination among untrusted components. The traditional in-memory implementation of tuple space makes it unsuitable in our current work due to security issues and memory utilization overheads [3]. 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. That makes such a construct problematic for use between independent processes [3].

We implemented a tuple space calculus that is compliant with the base model introduced in [8] but is applied on dedicated tuple spaces of individual applications instead of a global space. Our *tuple space calculus* comprises the following operations: (i) *create tuple space* operation, (ii) *delete tuple space* operation – deletes tuple space only if it is empty, (iii) *read* operation – returns the value of individual tuple without affecting the contents of a tuple space, (iv) *append* operation – adds a tuple without affecting existing tuples in a tuple space, and (v) *take* operation – returns a tuple while removing it from a tuple space. We adhere to the *immutability* property – tuples are immutable and applications can either append or remove tuples in a tuple space without changing contents of individual tuples.

An application 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.

Tuple space is implemented as an abstraction in the form of a filesystem directory with its calculus performed via TSL employed by the applications and the LPM. Therefore, this part of the unified framework is not transparent and the applications 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 applications require only limited collaboration, such as periodic requests for replicas of data objects (the case for daily logs), a separate data requester application that employs TSL can handle such a task without the need to modify the existing application such as a log analyzer.

The LPM plays a mediating role in the communication between applications. 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 applications (service components). The LPM periodically checks for control tuples in the tuple spaces for applications registered in its database. 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.

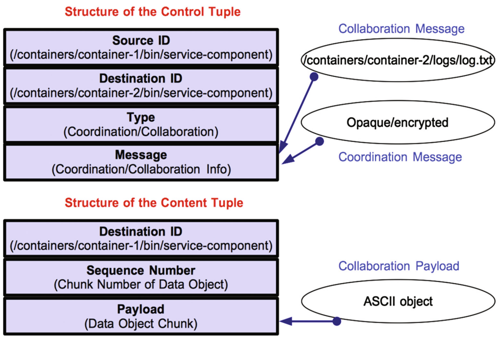
Structure of the Control Tuple
Source ID (/containers/container-1/bin/service-component)
Destination ID (/containers/container-2/bin/service-component)
Type (Coordination/Collaboration)
Message (Coordination/Collaboration Info)
Collaboration Message
/containers/container-2/logs/log.txt
Opaque/encrypted
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

Fig. 3. Tuples structure

The structure of the tuples is shown in Fig. 3. Control tuples are placed by an application into its tuple space for the purpose of coordination or for requesting data from other applications. A control tuple has the following fields: (i) *Source ID* – indicates an absolute path of the application that acts as an application ID of the communication requester. (ii) *Destination ID* – indicates an absolute path of the application that acts as an application ID of the communication recipient. (iii) *Type* – indicates whether it is a collaborative or coordinative communication. (iv) *Message* – 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-application communication. It may even be encrypted to ensure the security and privacy of inter-application coordination efforts. XML or JSON are possible formats that can be used for the representation of coordination messages. LPM merely shuttles the coordination tuples between respective applications' spaces and is not aware of their semantics. Content tuples are used for sharing data objects across applications and they have the following fields: (i) *Destination ID* – indicates the ID of recipient application that is an absolute path of an application. (ii) *Sequence Number* – 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-application collaboration. (iii) *Payload* – contains the chunk of a data object. Content tuples are placed by the LPM reference monitor into corresponding tuple space of the requesting application that needs to receive content. Note that content tuples are designed for collaboration only. Coordination is performed exclusively through control tuples.

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 filesystem 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.

3.3 Tuple Space Transactions

We provide the sample transactional flow involved in tuple space operations, necessary to carry out collaborative and coordinative types of communication between isolated service components. Since loosely coupled processes can not communicate directly due to isolation properties, the flow is conducted indirectly via the TSC.

Coordinative Transaction. Coordinative communication between two components is depicted in Fig. 4. 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 Fig. 3) 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*.

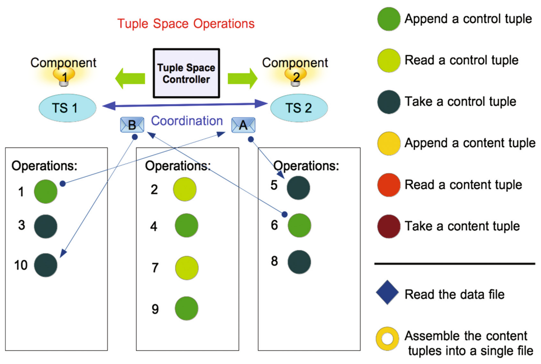

Fig. 4. Coordination through tuple spaces

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 information, 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 is XML, JSON or text strings. Such a choice is service-dependent. Moreover, the service components could utilize the serialization libraries such as XStream [16], 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.

Collaborative Transaction. Collaborative communication is depicted in Fig. 5. Intrinsically, collaboration is unidirectional, since the workflow is only

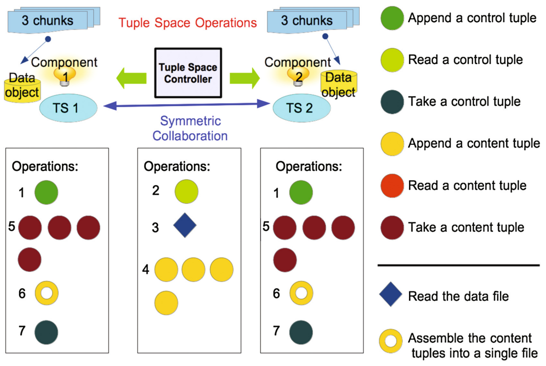

Fig. 5. Collaboration through tuple spaces

directed from a single requester to TSC and back in the form of content tuples. In contrast to a control tuple, a content tuple only has a Destination ID field, as depicted in Fig. 3. 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 Fig. 5. 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 filesystem. 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 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 creation of a tuple space and ends with its deletion after the transactional flow completes. The complexity for both types of communication is hidden from applications. TSL provides public Application Programming Interface (API) methods without exposing internal operations of tuple space calculus.

3.4 Security Aspects

Tuple space communication addresses the confidentiality, integrity, and availability issues with respect to tuple space implementation. Only members of the same communicative policy class can coordinate and/or share data. Extra protection mechanisms are also incorporated for each application's tuple space. Each application (service component) creates its own tuple space in the directory structure of the filesystem allocated to its isolated runtime environment. Only the individual application can perform all the operations, namely, *create tuple space*, *delete tuple space*, *read*, *append*, and *take*. TSC can only perform *read* and *append* operations on the tuple space. Thus, no one other than the application itself can remove anything from its tuple space. Moreover, the confidentiality and integrity are guaranteed by virtue of isolation from other services deployed on the node. Note that, from the confidentiality standpoint a malicious application cannot request a replica of a data object that belongs to another application deployed in a separate isolated runtime environment unless it is registered in the policy database containing communicative policies classes of the LPM and has the appropriate policy records. Removing it from the associated communicative policy class will disable the collaboration with other service components [3]. Unrestricted inter-application communication is avoided through the notion of *trust* between applications that is implicit for components of a single data service. Such components should be logically placed in the same communicative policy class as indicated in Sect. 2.

Applications may misbehave and cause Denial-of-Service (DOS) attacks by exhausting system resources. Our TSL facilitates the data and control flow that prevents an application from using all the allocated filesystem space in the directory structure of the isolated environment. The implementation of append operation for collaboration ensures that such an operation writes only a single content tuple at a given time and the application has to take the tuple before a new one is written in its tuple space. Such a strategy avoids overconsumption of filesystem space, alleviates disk/filesystem access loads with large numbers of concurrent transactions, and also serves as an acknowledgement mechanism before the next chunk of the replicated data object is written.

4 Experimental Results

The initial prototype of the TSL implemented in Java SE is publicly available through the LPM's GitHub repository [2]. The specification of the machine involved in the benchmarking is depicted in Table 1. Memory utilization and time information has been obtained using JVM's internal Runtime and System packages. Due to space limitations, 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 actual unit test for coordination is available at: https://github.com/kirillbelyaev/tinypm/blob/LPM/src/test/java/TSLib_UnitTests_Coordination.java.

Table 1. 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
Filesystem	EXT4-fs; Block size: 4096 bytes; Size: 53 GB; 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 Sect. 3, is performed on the EXT4 filesystem, 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 Fig. 6. The create_ObjectReplica() method in Utilities package of the TSL library is a reference method that sequentially 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 s to replicate a 64 MB object, 1.065 s a 128 MB object, 1.955 s a 256 MB object, 3.950 s a 512 MB object, 8.550 s a 1024 MB object and 17.505 s 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 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

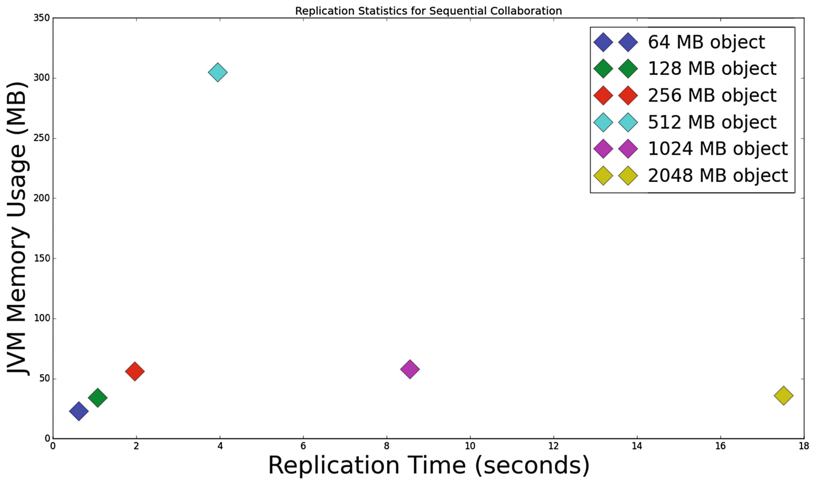

Fig. 6. Replication performance for sequential collaboration

actual JVM utilization will be roughly twice lower for two endpoints in the collaborative transaction when endpoints 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/LPM/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 Fig. 7, 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 Fig. 6. 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) [5]. Again, we can observe that the replication time progressively doubles with an increase of the object size. On average, it takes 17.152 s to replicate a 64 MB object, 23.8 s a 128 MB object, 37.1 s a 256 MB object, 63.8 s a 512 MB object, 117.5 s a 1024 MB object and 246.505 s to replicate a 2048 MB object. In line with sequential collaboration, JVM memory utilization during concurrent

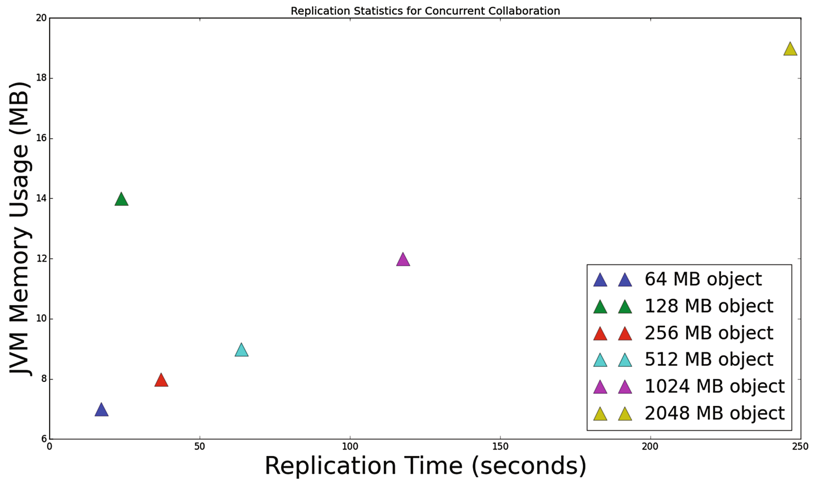

Fig. 7. Replication performance for concurrent collaboration

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, 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/LPM/src/test/java/TSLib_UnitTests_Collaboration.java.

5 Related Work

For the complete coverage of relevant research efforts, we direct the interested reader to our original work on the subject [3]. Application-defined decentralized access control (DCAC) for Linux has been recently proposed by Xu et al. [17] 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 [17]. 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.

The application-level access control is emphasized in Decentralized Information Flow Control (DIFC) [13]. 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 [10,14]. 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 [14]. However, 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 [14]. 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.

In the mobile devices environment, Android Intents [6] offers message passing infrastructure for sandboxed applications; this is similar in objectives to our tuple space communication paradigm 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. 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 [6]. We eliminate such a possibility in our communication architecture due to the virtue of tuple space communication that offers connectionless inter-application communication as discussed in Sect. 3. 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 filesystem. 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.

We have adapted the original Linda model to serve the requirements of inter-component communication. As covered in Sect. 3, the original paradigm has a number of resource-oriented limitations and does not offer security guarantees. For that matter, many researchers [4,12,15,18] have conducted adaptation of the original tuple space model to fit the domain-specific requirements. The LightTS tuple space framework [1] has adapted the original operations on Linda tuple space for use in context-aware applications. LightTS 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 content matching on a set of tuple fields. To the best of our knowledge, we offer the first persistent tuple space implementation that facilitates the regulated inter-application communication without a need for applications to share a common memory address space [3].

6 Conclusion and Future Work

In this work we implemented the communication sub-layer necessary for enforcement of inter-component access control policies expressed through the notion of communicative policy class. We have demonstrated how inter-application communication for isolated service components can take place through persistent tuple spaces. The prototype of TSL library demonstrates the feasibility of our approach. A lot of work remains to be done. A carrier-grade TSC, that is necessary for the enforcement of communicative policy class model needs to be developed for inclusion into our LPM reference monitor. Therefore, we plan to measure system utilization and performance degradation with a large number of concurrent tuple space transactions. The Parser and Persistence layers of LPM also have to be extended to support formulation and storage of policies for communicative class model.

References

1. Balzarotti, D., Costa, P., Picco, G.P.: The LighTS tuple space framework and its customization for context-aware applications. WAIS **5**(2), 215–231 (2007)
2. Belyaev, K.: Linux Policy Machine (LPM) - Managing the Application-Level OS Resource Control in the Linux Environment (2016). https://github.com/kirillbelyaev/tinypm/tree/LPM. Accessed 18 Sep 2016
3. Belyaev, K., Ray, I.: Towards access control for isolated applications. In: Proceedings of SECRYPT, pp. 171–182. SCITEPRESS (2016)
4. Cabri, G., Leonardi, L., Zambonelli, F.: XML dataspaces for mobile agent coordination. In: Proceedings of ACM SAC, pp. 181–188. ACM (2000)
5. Chen, X., Sha, E.H.-M., Zhuge, Q., Jiang, W., Chen, J., Chen, J., Xu, J.: A unified framework for designing high performance in-memory and hybrid memory file systems. JSA **68**, 51–64 (2016)

6. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proceedings of ACM MobiSys, pp. 239–252. ACM (2011)
7. Docker Developers. What is Docker? (2016). https://www.docker.com/what-docker/. Accessed 18 Sep 2016
8. Gelernter, D.: Generative communication in Linda. ACM TOPLAS **7**(1), 80–112 (1985)
9. Havoc Pennington Red Hat, Inc.: D-Bus Specification (2016). https://dbus.freedesktop.org/doc/dbus-specification.html. Accessed 18 Sep 2016
10. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. ACM SIGOPS OSR **41**(6), 321–334 (2007)
11. Linux Programmer's Manual. Kernel Namespaces (2016). http://man7.org/linux/man-pages/man7/namespaces.7.html. Accessed 18 Sep 2016
12. Minsky, N.H., Minsky, Y.M., Ungureanu, V.: Making tuple spaces safe for heterogeneous distributed systems. In: Proceedings of ACM SAC, pp. 218–226 (2000)
13. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM TOSEM **9**(4), 410–442 (2000)
14. Roy, I., Porter, D.E., Bond, M.D., Mckinley, K.S., Witchel, E.: Laminar: practical fine-grained decentralized information flow control. ACM SIGPLAN Not. **44**(6), 63–74 (2009)
15. Vitek, J., Bryce, C., Oriol, M.: Coordinating processes with secure spaces. Sci. Comput. Program. **46**(1), 163–193 (2003)
16. XStream Developers. XStream Serialization Library (2016). http://x-stream.github.io/. Accessed 18 Sep 2016
17. Xu, Y., Dunn, A.M., Hofmann, O.S., Lee, M.Z., Mehdi, S.A., Witchel, E.: Application-defined decentralized access control. In: Proceedings of USENIX ATC, pp. 395–408 (2014)
18. Yu, J., Buyya, R.: A novel architecture for realizing grid workflow using tuple spaces. In: Proceedings of International Workshop on Grid Computing, pp. 119–128. IEEE (2004)