ARTICLE IN PRESS

Future Generation Computer Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Component-oriented access control—Application servers meet tuple spaces for the masses

Kirill Belyaev *, Indrakshi Ray

Department of Computer Science, Colorado State University, Fort Collins, CO, USA

HIGHLIGHTS

- Comprehensive details of the component-oriented access control framework.
- Deployment of application services in isolated environments on a single multi-core server instance.
- Deployment on UNIX based OS in the cloud datacenters.
- User-space reference monitor and its performance evaluation.

ARTICLE INFO

Article history:
Received 15 November 2016
Received in revised form
13 April 2017
Accepted 5 May 2017
Available online xxxx

Keywords:
Access control
Service and systems design
Tuple spaces
Data and application security
Denial of service protection
Security architectures

ABSTRACT

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 application services on a single server instance. Individual application components of such services may run in different isolated runtime environments, such as chrooted jails or application containers, and may need access to system resources and the ability to collaborate and coordinate with each other in a regulated and secure manner. We implemented an access control framework for policy formulation, management, and enforcement that allows access to OS resources and also permits controlled collaboration and coordination for service components running in disjoint containerized environments under 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 Linux Policy Machine (LPM) that acts as the centralized reference monitor and provides a uniform interface for accessing system resources and requesting application data and control objects. We present the details of our framework and also discuss the preliminary implementation to demonstrate the feasibility of our approach.

1. Introduction

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 application services on a single server instance. Such deployments become increasingly common as large data centers and cloud-centric application services become more popular. New developments, such as Internet of Things (IoT), may contain a set of smart IoT devices which are interconnected and controlled through software services using cloud infrastructure.

The emergence of application containers [1,2], introduction of support for kernel namespaces [3], allows a set of loosely coupled service components to be executed in isolation from each other and also from the main operating system. Application 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 [4]. In conventional UNIX or Linux OS, applications can be deployed in isolated (containerized) environments, such as chrooted jails [5,6]. 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.

Our objective is to give each component minimum privileges to system resources and also provide regulated coordination and

* Corresponding author.
E-mail addresses: kirill.belyaev@outlook.com (K. Belyaev), Indrakshi.Ray@colostate.edu (I. Ray).

http://dx.doi.org/10.1016/j.future.2017.05.003
0167-739X/

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

data sharing across service components executing in isolated runtime environments. We introduce a new framework referred to as *Linux Policy Machine* [7] middleware (and shorten it to just LPM in the rest of the paper) to address the identified challenges. Our LPM allows the management and enforcement of access to OS resources for individual service components and it also allows regulated inter-component communication. Our uniform framework provides a coherent business logic interface available to administrative personnel to manage access control for a set of application services both at the level of OS resources and at the level of inter-component interaction.

Our policy formulation and enforcement framework consists of two types of policy classes. We propose the notion of *capabilities classes*, each of which is associated with a set of Linux capabilities [8]. 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 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.

We also propose the concept of *communicative classes* for communication of components that belong to different isolated environments. We adapt the generative communication paradigm introduced by Linda programming model [9] which uses the concept of tuple spaces for process communication. However, the traditional tuple spaces lack any security features and also have operational limitations [7,10]. We enhance the original paradigm and also provide rules such that only components belonging to the same communicative class can communicate using this approach. The communication between service components is mediated through a module of the LPM called a Tuple Space Controller (TSC) which is allowed limited access to a component's tuple space. Our LPM middleware allows a regulated way of coordinating and collaborating among components using tuple spaces. Note that, such coordination and collaboration will be allowed even if each of the components executes under different system UIDs and group identifiers (GIDs).

Our LPM is resident in user-space and it acts as a reference monitor [11] for a set of administered application services deployed on a single Linux server instance. We incorporate the access control modeling and decision control in user-space with robust and expressive persistence layer [7]. This allows high interoperability and usage of the framework on any general-purpose Linux OS without a requirement for custom kernel patching [11]. The current work extends our earlier works and provides additional details on the inter-component communication architecture realized through the concept of Tuple Space Library (TSL) [10] and also describes the implementation aspects.

The rest of the paper is organized as follows. Section 2 gives an overview of our framework that consists of two types of policy classes. Section 3 provides the details of the inter-component communication architecture. Section 4 provides the details of tuple space transactions that provides secure coordination and collaboration between service components. Section 5 provides details on security of our framework. Section 6 gives system architecture. Section 7 demonstrates the feasibility of our approach by describing the implementation experience with a focus on tuple space paradigm. Section 8 provides the benefits and drawbacks of user-space solution in contrast to kernel-space implementation. Section 9 discusses related work. Section 10 concludes the paper.

2. Component-oriented access control framework

We first present a real-world motivating example where a single application/data service can consist of several components, each deployed in isolation [10,7].

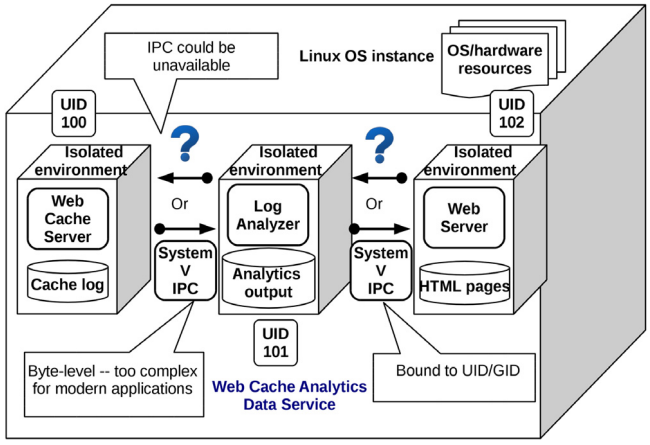

Fig. 1. Problems of controlled sharing.

2.1. Motivating example

Consider a service deployment scenario illustrated in Fig. 1 that is taken from a real-world telecom service provider [12]. 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 [13] 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 [7]. We now provide a brief overview of our access control framework based on the concept of such policy classes.

2.2. Capabilities class

The individual containerized components of an application 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 [14]. 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

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

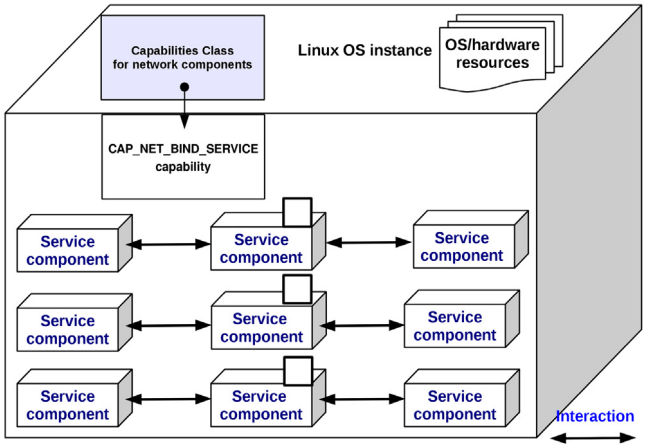

Fig. 2. Capabilities class for components accessing the network.

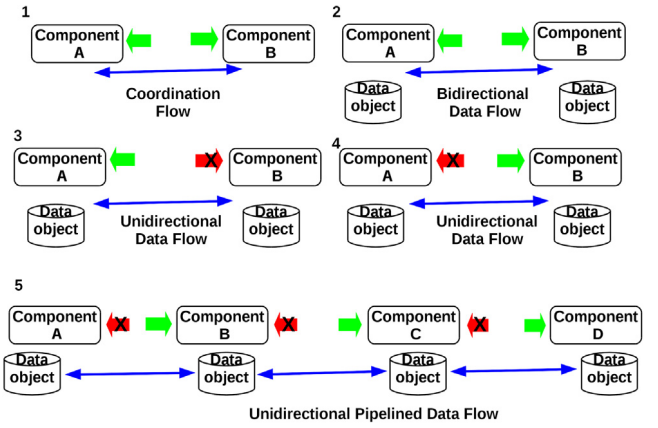

Fig. 3. Various types of flow control for service components in isolated environments.

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 [2] and CoreOS [15] heavily leverages the Linux capabilities model. Despite the incorporation of capabilities in mainstream Linux and application containers, capabilities management in user-space is challenging [16] and is only sufficiently addressed in our work [7].

In service provider context, the only capabilities, realistically required by the containerized data and application services deployed on the same OS server instance are the capabilities associated with access to 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. Therefore, individual service components could be conferred with specific capability(s) based on the principle of least privilege.

We introduced 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 Fig. 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. As a part of our unified framework, capabilities class supports a set of management operations on it [7].

2.3. Communicative class

In order to address the requirements of the regulated communication between isolated service components, we introduced 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 [7]. 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 in Fig. 3 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 class specify how the individual components in such a class can request a replica of mutual data objects. 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 data services hosted on a single server instance. The assignment of individual data service to a separate class facilitates the fine-grained specification of communication policies between various isolated service components [7,10].

The construct of communicative class is designed to support the following communication patterns between the components in a single class. (i) *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 [7]. 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. (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 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.

Based on the described communication patterns between service components, a single communicative class can be classified as a *coordinative class* if it contains a set of coordination policies. Consequently, it can also be classified as a *collaborative class* if it contains a set of collaboration policies. As a part of our unified framework, communicative class supports a set of management operations on it [7].

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

3. Communication architecture

As discussed in Section 2, 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 discuss the issues with available OS communication mechanisms and then provide details on our enforcement architecture for communicative class model that is one of the main contributions of this work.

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 [17]. 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 data service components also prefer information-oriented communication at the level of objects [18]. The necessity of adequate authentication primitives to prove application identity may also be non-trivial. Moreover, as illustrated in Section 2.3, 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 [7].

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 [4]. In other words, IPC may be disabled on the level of OS kernel [19]. 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 UID/GID access control associations that does not provide fine-grained control at the level of individual applications [19]. Therefore, kernel-space IPC mechanisms do not offer regulated way of inter-application interaction [7].

The usage of system-wide user-space IPC frameworks such as D-Bus [20] 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 data services. In fact, the main design objective of D-Bus is not message passing but rather process lifecycle tracking and service discovery [20]. 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 [7].

3.2. Tuple space paradigm

In order to address the complexities introduced in 3.1, we proposed an alternative approach that can be classified as a special case of *generative communication* paradigm introduced by Linda programming model [9]. 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 [17]. 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 [21]. In comparison to general-purpose message-passing that provides a rather low-level programming abstraction for building distributed systems and enabling inter-application interaction, Linda, instead, provides a simple coordination model with higher level of abstraction that makes it very intuitive and easy to use [22].

3.2.1. Paradigm limitations

The lack of any protection mechanism in the basic model [21,17] 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 belongs to a large set of divergent applications in a tuple space increases, there is an increasing chance of accidental matching of a tuple that was requested by another application. Moreover, the traditional in-memory implementation of tuple space, oriented at language-level interaction [23], makes it unsuitable in our current work due to a wide array of possible security attacks and memory utilization overheads. 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 [7]. Note that, in this adaptation the content-based nature of retrieval from a tuple space will necessitate content-based access control approaches [17].

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 [24]. 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 data service applications that do not share the same address space in memory which makes such a solution unsuitable [18]. 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 [25]. The discussed limitations are depicted in Fig. 4.

3.2.2. Paradigm adaptation

We proposed a tuple space calculus that is compliant with the originally introduced base model [9] 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

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

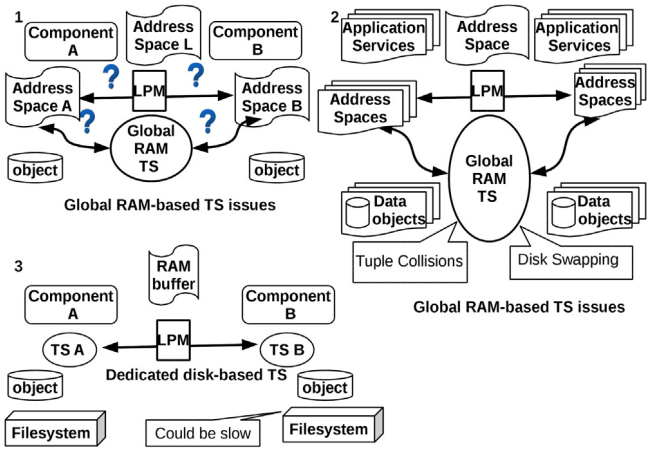

Fig. 4. Tuple space limitations.

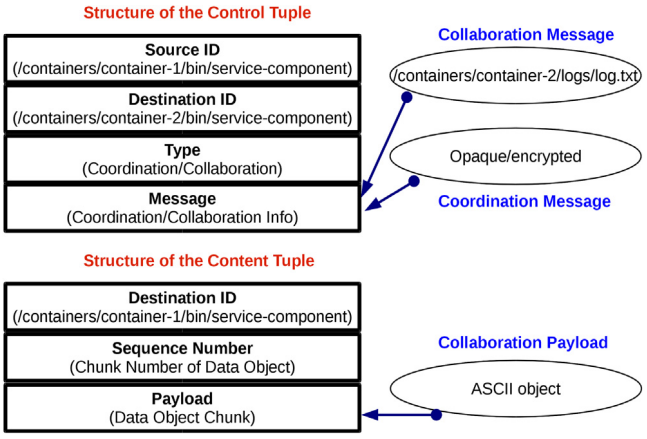

Fig. 5. Tuples structure.

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 [7].

Tuple space is implemented as an abstraction in the form of a filesystem directory with its calculus performed via Tuple Space Library (TSL) employed by the applications and the LPM reference monitor through its TSC. Therefore, this part of the proposed 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 components 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 [7].

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. We have two different types of communication between a pair of applications. The first case is where the two applications do not share any data but must communicate with each other in order to coordinate activities or computation. The second case is where an application 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 [7,10].

The structure of the tuples is shown in Fig. 5. 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 [7].

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 [7].

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 [10].

4. 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 cannot communicate directly due to isolation properties, the flow is conducted indirectly via the Tuple Space Controller (TSC) [10].

4.1. Coordinative transaction

Coordinative communication between two components is depicted in Fig. 6. Intrinsically, coordination is bidirectional,

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

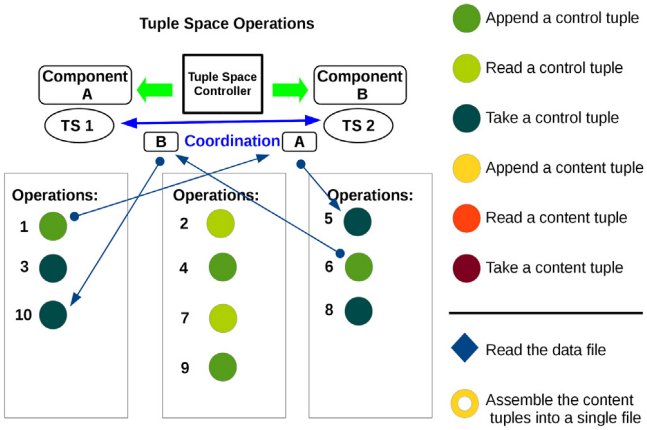

Fig. 6. Coordination through tuple spaces.

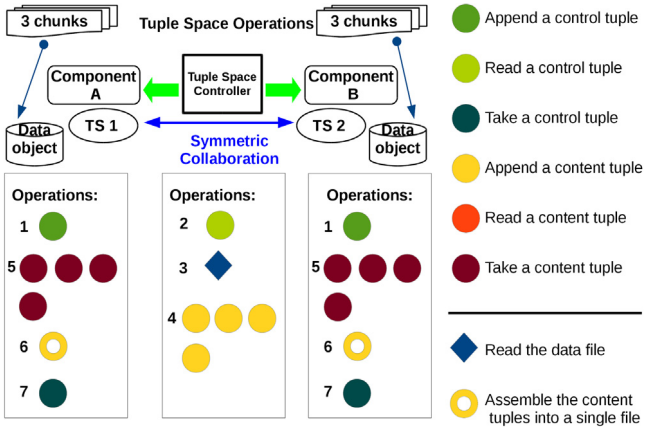

Fig. 7. Collaboration through tuple spaces.

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. 5) 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 information, or serve as the acknowledgment 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 [26], 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 [10].

4.2. Collaborative transaction

Collaborative communication is depicted in Fig. 7. Intrinsically, collaboration is unidirectional, since the workflow is only directed from a single requester to TSC and back in the form of content tuples [10]. In contrast to a control tuple, a content tuple only has a Destination ID field, as depicted in Fig. 5. 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. 7. 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 the creation of a tuple space and ends with its deletion after the transactional flow completes.

4.3. Transactional API

The complexity for both types of transactional communication is hidden from applications that want to use it. TSL provides public

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

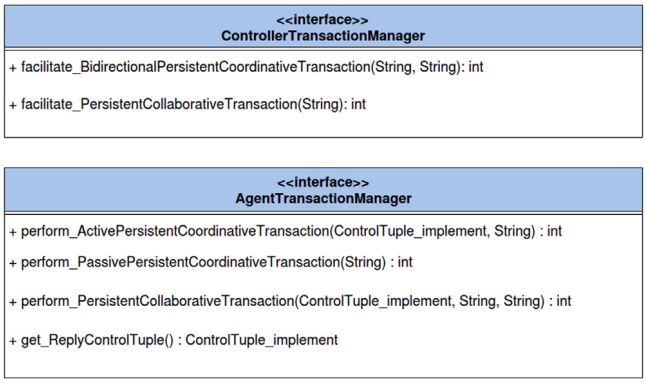

Fig. 8. Tuple space transactions API.

Application Programming Interface (API) methods without exposing internal operations of tuple space calculus [27]. The API methods for Tuple Space Transactions are depicted in Fig. 8. TSC executes the implementation of the ControllerTransactionManager class while the service component executes the implementation of the AgentTransactionManager class within the TSL library.

ControllerTransactionManager implementation implements the following public methods:

- *facilitate_BidirectionalPersistentCoordinativeTransaction()*—facilitates the exchange of control tuples between corresponding tuple spaces of service components. The actual implementation of this method uses the private *facilitate_UnidirectionalPersistentCoordinativeTransaction()* method to append control tuples to individual tuple spaces involved in coordination
- *facilitate_PersistentCollaborativeTransaction()*—facilitates the replication of a data object requested in the collaborative request issued by the component.

AgentTransactionManager implementation implements the following public methods:

- *perform_ActivePersistentCoordinativeTransaction()*—initiates a start of coordinative transaction by appending the initial control tuple
- *perform_PassivePersistentCoordinativeTransaction()*—initiates an end of coordinative transaction with a wait for a control tuple from original destination
- *perform_PersistentCollaborativeTransaction()*—initiates and completes the collaborative transaction by assembling the replica at component's end
- *get_ReplyControlTuple()*—obtains the control tuple appended by TSC from the participating component.

5. Security aspects

In this section, we discuss the security of our component-oriented access control framework. Each component executes in an isolated runtime environment. 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 [7].

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 is 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.

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 filesystem 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.

A malicious entity cannot impersonate as an honest component and compromise the confidentiality and integrity of the data of any components. 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.

6. System architecture

LPM acts as a centralized enforcement point and reference monitor [7,11] for the application services deployed on the single OS server instance. The unified framework uses the embedded SQLite 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 and application containers. Fig. 9 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 the user input from the CLI shell and forwards the parsed input to the underlying layers for execution.
- [**Enforcer layer**:] This layer enforces the capabilities on the given application using Linux LibCap [8] library and grants/denies access to OS resources depending on the capabilities class associated with the component. The layer also integrates a TSC [17] responsible for tuple space operations for the enforcement of collaboration and coordination of service components in a single communicative class.
- [**Persistence layer**:] This layer provides the Create/Read/Update/Delete (CRUD) functionality to manage records using embedded database facilities. The schema of the embedded database for storing framework policies appears in Fig. 10. Due to considerable complexity, the detailed description of the schema, the business logic, the relationship between three main tables, the possible access control records and formal model is subject to a separate publication.

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

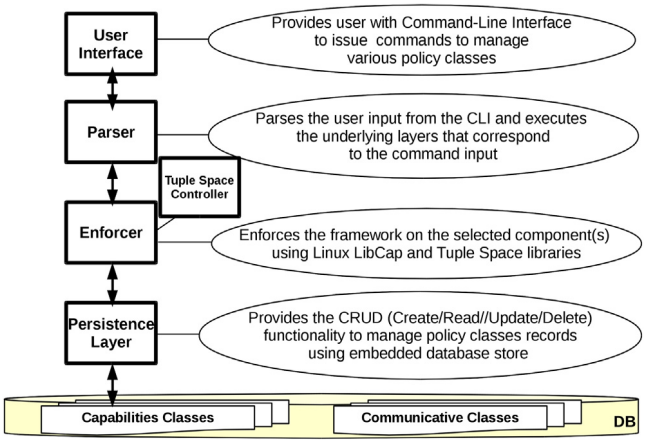

Fig. 9. Linux policy machine (LPM) architecture.

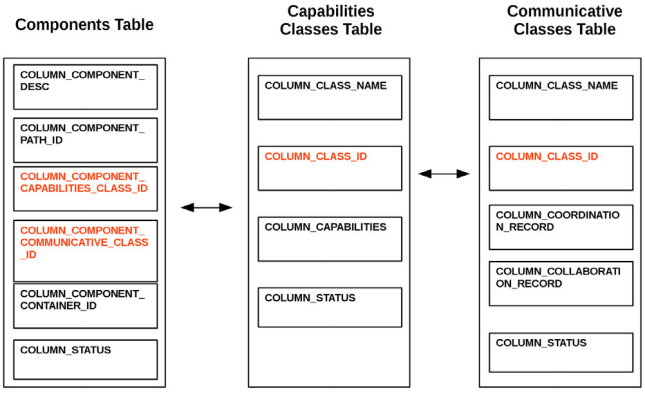

Fig. 10. Database schema of persistence layer.

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 [27].

7. 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 [8] that essentially updates the filesystem capabilities metadata information for a process [28]. 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 [29,30]. 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 [17]. 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 [21]. Those properties essentially pose a limit on a number of tuple objects in a given tuple space [21,17,9]. Taking that into consideration as discussed in Section 3, the design of our tuple space implementation is

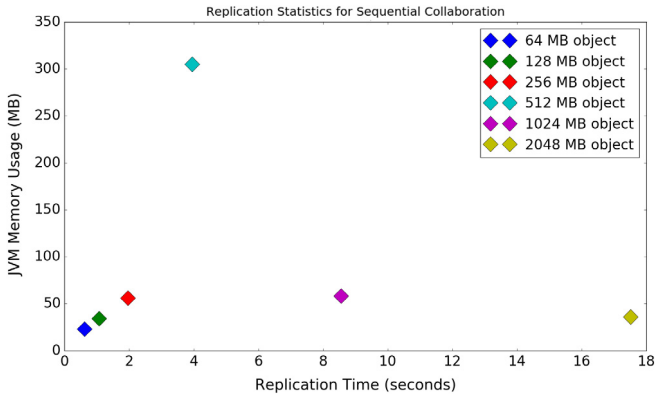

Fig. 11. Replication performance for sequential collaboration.

reliant on the alternative strategy of the persistent filesystem-based solution with personal tuple space per component [10].

The initial prototype of the TSL implemented in Java SE is publicly available through the LPM's GitHub repository [27]. 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/LPM2/src/test/java/TSLib_UnitTests_Coordination.java.

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 Section 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. 11. 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 [31] 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 execute in separate

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

Table 1
Node specifications.

Attribute	Info
CPU	Intel(R) Xeon (R) X3450 @ 2.67 GHz; Cores: 8
Disk	SATA: 3 Gb/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

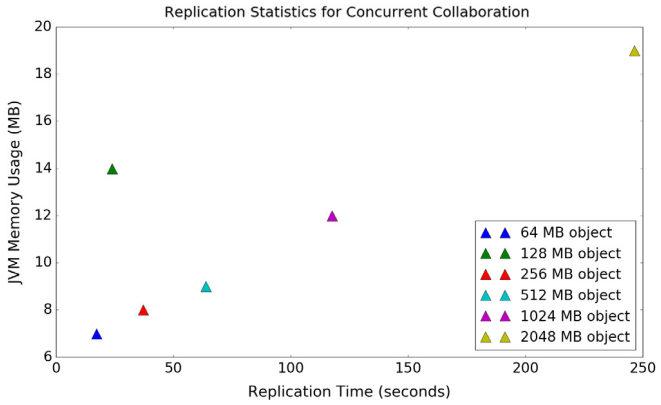

Fig. 12. Replication performance for concurrent collaboration.

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 Fig. 12, 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. 11. 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) [32]. 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 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 Fig. 11 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. In the future, we may consider to conduct explicit comparison of replication speeds between our solution and basic copy operation using cp command in user space.

7.1. 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 2.

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. 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

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

Table 2
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
Filesystem	EXT4-fs; Block size: 4096 bytes; Size: 40 GB; Use: 1%
RAM	24 GB
OS	Fedora 24, Linux kernel 4.7.5-200
Java VM	OpenJDK 64-Bit Server SE 8.0_102

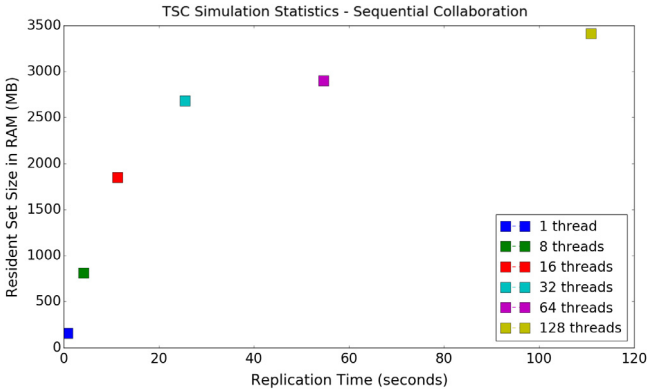

Fig. 13. Tuple space controller simulation information for a 64 MB Object.

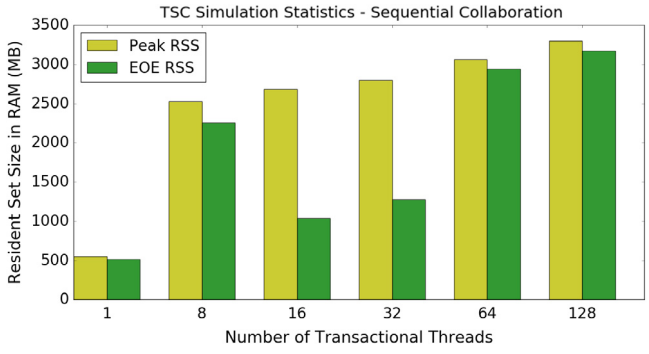

Fig. 14. Tuple space controller simulation information for a 2048 MB object.

focus is on RAM consumption 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 Fig. 13. 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 2. 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 Fig. 14. 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 Figs. 13 and 14. In fact, the RSS is slightly larger for 128 transactions associated with a 64 MB data object. That 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 [31] 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 [33].

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 Fig. 14. 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 filesystem activity. That could significantly increase the completion time of a single collaborative transaction for a set of large data objects.

8. Discussion on user-space LPM

Our framework is supported through the user-space LPM reference monitor which becomes the main differentiator in contrast to most existing works (covered in Section 9) that are mainly based on kernel-space solution such as SELinux [34] and DTE [29]. In contrast to such kernel-level solutions, LPM provides access control facilities that are mainly oriented towards containerized application 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 Section 2) without a need to incorporate direct kernel-level support

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

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 [24]. Second, service components execute in user space and therefore benefit from IPC mechanism implemented through user-level libraries [24] such as our TSL implementation that avoids complexities of message passing that requires additional synchronization primitives [7]. For instance, user-space IPC daemons such as D-Bus [20] (discussed in Section 3) 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 [24,11].

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 [7, 11]. 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 [28] (LSM) hooks or related mediation layers. For instance, LPM already utilizes the capabilities management through calls to user-space LibCap [8] 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. Related work

Traditionally, Linux environments supported 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.

Researchers have proposed Domain and Type Enforcement (DTE) [35,29,30] for Linux and UNIX environments. Type enforcement views a system as a collection of active entities (subjects) and a collection of passive entities (objects) [29,30]. 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, unlike the other Linux approaches, avoids the concept of users and only concentrates on applications [29,30]. Our work, like 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 application 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) [36,37] allows for the specification and enforcement of MAC policies at the kernel level. SELinux uses the Linux Security Modules (LSM) [28] hooks in the kernel to implement its policy. The SELinux architecture is based on the Generalized Framework for Access Control (GFAC) proposed by Abrams [38] 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. SELinux supports the concepts of roles and users but is not intended for enforcing policies at the level of individual applications. 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 [40]. Our work complements the efforts of SELinux in that it provides access control for isolated applications in user-space.

The Rule Set Based Access Control (RSBAC) [41] 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. 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 [42] 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. The set of patches provides protection mechanisms for file systems, executables and networks. 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. Consequently, analysis of the security properties of the various mechanisms is non-trivial.

The application-level access control is emphasized in Decentralized Information Flow Control (DIFC) [43]. 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 [11,44]. 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 [44]. 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

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

DIFC security guarantees are not trivial and require extensive API and domain knowledge [44]. 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.

Application-defined decentralized access control (DCAC) for Linux has been recently proposed by Xu et al. [40] 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 [40]. 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) [45] 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 [45] 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 [45].

In the mobile devices environment Android Intents [46] 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 [47]. 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 [46]. 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 Section 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.

Our work also bears resemblance to the Law-Governed Interactions (LGI) proposed by Minsky et al. [17,48] 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-application 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-application interaction [48,49].

The tuple space model as a type of shared memory, originally introduced by Linda [9] has been widely adapted for parallel programming tasks [50,22], support for language-level coordination [21], multi-agent systems [18,49] and distributed systems [51,17,48] 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 [22]. We have adapted the original Linda model to serve the requirements of secure inter-component communication within a single-node OS with dedicated filesystem-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 Section 3, the original paradigm has a number of resource-oriented limitations [22] and does not offer security guarantees. For that matter, many researchers [17,18,21,52] have conducted adaptation of the original tuple space model to fit the domain-specific requirements. The LighTS tuple space framework [53] 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

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

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-application communication without a need for applications to share a common memory address space or requirements for address space mapping mechanisms [10,24].

10. Conclusion and future work

We have demonstrated how a Linux Policy Machine (LPM) can be developed for the Linux environment that provides access control specification and enforcement for various service components running in isolated environments. LPM may also be utilized in other UNIX-based operating environments. We proposed the notion of policy classes to manage policies pertaining to accessing system and application level resources and demonstrated how regulated inter-component communication can take place through tuple spaces. The initial prototype demonstrates the feasibility of our approach. We plan to extend this work for distributed settings [54–56] where service policies are managed, formulated and updated in a centralized location, and then distributed and enforced at LPM nodes in data center settings.

We also plan to investigate the possibility of adapting the developed framework for use with resource-constrained devices such as IoT nodes. Consider a smart home that has numerous interconnected devices each providing different types of functionalities to enhance the user experience. A smart home contains sensitive personal information about individuals, disclosure of which has catastrophic consequences. Examples include data related to health, finance, utility usage patterns, and physical security. The data belonging to such multiple domains (health, finance, work, home operation) are needed by different stakeholders (health care providers, home security agency, utility companies) and the access control mechanism must be ensured that only authorized users have access to the data they need. Many current applications and also some futuristic ones often span multiple domains. This makes access control extremely challenging. The access control enforcement using tuple space paradigm may be extended for such ubiquitous environments [57].

Acknowledgments

This work was supported, in part, by support from NIST under award nos. 70NANB15H264, 60NANB16D249 and 60NANB16D250 and by support from National Science Foundation (NSF) under award nos. CNS 1619641 and IIP 1540041.

References

[1] Linux Containers Developers. What are Linux Containers? 2016. https://linuxcontainers.org/lxc/introduction/ (accessed 18.09.16).
[2] Docker Developers. What is Docker? 2016. https://www.docker.com/what-docker/ (accessed 18.09.16).
[3] Linux Programmers Manual. Kernel Namespaces, 2016. http://man7.org/linux/man-pages/man7/namespaces.7.html (accessed 18.09.16).
[4] Poul-Henning Kamp, Robert Watson, Building systems to be shared, securely, ACM Queue 2 (5) (2004) 42.
[5] Poul-Henning Kamp, Robert Watson, Jails: Confining the omnipotent root, in: Proc. of SANE, Vol. 43, 2000, p. 116.
[6] Solaris Zones Developers. What are Solaris Zones? 2016. http://docs.oracle.com/cd/E36784_01/html/E36848/docinfo.html (accessed 01.11.16).
[7] Kirill Belyaev, Indrakshi Ray, Towards access control for isolated applications, in: Proc. of SECRYPT, SCITEPRESS, 2016, pp. 171–182.
[8] Linux Programmer's Manual. LIBCAP Manual, 2016. http://man7.org/linux/man-pages/man3/libcap.3.html (accessed 18.09.16).
[9] David Gelernter, Generative communication in Linda, ACM Trans. Program. Lang. Syst. (TOPLAS) 7 (1) (1985) 80–112.
[10] Kirill Belyaev, Indrakshi Ray, Component-oriented access control for deployment of application services in containerized environments, in: Proc. of CANS, in: LNCS, vol. 10052, Springer, 2016, pp. 383–399. volume.
[11] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, Robert Morris, Information flow control for standard OS abstractions, ACM SIGOPS Oper. Syst. Rev. 41 (6) (2007) 321–334.
[12] n–Logic Ltd. n-Logic Web Caching Service Provider, 2016. http://n-logic.weebly.com/ (accessed 18.09.16).
[13] Linux Kernel Developers. Transparent Proxy Support, 2017. https://www.kernel.org/doc/Documentation/networking/tproxy.txt (accessed 10.03.17).
[14] Linux Developers. Linux Programmer's Manual, 2016. http://man7.org/linux/man-pages/man7/capabilities.7.html (accessed 18.09.16).
[15] CoreOS Developers. What is CoreOS? 2016. https://coreos.com/docs/ (accessed 18.09.16).
[16] Serge E. Hallyn, Andrew G. Morgan, Linux capabilities: Making them work, in: Proceedings of OLS, 2008, p. 163.
[17] Naftaly H. Minsky, Yaron M. Minsky, Victoria Ungureanu, Making tuple spaces safe for heterogeneous distributed systems, in: Proc. of ACM SAC, 2000, pp. 218–226.
[18] Giacomo Cabri, Letizia Leonardi, Franco Zambonelli, XML dataspaces for mobile agent coordination, in: Proc. of ACM SAC, ACM, 2000, pp. 181–188.
[19] Michael K. Johnson, Erik W. Troan, Linux Application Development, Addison-Wesley Professional, 2004.
[20] Inc. Havoc Pennington, Red Hat. D-Bus Specification, 2016. https://dbus.freedesktop.org/doc/dbus-specification.html (accessed 18.09.16).
[21] Jan Vitek, Ciarán Bryce, Manuel Oriol, Coordinating processes with secure spaces, Sci. Comput. Programming 46 (1) (2003) 163–193.
[22] Vitaly Buravlev, Rocco De Nicola, Claudio Antares Mezzina, Tuple spaces implementations and their efficiency, in: Coordination, Springer, 2016, pp. 51–66.
[23] George C. Wells, New and improved: Linda in Java, Sci. Comput. Programming 59 (1) (2006) 82–96.
[24] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy, User-level interprocess communication for shared memory multiprocessors, ACM Trans. Comput. Syst. (TOCS) 9 (2) (1991) 175–198.
[25] George Wells, Interprocess communication in Java, in: PDPTA, 2009, pp. 407–413.
[26] XStream Developers. XStream Serialization Library, 2016. http://x-stream.github.io/ (accessed 18.09.16).
[27] Kirill Belyaev, Linux Policy Machine (LPM) – Managing the Application-Level OS Resource Control in the Linux Environment, 2016. https://github.com/kirillbelyaev/tinypm/tree/LPM2 (accessed 18.09.16).
[28] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, Greg Kroah-Hartman, Linux Security Modules: General security support for the Linux Kernel, in: Proceedings of USENIX SS, 2002, pp. 17–31.
[29] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, Sheila Haghighat, Practical domain and type enforcement for UNIX, in: Proceedings of IEEE SSP, IEEE, 1995, pp. 66–77.
[30] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, Sheila A. Haghighat, A domain and type enforcement UNIX prototype, Comput. Syst. 9 (1) (1996) 47–83.
[31] Java NIO Developers. Java Non-blocking I/O Library, 2016. http://en.wikipedia.org/wiki/Non-blocking_I/O_(Java) (accessed 25.10.16).
[32] Xianzhang Chen, Edwin H.-M. Sha, Qingfeng Zhuge, Weiwen Jiang, Junxi Chen, Jun Chen, Jun Xu, A unified framework for designing high performance in-memory and hybrid memory file systems, J. Softw. Appl. 68 (2016) 51–64.
[33] Matt Welsh, David Culler, Eric Brewer, SEDA: An architecture for well-conditioned, scalable Internet services, ACM SIGOPS Oper. Syst. Rev. 35 (5) (2001) 230–243.
[34] SELinux Developers. Security Enhanced Linux, 2016. http://selinuxproject.org (accessed 18.09.16).
[35] Serge Hallyn, Phil Kearns, Domain and type enforcement for Linux, in: Proceedings of ALS, 2000, pp. 247–260.
[36] Peter Loscocco, Integrating flexible support for security policies into the linux operating system. in: Proceedings of USENIX ATC, FREENIX Track, 2001, p. 29.
[37] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, J. Lepreau, The flask security architecture: System support for diverse security policies, in: Proceedings of USENIX SS, 1999.
[38] M.D. Abrams, K.W. Eggers, L.J. LaPadula, I.M. Olson, Generalized framework for access control: An informal description, in: Proceedings of NCSC, 1990.
[39] Leonard LaPadula, Rule-set modeling of trusted computer system, in: M. Abrams, S. Jajodia, H. Podell (Eds.), Information Security: An Integrated Collection of Essays, IEEE Computer Society Press, 1995.
[40] Yuanzhong Xu, Alan M. Dunn, Owen S. Hofmann, Michael Z. Lee, Syed Akbar Mehdi, Emmett Witchel, Application-defined decentralized access control, in: Proc. of USENIX ATC, 2014, pp. 395–408.
[41] Amon Ott, Simone Fischer-Hübner, The rule set based access control (RSBAC) framework for Linux, in: Proceedings of ILK, 2001.
[42] GrSecurity Developers. What is GrSecurity? 2016. https://grsecurity.net (accessed 18.09.16).
[43] Andrew C. Myers, Barbara Liskov, Protecting privacy using the decentralized label model, ACM Trans. Softw. Eng. Methodol. (TOSEM) 9 (4) (2000) 410–442.
[44] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. Mckinley, Emmett Witchel, Laminar: Practical fine-grained decentralized information flow control, ACM SIGPLAN Not. 44 (6) (2009) 63–74.
[45] David Ferraiolo, Serban Gavrila, Wayne Jansen, On the unification of access control and data services, in: Proceedings of IEEE IRI, IEEE, 2014, pp. 450–457.
[46] Erika Chin, Adrienne Porter Felt, Kate Greenwood, David Wagner, Analyzing inter-application communication in android, in: Proc. of ACM MobiSys, ACM, 2011, pp. 239–252.
[47] Alessandro Armando, Roberto Carbone, Gabriele Costa, Alessio Merlo, Android permissions unleashed, in: Proceedings of IEEE CSF, IEEE, 2015, pp. 320–333.

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003

[48] Naftaly H. Minsky, Victoria Ungureanu, Unified support for heterogeneous security policies in distributed systems, in: Proc. of USENIX SS, 1998, pp. 131–142.
[49] Marco Cremonini, Andrea Omicini, Franco Zambonelli, Coordination and access control in open distributed agent systems: The TuCSoN approach, in: Proc. of Coordination, Springer, 2000, pp. 99–114.
[50] George C. Wells, Alan G. Chalmers, Peter G. Clayton, Linda Implementations in Java for concurrent systems, Concurr. Comput.: Pract. Exper. 16 (10) (2004) 1005–1022.
[51] Roberto Lucchi, Gianluigi Zavattaro, WSSecSpaces: a secure data-driven coordination service for web services applications, in: Proceedings of ACM SAC, ACM, 2004, pp. 487–491.
[52] Jia Yu, Rajkumar Buyya, A novel architecture for realizing grid workflow using tuple spaces, in: Proc. of Intl. Workshop on Grid Computing, IEEE, 2004, pp. 119–128.
[53] Davide Balzarotti, Paolo Costa, Gian Pietro Picco, The LighTS tuple space framework and its customization for context-aware applications, WAIS 5 (2) (2007) 215–231.
[54] Kirill Belyaev, Indrakshi Ray, Towards efficient dissemination and filtering of XML data streams, in: Proc. of IEEE DASC, IEEE, 2015, pp. 1870–1877.
[55] Jatinder Singh, Jean Bacon, David Eyers, Policy enforcement within emerging distributed, event-based systems, in: Proceedings of ACM DEBS, ACM, 2014, pp. 246–255.
[56] Kirill Belyaev, Indrakshi Ray, Enhancing applications with filtering of XML message streams, in: Proc. of IDEAS, ACM, 2016, pp. 322–327.
[57] Paolo Costa, Luca Mottola, Amy L. Murphy, Gian Pietro Picco, TeenyLIME: Transiently shared tuple space middleware for wireless sensor networks, in: Proc. of MidSens, ACM, 2006, pp. 43–48.

Kirill Belyaev is a Ph.D. Candidate in the Department of Computer Science at Colorado State University.

Indrakshi Ray is a Professor of Computer Science at Colorado State University. She obtained her Ph.D. from George Mason University.

Please cite this article in press as: K. Belyaev, I. Ray, Component-oriented access control—Application servers meet tuple spaces for the masses, Future Generation Computer Systems (2017), http://dx.doi.org/10.1016/j.future.2017.05.003