Efficient and Secure Information Sharing in Distributed, Collaborative Environments

Partha Dasgupta[1], Vijay Karamcheti[2], and Zvi M. Kedem[2]

[1]Dept. of Computer Science and Engineering, Arizona State University
[2]Dept. of Computer Science, New York University

Abstract. Collaboration enables a set of partners to work together while sharing information, resources, and capabilities in a controlled and accountable fashion. The partners themselves are organizations composed of people, departments, computational entities, and agents who perform tasks consistent with the internal rules of their organization. This paper presents a framework for managing some aspects of collaborations, both within and across organizations, between agents and resources or services, while guaranteeing that these interactions are secure, efficient, and satisfy specified access constraints. It describes some current and planned activities in a joint project by the coauthors at Arizona State University and New York University. We view collaborative operations in terms of the direct interactions between computations and services. To support collaborative operations we will rely on several techniques, including transitive delegation, cryptographic file systems, capacity sandboxing, reverse sandboxing, and fine-grained access control. These techniques facilitate scalable authentication and revocable authorization of agent computations even when they span resources of different organizations. In addition, they improve overall efficiency by permitting migration of computations to, and caching of services in, partly trusted environments of another organization.

1 Introduction

This paper describes a design for managing the interactions of humans and computers with computations and data in a dynamically changing collaboration [8], which can be informally described as "a possibly adhoc partnership," and possibly short limited lifetime. We assume that in general:

- Collaboration partners have diverse "strengths" and "weaknesses"
- Collaboration partners have constraints on their individual behavior during the collaboration's lifetime
- The collaboration partners' objectives translate into overlapping sets of short-term goals, and that's why they participate in a collaboration
- The collaboration partners prefer to behave in a reliable manner, to increase reputation for future collaboration.

Some consequences:

– The collaboration partners are willing to share some of their resources/capabilities, with the level of sharing separately decided for each "activity" (such as data access, computational access, service access) during the collaboration's lifetime.
– The mutual trust levels among collaboration partners can change dynamically, are very diverse, and may not even be symmetric for a pair of partners.

Let's consider a simple example. Three partners (corporations), Alpha, Beta, and Gamma, collaborate to develop, produce, and market a car. They would like to support the following:

– The Engineers (a class of agents) at Alpha can see the design data in Beta of the components that Alpha buys, but Beta's personnel cannot access Alpha's product design data.
– The Managers at all partners can access selected personnel data at all partners.
– Gamma allows Marketers of Beta to use its computers to run modeling programs. But, Beta's data is not leaked to Gamma, and Gamma cannot tamper with the computations.

Some desirable properties, informally stated:

– A new Engineer at Alpha can access the data stored at Beta, without Alpha informing Beta of every personnel change.
– An Engineer at Alpha, who becomes a Marketer, loses access to Beta's data.

2 Collaborations: Model and Issues

Our model of secure collaborations is similar to the CORBA Security Services reference model but differs from it in: (1) incorporating support for changes in trust relationships, and (2) assuming simpler policies in certain situations, allowing us to better focus on the underlying technical problems, and to strive for effective solutions. As the two models are closely related, our proposed techniques can be easily integrated into full-fledged secure distributed computing architectures such as CORBA.

Our key concepts for modeling collaborations are:

– **Partner:** A member of the collaboration. Examples: a city or a corporation.
– **Agent:** An agent belongs to a specific partner. It is an identifiable and an accountable actor. Agents are frequently grouped into classes for the purpose of assigning directives. Examples: an engineer or an automatic car navigational system.
– **Asset:** A physical host (for objects and delegates). Examples: a building or a car.
– **Object:** A program or a physical device controlled by a program, with a set of interfaces. Example: a database with a querying language or a robot.

- **Delegate:** An activity (mostly computerized) of limited lifetime (at most the lifetime of a collaboration) created by an *agent*, or by another *delegate*. (A delegate is *owned* by its creator.) Delegates can traverse multiple assets and access multiple objects.
- **Directive:** *Partnerss*, *agents*, *assets*, *objects*, and *delegates* operate adhering to a set of directives. These are essentially access control rules which specify where objects can be hosted, which interfaces are exposed to which delegates and under what conditions, and which trust levels have to be obeyed under what circumstances. Delegates and objects serve as atomic entities for enforcing directives, though it may be convenient to specify them in terms of higher-level concepts.

Directives are defined using a predetermined set of trust categories. Partners classify other partners (vis-a-vis their asset classes/object classes) into levels of trust that are commensurate with expectations of benefit derived from a history of previous interactions. We focus our attention on addressing the following key issues and those implied by them:

- *Asset Protection and Access Control:* All assets should be protected from attacks and only delegates with required credentials can access them. Assets should be protected from delegates' excessive resource use.
- *Object Protection and Object Access Control:* All objects should be accessible only by delegates with required credentials.
- *Delegate Protection:* Delegates running on remote assets should have some protection from the assets.
- *Dynamic Certificate Management:* All access control, especially when authenticated via certificates should be cognizant of the dynamic nature of certificates and directives, i.e. certification and rights revocation must be quick and efficient. Algorithms used should be scalable and hence distributed.
- *Dynamic Directive Management:* As directives change, the implications have to be effective very quickly. The management techniques must scale.
- *Efficiency:* A secure, but too slow collaboration support is ineffective.

There are important issues that we either do not address, or address only tangentially:

- *Design of Directives:* While we assume the existence of directives, we do not address the problem of expressing, creating, and verifying the consistency of directives.
- *Network Security:* We assume that all channels are secure, as this can be handled by standard techniques—albeit at some cost.

Our approach for managing collaborations relies on three main techniques, elaborated later: (1) *secure scalable access-control protocols*, (2) *system infrastructure for secure code and data sharing*, and (3) *fine-grained access control*. These techniques contribute to efficiency by permitting the co-location of delegates and objects, while ensuring that these delegate-object interactions respect collaboration access-control directives.

3 Secure Scalable Access Control

3.1 Certificates, Delegates, and Object Invocations

All *agents*, *assets*, and *objects* have certificates. Certificates contain (1) a public part: holder's identity, public key, additional information as described below, and the issuer's signature; and (2) a private part: holder's private key. (When a certificate is received, its validity can be tested by a standard challenge/response sequence, which needs the private part of the certificate.) Each partner has one *Certificate Authority (CA)*, which signs the partner's agents' and assets' certificates. Some additional details on certificates:

- Agent Certificate. The public part also specifies the agent's role in the partner's organization, trust level, generic access rights, etc. The private part of the certificate is stored in possibly multiple locations, referred to as trusted assets for this agent—and the agent is responsible for this.
- Asset Certificate. The public part also specifies whether this is a secure asset or not, and what are its specific characteristics. A secure asset runs a secure OS and is administered by a trusted set of agents. The private part is stored within the asset.
- Object Certificate. The asset on which it is currently stored signs it. The public part also contains a signature that certifies its "most recent content."

Note: If an asset is trusted by an agent, the asset belongs to the set of trusted assets of that agent. A trusted asset (of an agent) may be secure or insecure. A secure asset does not have to be trusted by any agent.

Agent certificates have short lifetimes and are renewed regularly to capture changing agent roles in the partner's organization. Asset certificates are valid for longer periods and are renewed when they expire. Object certificates expire and are updated when the objects are checkpointed. After an object migrates, the new asset reissues the certificate.

Delegates perform two external operations: *creation of delegates* and *invocation of objects*. The first refers to delegate spawning on other trusted or untrusted assets, and the second refers to the invocation of object services by delegates, the access control of which serves as the basis for collaboration security.

Creation of Delegates Agents can create delegates, and delegates can create delegates. Therefore, in general, at any time a forest of rooted trees of delegates does an agent's work. In this exposition, we consider a single agent. For this agent, some assets contain its complete certificate (public and private parts). As stated earlier such assets are called trusted (with respect to that agent). An asset (trusted or untrusted) running a delegate has access to the public part of the certificate of the agent that owns the delegate.

We present three representative cases for delegate creation, from which the full procedure follows. We assume that all assets have access to the appropriate directives, and only consider requests that conform to them.

An agent on a trusted asset creates a delegate on the same asset. This is a particular, simple case. The asset has the agent's certificate, so it can (1) authenticate the agent, and (2) allow creation of the delegate. We will refer to such a delegate as a *root delegate*, to this asset as a *root asset*, and to the agent as the *owning agent*. If an agent trusts several assets, and a delegate on a trusted asset wants to create a delegate on another trusted asset, the source asset authenticates the agent to the target asset and the target asset creates the delegate.

A delegate on a trusted source asset creates a delegate on an untrusted target asset. The source asset passes to the target asset the public part of its own certificate and the public part of the certificate of the agent who owns the delegate. The target asset authenticates the source asset. Then the target asset asks the source asset to authenticate the creating delegate, which the source asset can do, as it has the private part of the certificate of the agent who owns the delegate. Then the target delegate is created.

A delegate on an untrusted source asset creates a delegate on an untrusted target asset (transitive delegation). The creating delegate is a descendant of the root delegate running on a trusted asset. Before the target delegate can be created: (1) the target asset must verify the owning agent and (2) the root delegate must verify that the creation of the target delegate is allowed. There are straightforward methods of doing the transitive authentication, such as using a chain of signed certificates. However, these methods can be expensive. Later, we describe an alternative approach.

Invocation of Objects Securing object invocations involves two steps: (1) creating a secure association between source and target assets, and (2) enforcing desired access control. The first step authenticates the delegate and the source asset as having permission to access the object on the target asset and relies on protocols similar to the ones described above for verifying permissions of delegate creation on untrusted assets. Since object invocations exhibit temporal locality, we would like to avoid repeated authentication. Our approach relies on the source asset sending a time-stamped token and its signature to the target asset on the first invocation. On subsequent invocations, only the token is resent and all verification is performed at the target. Since the lifetimes of the tokens are short, stolen tokens are not a serious security risk.

Once the delegate has been authenticated, the second step verifies that the delegate does have authorization to invoke the requested method. We assume that the directives influencing this decision are captured in an *AccessDecision* module associated with the object (or the owning asset), which returns an access allowed/access denied response for access to an object method by a delegate. Changes in trust relationships between collaboration partners change the state in this module and are efficiently propagated as described in later. While secure, the scheme still suffers from an efficiency problem since each method invocation must be individually authorized. Later, we present a technique for providing

flexible specification and efficient enforcement of access-control decisions at user-specified granularities.

Efficient Transitive Delegation The goal is to allow a delegate running on an asset, possibly not trusted by the agent to whom the delegate belongs, to create a delegate on another asset, possibly not trusted by the agent—and to do this securely and efficiently. We sketch here, in simplistic terms, a protocol for this.

The source asset passes to the target asset: (1) the public part of its certificate and (2) the public part of the certificate of the owning agent of the creating delegate. From these, the target asset can also determine who are the owning agent and the root asset:

- The target asset authenticates the source asset and logs the request (maintaining an accountability trail).
- The target asset forwards the request to the root delegate.
- The root delegate, in cooperation with the trusted asset, evaluates the request and authorizes the target asset to create the target delegate.

The evaluation and the authorization need elaboration. The evaluation has two parts: (1) the root delegate based evaluation, and (2) trusted asset based evaluation.

The root delegate runs a verification routine that checks the request, the parameters, the asset and delegate it is coming from; and evaluates whether this request is part of the computation initiated by the root delegate. Then, the trusted asset that hosts the root delegate tries to ensure that the delegate being created will really be part of a rooted tree. This is done by the use of a token that is transmitted to each delegate in the tree (at creation time) and this token has to be produced when a new delegate is created. The trusted asset can verify this token to ensure graph connectivity. Once this is done, the root delegate is informed of the request to create a target delegate.

Note that the root delegate (or maybe its replica) must remain accessible at all delegate creation points. This is in fact an advantage, as it would be insecure for disconnected "descendant" delegates to act too autonomously. As remote delegate creation is not very frequent, this will not impact performance.

The delegate creation and the object invocation are scalable. Authentication happens locally, i.e., all delegate authentications are done at an asset (or assets) a particular agent trusts. Since each agent will have different trusted assets, the workload is distributed. In addition, the scheme ties into our design for scalable trust revocation.

3.2 Cryptographic File System

So far, we have discussed the general problem of sharing services across assets of collaboration partners. Techniques that are more efficient are possible when these services are accesses to immutable data (files). We describe a useful technique,

Cryptographic File System (CryptoFS). The CryptoFS is a general and efficient enabling technology for data sharing among large diverse groups of users.

All data that is accessible to the "participants" (which for simplicity, will be the agents in this exposition) will be stored on a (relatively large) number of *file servers*. It is not practical to ensure that all of these file servers are secure, resistant to attacks, and that they perform authentication correctly. In addition, since the trust relationships are complex and dynamic, it is not practical for every server to understand the rules of sharing, embodied in the directives. Hence, all files stored on file servers are pre-encrypted with a strong symmetric key (one key per file). The file servers perform no authentication or access control. They will send files to anybody, defending against denial of service attacks using a "capacity sandbox" described later.

The key of each file is stored in a (small) set of *key servers* (a type of asset), belonging to the partner who owns the file. The key server also stores the access control list of the file—a directive—if any.

To "use" the file (after getting it) a participant (i.e. an agent, via a delegate) needs a key. The agent authenticates itself to a key server using its certificate. If the directives allow it, the agent acquires the key and is able to decrypt the file. The decryption is done locally at the agent's site of choice.

All files stored at the file servers are immutable, thus the contents and the keys do not change. When an agent updates a file, the agent sends the new encrypted file to the file server. The file server stores this file without any performing any authentication (however, the agent's certificate can be checked to foil denial of service attacks). Then the agent has to register the new version of the file and the new key with the key service of the partner. If this operation succeeds, a new version of the file is created. The key servers keep audit trails of updates to ensure accountability.

For efficiency, it might be good to store an arbitrarily long logical file using a set of short physical files, which are distributed across multiple file servers. Then assuming physical access locality, the reads and the writes will be more efficient.

The central idea in CryptoFS is the separation of authentication and access. Such separation leads to lower administrative overheads and efficient runtime execution. This technique can also be extended to *object invocations* and we describe a proposed approach. First, a delegate acquires an object key, from the key server. Then it contacts the asset containing the object and receives a view key (we distinguish between objects and views—the latter is a run-time representation of the object with a restricted interface). Finally, the delegate performs an object invocation and sends the view key to the server handling the object, which performs the invocation, if the key is correct.

3.3 Scalable Certificate Revocation

As stated earlier, we use time-limited, short-lived certificates. An agent's certificate can be efficiently renewed as needed by the owning partner's certificate authority (CA), as the private key does not need to change. The agent presents its (soon to be expired) certificate, the CA authenticates the agent, and then

using the directives, it changes the expiration time, signs the certificate, and returns it. Assets' certificates are handled similarly. Since the CA is responsible for renewing only those certificates that belong to the agents of that partner, the workload on the CA is limited and the certificate lifetimes can be kept short.

The owning assets handle renewals of object certificates. Object certificates have to be renewed whenever the object is "significantly" updated (as the certificate contains the signature of the content) and when the certificate expires. Since the private keys of the assets do not change, the object certificates do not have to be renewed when an asset certificate is renewed.

3.4 Directives and Trust

The directives control trust among the agents and specify access privileges that the delegates, assets, and objects are to follow. Changes in the collaboration structure (such as partners joining or leaving) can be reflected as soon as the directives are updated. Directives are stored on all those assets of the collaboration, which are either trusted to store directives or need to access them. Updating the directives involves a secure broadcast, which is performed hierarchically to efficiently update the directive stores.

The directives also control how delegates are run on assets. When a delegate starts execution, the executable code can come from two sources. The code can belong to the asset, in which case the asset trusts the code; or it can be provided by the delegate, in which case the asset may or may not trust the code. Depending on the level of trust, the execution will either be denied, or the code is run normally. If the delegate's trust level is weaker than the asset's, the code is run in a sandbox that imposes qualitative and quantitative restrictions on its consumption of resources. If the delegate does not trust the asset, it can require the use of a reverse sandbox and verify whether such a reverse sandbox is used.

4 Secure Environments

The system infrastructure permitting secure deployment of delegates and object copies in partly trusted assets will consist of two components:

- *Sandboxing*, which refers to an execution environment that insulates the asset resources, both qualitatively and in terms of capacity constraints, from the delegate or object (hitherto jointly referred to as *client*).
- *Reverse Sandboxing*, which refers to a complementary execution environment that insulates the client code and data from the asset, ensuring that the latter can only interact with the former using a restricted pre-specified interface.

4.1 Sandboxing

Our approach for constructing a sandbox environment will take advantage of recently developed wrapper techniques such as system-call or API interception.

Most commodity OS permit the interactions of a process with the underlying OS to be routed through a wrapper layer (either in the delegate or as a separate process), which can then interpret these interactions as appropriate.

Primary distinguishing features of our sandboxing approach will be:

- *Secure user-level implementation.* The latter implies that the wrapper code appears to the OS as another user process, permitting flexible specification and efficient enforcement of sandboxing policies.
- *Imposition of quantitative capacity constraints* on delegate usage of asset resources such as CPU, memory, disk, and network bandwidth to within pre-specified limits

Secure User-level Sandboxing A rogue delegate can easily defeat a user-level sandbox, by simply bypassing the sandbox and directly calling the systems services (API's, system calls etc.). We are investigating techniques for building *non-by-passable user-level sandboxes* [5]. We present here only some of our initial ideas, using Windows NT for specificity, and put them in context:

- The sandbox is an API interceptor, attached to the delegate process when it is loaded, by modifying the memory images of its dynamically linked libraries (DLL's).
- The DLL images are write-protected, after the sandbox is attached, to prevent the delegate from re-modifying them.
- The VirtualProtect (the page protection API) call is itself sandboxed, to ensure that the delegate cannot unprotect these memory locations.
- The LoadDLL calls are also sandboxed, to ensure that the delegate cannot load a trusted DLL into another region of its memory to circumvent the sandbox.
- The delegate's binary is scanned for trap instructions—trap instructions are not allowed in the delegate code, as they may be used to directly invoke NT system calls—only binaries without such instructions are permitted to execute in the secure sandbox.

We briefly elaborate on some additional issues. A rogue delegate can still defeat the system by making direct calls to addresses that contain the system DLL's. This is prevented by a *stamping/tracking* scheme. The basic idea is that when an application enters the NT-DLL (the lowest-level DLL that actually makes the system call), we need to verify that the execution did indeed include the code fragments implementing the sandbox. We instrument the DLL load images to check the "path of execution" of the process from the point it leaves delegate code until the point it enters a function in NT-DLL. This check uses a stamping scheme that is not spoofable. Although the complete scheme requires elaboration, the main idea involves storing "stamps" at different points in the program trace in un-tamperable memory. The stamp captures the instruction pointer address and because of where it is stored, is accessible to the sandbox but not to the delegate. The sandbox code can then check the call stack against this stamp trace to verify that the code fragments implementing the sandbox have not been bypassed in the execution path.

Capacity Constraints Building a sandbox to enforce *quality constraints that only* affect handling of delegate program accesses to specific OS resources (can the resource be accessed or not) is now well understood. Enforcing capacity constraints (e.g., a delegate program must not use more than 20 MB of RAM) is more challenging. For example, system resources such as the CPU, memory, and network can be accessed without going through a high-level API call that can be intercepted. Moreover, individual API calls may not provide information sufficient to determine whether or not the delegate program has exceeded its capacity constraints and how to rectify that.

We briefly sketch the central ideas of our approach. In general, the delegate's program access to system resources can be modeled as a sequence of requests (implicit such as to a physical memory page, or explicit such as for a disk access) spread over time. Resource utilization can be constrained by either (1) control of the resources available to the delegate program at the point of the request, or (2) control of the time interval between resource requests. The primary challenges lie in employing primitives available in a commodity OS to (1) estimate how much progress the delegate has made (benefit derived), and (2) to effect the necessary control on resource requests with as low an overhead as possible.

To consider an example, let us see how this approach can constrain a client's consumption of physical memory resources. In this case, a good progress metric can be obtained by periodically sampling OS statistics about the current resident size of the client process. Achieving the desired control (adding or releasing some physical memory) is more complicated. We will use general OS support for virtual memory protection at the page level to dynamically track resident pages, and use this list to evict some resident pages whenever the progress metric exceeds the specified limit.

We have tested this approach for several other representative resources as well, including CPU and network. Our preliminary prototypes show that it is possible to control the actual utilization of the delegate program for each resource type to within 3% of the specified limit.

4.2 Reverse Sandboxing

While a sandbox protects the asset from a delegate, a reverse sandbox protects the delegate from the asset. Ideally, we would like to:

- Disallow the asset from tampering with the execution of the delegate, such as monitoring, spoofing, and ignoring API requests.
- Detect any errors in execution caused by the asset tampering with the delegate.
- Ensure that the delegate is made aware of any malfunction (malicious or not) in the asset.

A complete reverse sandboxing scheme, while highly desirable, requires solutions to some fundamental problems. We have developed an approach that partially addresses some of these problems, providing higher levels of confidence

in tamper-free executions. It includes mechanisms for (1) verifying that the asset does not tamper with the code and data of the delegate either before or during execution, (2) verifying that the OS and hardware are trustworthy, and (3) detecting emulation and API interception.

It is possible that the asset will put a delegate that wants to run within a reverse sandbox, in a sandbox. In this case, the delegate should be able to detect it, and then choose whether to run.

Delegate Tampering. Tampering with code or data, or the execution path of a program can be detected by the use of existing code mixing and code obfuscation schemes. These schemes detect whether the execution of a program produces expected results and can detect (with a high degree of confidence) any tampering during execution.

Trusting the Environment. The delegate must ensure that the underlying OS and the hardware of an asset have not been tampered with to construct a trusted component base. The latter permits the delegate to rely on trusted OS or hardware features (such as the availability of protected swap space, thread-local storage, and a CPU timer instruction) for defending against run-time attacks such as emulation or API interception. Our approach relies on the use of certificates, generated by of external smart card-like devices attached to the hardware itself. These devices check the integrity of the hardware and perform checksums on relevant portions of the OS. Integrity information is provided to the delegate, or its representative running on a trusted asset, using signed, verifiable, and tamper-resistant certificates.

Emulation and API interception. Some delegates may not want to be emulated, have certain API calls intercepted, or run with a debugger process monitoring their execution. These are examples of common run-time attacks, which can be mounted by an asset to control the delegate's interactions with the underlying OS and hardware. Detecting API interception relies on running checksums of the address space and checking against known hash values, or by verifying entries in specific DLL import tables. Several anomaly detection schemes developed in other situations (for instance, network-based intrusion detection) can be modified to detect emulation of instructions or functions. For instance, the delegate can securely interact with an external entity to compare execution times of specific code fragments against a previously constructed baseline. The occurrence of anomalies indicates the presence of monitors, emulators, or debuggers.

5 Fine-grained and Efficient Access Control

Our object invocation strategy relies on the separate run-time verification of access authorization for each object method that is invoked by a delegate. We have developed a programming and execution abstraction, object views [13], which can be very effectively used to specify object access-control policies at

arbitrarily fine-granularity (down to individual methods) and enforce them with small run-time overheads. As a programming abstraction, object views specify restricted interfaces through which objects can be accessed-these interfaces can be specified at fine granularity. As an execution abstraction, they allow efficient enforcement of a delegate's access to an object and efficient execution by caching only the necessary state.

5.1 Object Views

An object view consists of a restricted interface to one or more objects from the perspective of the invoking clients (delegates). This restricted interface prevents the delegate from obtaining access to the whole object; the delegate is provided access to only a subset of the defined methods. Since views represent a subset of the object functionality, they provide a natural granularity at which to specify access-control requirements. Views can be arbitrarily fine-grained (e.g., a single object method) as well as can capture all of the object functionality.

More importantly, views provide an appropriate granularity for performing access control at run time. The central idea of our scheme is that delegates can only access objects by first binding to the permitted view. Since the run-time representation of an object view implicitly captures exactly the set of methods that the delegate is permitted access to, the entire authentication and authorization (certificate and directive checking) can be completed on the first access to the view. Subsequent accesses can proceed with high performance using an access token generated on the first access. A change in trust relationships is handled by permitting the view (actually the server delegate running there) to unilaterally break the binding, forcing the client delegate to renegotiate its authorization.

Both the specification and implementation of views can be conveniently incorporated into an existing object-oriented programming language with minimal extensions. We have defined an extension to the Java programming language, VJava, which requires the addition of only two new keywords. We have also designed a preprocessor that generates the run-time representations of the defined views and interfaces with the AccessDecision module.

5.2 Custom Views and Caching

A more general form of object views, called *arbitrary views*, augments the basic view functionality described above with view-local fields and methods. These additions permit convenient specification of domain-specific access-control policies (referred to as "application access-control policies" in the CORBA Security Services reference model). For example, a partner can require that access to delegates executing on behalf of a particular class of agents is restricted to some fixed number of invocations in a fixed period. The advantage of arbitrary views is that such policies can be implicitly realized, just by including appropriate additional computation against view-local state. In this example, a view-local field can keep track of the number of invocations that have already been satisfied.

Additionally, object views improve the performance of object caching. Object caching improves overall efficiency by reducing the number of network transactions that are required to support delegate-object interactions. Subset views permit objects to be cached while maintaining coherence at subobject granularity, with coherence operations triggered only when conflicting views are accessed. Coherence operations are triggered only when conflicting views are accessed. Arbitrary views provide more flexibility, supporting the specification of domain-specific caching protocols. In general, three categories of custom protocols are possible:

- Synchronous views are tightly coupled with the home node copy: all object views have state identical to that at the home node.
- Asynchronous views are loosely coupled: object views differ for a period but eventually become consistent. This provides the underlying implementation with the freedom to de-couple protocol actions from data transfer, and optionally delay updates to coherence meta-data structures.
- Detached views have no coupling with the copy at the home node: they are essentially a snapshot of the current state. Detached protocols require only data transfer actions.

The programmer of the object can control the view models and hence provide higher performance object access, when the consistency requirements of the accesses are either relaxed or can be handled with alternative algorithmic means.

6 Selected Related Work

The objective of our project is to enable the secure sharing of information and services among collaborating partners with dynamically changing mutual trust relationships. There are many previous and ongoing efforts whose objectives overlap ours. Instead of listing these efforts individually, we identify three broad categories of related work and restrict ourselves only to some representative efforts in each category.

6.1 Secure Collaboration Infrastructures

Infrastructures for secure interactions between principals acting on behalf of collaboration partners typically fall into one of three broad classes:

Unicast-based infrastructures orchestrate interactions among collaboration members by building collaboration-wide interactions on top of separate secure pair-wise interactions between two collaboration members [12]. These efforts benefit from the relative maturity of security technologies such as public-key infrastructures and certificate distribution and verification services.

Multicast-based infrastructures [15] construct collaboration interactions on top of collaboration-wide secure group communication primitives [19].

Object-based distributed computing infrastructures, many of which are under development [16] [18], view the security problem as one of securing the invocations made by computations executing on behalf of one collaboration partner on the objects (encapsulating information and services) belonging to another. Securing invocations in turn requires authentication of client and server, access control to the methods of the object, ensuring the integrity of messages between them, and ensuring accountability. Our approach shares several of the goals of such infrastructures but there are differences: (1) we make explicit the entities acting on behalf of a collaboration partner such as agents and delegates, (2) we identify the state that is propagated whenever trust relationships between collaboration partners change, and (3) we support efficiency by relying on migration of delegates and caching of objects.

6.2 Secure Execution Environments

Techniques for protecting execution environments from mobile code can be classified into two broad classes. The first verifies, at start time, that the mobile code adheres to the security constraints imposed on it by the environment. Such verification can be accomplished either using certificate-based techniques [20], language-based protection approaches [4] [1], or recent techniques based on proof-carrying code [14]. The second class ensures that the mobile code does not violate its access constraints at run time by running it in a "sandbox," relying on either binary modification approaches [21] or active interception of the program interactions with the underlying OS [10]. Our capacity sandbox technique falls into this second class, but differs from previous approaches in being a user-level technique and additionally permitting the imposition of quantitative capacity constraints on application's use of resources to within pre-specified limits.

Techniques for protecting mobile code from tampering by malicious environments have typically relied on cryptographic techniques such as encrypted functions [3] [9] [17], code obfuscation [2] [6] [7], self-protecting code, and approaches that "salt" code/input with a random seed. All of these approaches achieve one or more of the following: (1) modification of the code and accompanying data to prevent meaningful disassembly, (2) the guarantee that any tampering will produce meaningless results, and (3) the assurance that any code tampering will be detected. Our approach for constructing reverse sandboxes builds upon these techniques but focuses on the complementary problem of preventing run-time interception of the program interactions with the underlying OS. To achieve this, our approach integrates multiple system components such as a secure channel to verifiable OS/hardware [11] and anomaly-based intrusion and monitoring detection.

6.3 Object Invocation Access Control

The granularity of run-time access control in most object-based distributed computing infrastructures (e.g., [16] [18]) has traditionally been a single method. Each method access is authorized independently, consulting an access decision

module associated with the object. This situation is true despite flexible specification of access control at different granularities such as multiple methods ("rights" in CORBA security) or the association of methods with one or more "security roles." This disparity between the granularity for specification and granularity for access-control enforcement results in heavyweight object interactions. In contrast, object views is accompanied with a different run-time representation of the object, which permits only the desired interface. Also our object access-control model captures the state required by the permitted methods, enabling the use of more efficient object caching protocols.

7 Sponsor Acknowledgment

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement numbers F30602-96-1-0320 and F30602-99-1-0517; by the National Science Foundation under CAREER award number CCR-9876128; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

1. K. Arnold and J. Gosling. *The Java programming language*, 2nd Ed., Addison-Wesley, 1998.
2. D. Aucsmith. Tamper resistant software: An implementation. In Ross Anderson, editor, *Information Hiding - Proc. 1st Int. Workshop, LNCS no. 1174*, Springer-Verlag, 1996.
3. D. Beaver, J. Feigenbaum, and V. Shoup. Hiding instances in zero-knowledge proof systems. *Advances in Cryptology – CRYPTO '90*, Springer-Verlag, 1990.
4. B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15th Symp. on Operating Systems Principles*, 1995.
5. F. Chang, A. Itzkovitz, and V. Karamcheti. User-level resource-constrained environments, In preparation, November 1999.
6. C. Collberg, C. Thomborson. On the limits of software watermarking. In *Proc. ACM Symp. on Principles of Programming Languages*, 1999.
7. C. Collberg, C. Thomborson, D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. ACM Symp. on Principles of Programming Languages*, 1998.
8. P. Dasgupta, V. Karamcheti, and Z. Kedem. Transparent distribution middleware for general-purpose computations, In *Proc. Parallel and Distributed Processing Techniques and Applications*, 1999.

9. J. Feigenbaum. Encrypting problem instances, or, ..., can you take advantage of someone without having to trust him. *Advances in Cryptology – CRYPTO '85*, Springer-Verlag, 1985.
10. I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proc. 6th Usenix Security Symp.*, 1996.
11. C. Harrison, D. Chess, and A. Kerschenbaum. Mobile agents: Are they a good idea? IBM Research Report, 1995.
12. S. Kent, R. Atkinson. Security architecture for the Internet protocol, Internet Engineering Task Force (IETF), Network Working Group, RFC 2401, 1998.
13. I. Lipkind, I. Pechtchanski, and V. Karamcheti, Object Views: Language support for intelligent object caching in parallel and distributed computations, In *Proc. Object-Oriented Programming Systems, Languages, and Applications*, 1999.
14. G. Necula and P. Lee. Proof-carrying code In *Proc. 24th ACM Symp. on Principles of Programming Languages*, 1997.
15. D. Malkhi and M. Reiter. A high-throughput secure reliable multicast protocol, J. of Computer Security, 1997, pp. 113–127
16. Object Management Group. CORBA Services: Common object services specification, security service, v. 1.2, 1998.
17. T. Sander and C. Tschudin. Protecting mobile agents against malicious hosts. *Mobile Agent Security*, LNCS, Springer-Verlag, 1997
18. Sun Microsystems. Java 2 Platform, Enterprise Edition Specification Version 1.2, 1999.
19. R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr, Building adaptive systems using Ensemble, *Software - Practice and Experience*, 1998, pp. 963–979.
20. VeriSign, www.verisign.net
21. R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symp. on Operating Systems Principles*, 1993.