To appear in the Proceedings of IEEE VR 2008.

Object-Capability Security in Virtual Environments

Martin Scheffler Jan P. Springer Bernd Froehlich

Bauhaus-Universität Weimar

Abstract

Access control is an important aspect of shared virtual environments. Resource access may not only depend on prior authorization, but also on context of usage such as distance or position in the scene graph hierarchy. In virtual worlds that allow user-created content, participants must be able to define and exchange access rights to control the usage of their creations. Using object capabilities, fine-grained access control can be exerted on the object level. We describe our experiences in the application of the object-capability model for access control to object-manipulation tasks common to collaborative virtual environments. We also report on a prototype implementation of an object-capability safe virtual environment that allows anonymous, dynamic exchange of access rights between users, scene elements, and autonomous actors.

Keywords: Object Capabilities, Security, Virtual Environments

Index Terms: D.1.5 [Programming Techniques]: Object-Oriented Programming; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality; K.6.5 [Computing Milieux]: Management of Computing and Information Systems—Security and Protection

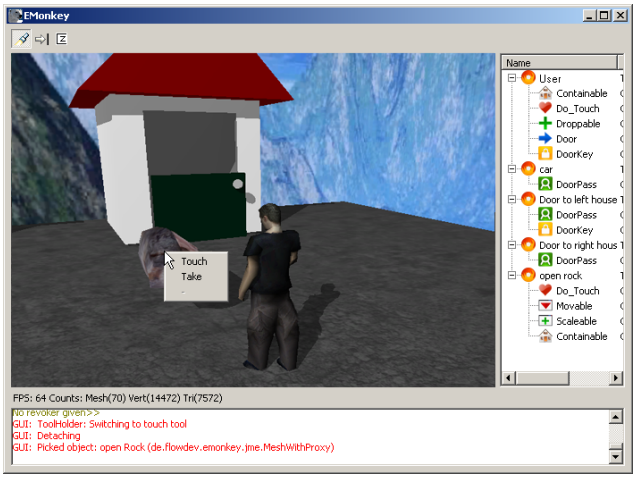

Figure 1: Screenshot of a prototype virtual environment using object-capability security.

1 Introduction

The rise of a new category of virtual environments could be observed in recent years: virtual worlds that allow thousands of users to interact and shape their surroundings. The premier example of this kind of virtual world is Second Life (http://www.secondlife.com). In Second Life, a number of tools can be used to add virtual objects to the world. Using a scripting language, users can program their objects to let them interact with other users or objects. It is possible to create houses, vehicles, clothes, and even autonomous actors. Because the objects of such a programming environment are not centrally controlled but belong to many different users with diverse agendas, safety and security play an important role. Programmers have to be able to define ways in which their creations may interact with other parties without exhibiting vulnerabilities. The common way to achieve this is to use access control lists (ACL), which have a number of shortcomings. To use resources restricted by ACLs, requesting programs have to have an identity, which is usually assigned by a central authority. This can become a performance problem as well as a scalability bottleneck. In addition, ACLs do not support the delegation of access rights, which is an important aspect of cooperative work.

In real life, although the use of ID cards is becoming more common, most doors are opened using keys. They provide access to specific objects (e. g. a car, a room). Handing keys to other people is a natural way of delegating access rights. This is the basic idea behind object-capability security. A capability is an unforgeable reference to an object that can be used to interact with that object in a specific way. Just like keys, capabilities can be communicated to other parties. There are a number of established software patterns that allow for dynamic assignment and revokation of fine-grained access rights in an anonymous way.

We created a prototype virtual environment using the capability-secure programming language E (cf. figure 1). In our system, capabilities define how actors can be accessed and manipulated (e. g. how they can be moved or how to change their appearance). Capabilities can be attached to the visual representation of their actors to make them publicly available and they can be exchanged using a drag and drop mechanism. Using the method of rights amplification [1], capabilities can be given to groups of users. By exchanging capabilities through a collision detection mechanism or a common parent in the scene graph, access can be made dependent on spatial properties such as distance and containment.

Object capabilities allow the participants of virtual worlds to define and exchange access rights in an anonymous and decentral way. This can be very useful in the creation of scalable and flexible virtual worlds that act as platforms for tasks such as simulation, entertainment, or commerce.

2 Related Work

2.1 Access Control Lists

Access control lists are attached to restricted resources and describe who is allowed to affect that resource in what ways. Usually, each row of an ACL contains an identity and an operation that can be executed on the resource with that identity. To execute an operation on a resource protected by an ACL, the requesting program has to authenticate itself to the ACL, which then determines if the operation is allowed. This approach to access control has a number of problems.

To access a resource restricted by an ACL, the requesting program has to authenticate itself. Authentication usually requires a central authority such as a login server, which can turn into a performance bottleneck and require a large amount of administrative work. Also, users may be unwilling to expose their identity for some operations.

In ACL systems, authority dependents on identity. A program using its own identity and authority to perform an action on behalf of another program can be tricked into using its authority for unintended actions. This transfer of authority is known as the Confused Deputy Problem [2].

When using ACLs, the identity of the user determines the operations that he may execute on the restricted resource. But not all actors in a virtual world are necessarily human and have an identity of their own. User-created objects either have to establish an identity of their own or act with the identity (and the full authority) of their creator, which is a security risk.

ACLs do not allow the delegation of access rights, which is an important aspect of cooperative work that should be supported. If a user has the right to move an object, he should be able to delegate that right to a tool in his hand, even if that tool belongs to another user. In ACL systems, it is possible to circumvent this restriction by creating proxies. This can become more of a security risk than explicitly allowing delegation, because a badly constructed proxy could be tricked into executing arbitrary commands under the authority of its creator.

Finally, it can be argued that ACLs are not a suitable metaphor for assigning access rights in a virtual world. Users have no way to dynamically react to changes on their access rights and restricted operations can only be executed by trial and error.

2.2 Second Life

Second Life (http://www.secondlife.com) is an Internet-based virtual world with the goal of enabling communication, interaction, and trade in a shared, user-controlled environment. All contents in Second Life are created by its users. The client software offers tools that can be used to create buildings or landscapes, build cars and other vehicles, or change the look and appearance of the user's avatar. Virtual objects can be programmed to react to outside events by using a scripting language. Scripted objects are able to communicate with their surroundings, change their appearance, or interact with servers outside of the Second Life environment using network protocols such as HTTP.

Access to user-created objects is restricted by ACLs, similar to UNIX file permissions. The rights to modify, move, or copy can be assigned to groups or all users. Scripts can request a number of specific permissions from users during their run time, for instance permission to control the point of view of the client. Furthermore, rules of interaction are encoded in the semantics of the builtin scripting language, which provides certain operations but does not support others, thereby shaping the style in which interactions and communication are possible.

Users of Second Life are not able to define custom permissions. The predefined permissions are limited in their flexibility. Some of them seem to be tailored for specific purposes, for instance for letting objects act as vehicles. Storage of permissions in a central database causes severe performance problems.

2.3 Den

The text-based distributed multi-user dungeon Den (http://homepage.mac.com/kpreid/elang/den.html) is a peer-to-peer system with the goal of supporting mutual suspicion between parts of its world. The Den project has made many interesting contributions on the subject of access control in virtual worlds; unfortunately it is largely undocumented. Den is written in the E language and makes extensive use of E's security features. Den's world model is room-based. Users can move from room to room and interact with the contents by using text commands. Rooms can be connected using entrances and exits. To set up an exit, access to the entrance of the room it should lead to is required. Users are only able to leave the current room if they have access to an exit. This access can be restricted by a door. To open a locked door, a key is needed, which is a normal object of the virtual environment. Doors support multiple keys as well as individual revocation of keys. Rooms residing on different servers can be connected over the network. This allows users to move from room to room and thus from server to server.

2.4 Spatial Model of Interaction

The spatial model of interaction uses the properties of space and distance as the basis for mediating interaction [3]. Each virtual object is enclosed by a bounding volume called aura. Only objects whose auras collide are able to interact. Objects can adjust the amount of awareness information they offer and receive dependent on distance. Each object has two more bounding volumes, nimbus and focus. The amount of awareness information that an object receives from other objects is dependent on the size of the sender's nimbus and the receiver's focus. Awareness information is exchanged only if the two collide. Focus and nimbus can adjust the amount of awareness information to be exchanged by gradual increase with closing distance.

2.5 The Avango Tool System

The virtual reality framework Avango [4] provides a virtual tool system [5] that allows users to modify nodes in the scene graph with tools attached to input devices. When a tool is applied to a node, the scene graph is traversed upwards until a node is encountered that has a point of application corresponding to the type of the tool.

3 Object Capabilities

3.1 The Object-Capability Model

In programming languages that support encapsulation, outsiders have no way of accessing the private state of an object without its consent. Access to encapsulated state can only be obtained via the methods of the object. Encapsulation is not only important for hiding implementation details of objects and thereby making programs easier to maintain. It can also be used to protect objects from each other. The object-capability model of access control is based on the model of object-oriented computation [6]. In an object-capability secure programming language, object references are treated as unforgeable capabilities. Possession of a reference gives the owner authority to communicate with the referenced object. All object references are capabilities and inter-object communication can only occur along these capabilities. For the remainder of this work the terms capability, object-capability, and reference will be used interchangeably.

In object-oriented programs objects communicate by sending messages along references. In a capability-safe programming language an object can obtain a reference to other objects in one of three ways:

- By endowment: when the object is constructed, the reference is passed as an argument to its constructor.
- By parenthood: the creator of a new object receives a reference to the new object.
- By introduction: the object receives a reference either as part of a message or as the return value of a message sent to another object.

The process of introduction is illustrated in figure 2. Carol and Bob have no way of communicating as they do not have references to each other. Alice holds references to both of them. She introduces Carol to Bob by sending Bob a reference to Carol as part of the message `foo`. Bob then can use this reference to communicate with Carol. In program code this is simply expressed as:

```
bob.foo(carol)
```

An object-capability secure language does not provide objects any way to obtain references other than along the reference graph.

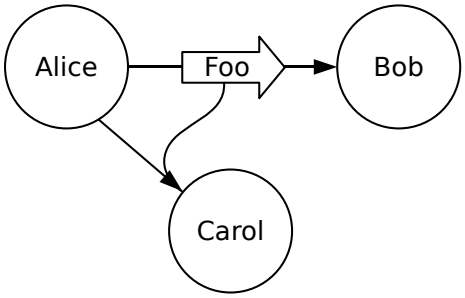

Figure 2: Access by introduction [7].

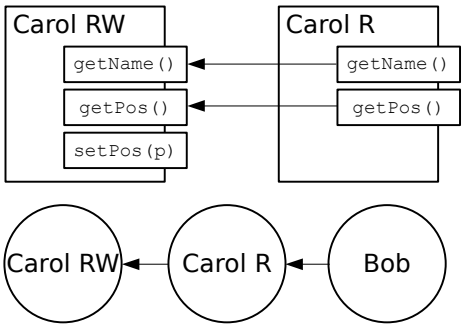

Figure 3: The facet pattern.

C++, for example, is not capability-secure because it allows pointer arithmetic, which can be used to obtain access to any object in process memory. Also, object-capability secure systems do not provide global functions and global mutable state that can be used to obtain powerful capabilities. Some programming language environments have a globally reachable operation `fopen` that can be used by any object or piece of code in a program to open files in the file system. This way, references to powerful objects can be obtained outside the reference graph. In an object-capability secure system, such an operation must not be globally reachable. Instead, the only way for an object to gain access to a file object should be by introduction through a third party.

When these conditions are fulfilled, a reference can be seen as a key to an object. Only the possession of a reference enables for sending messages to that object. Capability security allows strong modularization of code: untrusted code can be executed without having to worry about negative effects, because an object with no access to critical parts of the system (e. g. file system or network) can obviously do no harm; apart from using processing time. Only if an object has powerful capabilities it may affect the outside world.

3.2 The E Programming Language

E is an object-oriented programming language designed to enable secure, distributed computing (http://www.erights.org). E is a dynamically typed, pure object language. It employs the object-capability model to enable strong modularization. The E language environment provides a network protocol that allows processes to exchange capabilities over the network. Network capabilities can be used to asynchronously execute methods of remote objects, i. e. it does not block the sender; instead, asynchronous operations return promise objects acting as placeholders that are resolved once the return value arrives from the remote object. Actions can be registered to be executed when a promise is resolved. Also, messages can be send to the promise object even before it is resolved to a value—the message will be sent to the future location of the result object and queued until it is available. This way, delays caused by round trip time are minimized.

E provides a mechanism for encoding references as universal resource identifiers (URI). These URIs can be converted to references. Although these URIs contain a random string to prevent forging attempts, E programs can ensure that the URI to an object remains the same every time the process is restarted by using a serialization system. This way, the URI to an object becomes a persistent capability that can be communicated in text form or stored in a file.

3.3 Capability Patterns

Facet In some situations it can be useful to further restrict the access given by a capability. The facet pattern [8] (cf. figure 3), is used to create capabilities that only allow certain messages to be sent to the target object. The facet is an object that acts as a proxy for the target object, only forwarding those messages allowed by the facet definition. A common application of the facet pattern is to give non-mutable access to an object: only messages not changing the state of the target object are forwarded. The facet is provided to untrusted objects—they will be able to read the state of the target object, but will not be able to change it. In E a facet can be defined in the following way:

```
def carolR {
  to getName() { return carolRW.getName() }
  to getPos()  { return carolRW.getPos()  }
}
bob.foo(carolR)
```

Revocable Forwarder Another useful capability pattern is the revocable forwarder or caretaker, which can be used to allow temporary, terminable access to an object [9] (cf. figure 4). Once an object has access to another object it cannot be taken away. The object encapsulation makes it impossible to remove object references from the object, so the only way to revoke the access given by a capability is to destroy its target. Similar to the facet, the forwarder transparently forwards all messages to its target object. The forwarder can be used to communicate with its target just as if direct access was given. Each forwarder is paired with a revoker that can be used to disable it. Once disabled, the forwarder stops forwarding messages. All further messages to the forwarder will cause exceptions to be thrown, making the forwarder useless. In figure 4 Alice does not give Bob direct access to Carol, instead she provides a forwarder that transparently forwards all communication to Carol. Alice keeps the revoker and can use it to cut off access from Bob to Carol. Alice's actions can be written as E code like this:

```
def [forwrdr, revkr] := makeCaretaker(carol)
bob.foo(forwrdr)
revkr.revoke()
```

Sealer/Unsealer The next pattern is called Sealer and Unsealer [10]. They are constructed as pairs. A sealer is used to encapsulate

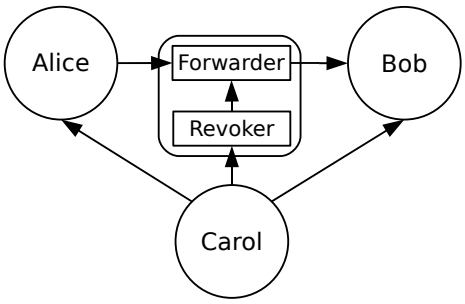

Figure 4: The revocable forwarder pattern.

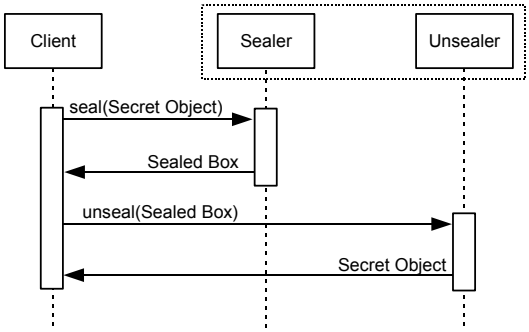

Figure 5: The sealer/unsealer pattern.

an object in another object that acts as a "sealed box." This box can be passed to untrusted intermediaries for storage or transportation purposes. Only parties in possession of the matching unsealer are able to unseal the box and use its contents (cf. figure 5). Sealing with sealer/unsealer pairs is comparable to asymmetric public key cryptography, although no actual encryption takes place; the sealed box protects its contents by encapsulation. In E, a pair of sealer and unsealer (called a brand) is constructed in the following way:

```
def [sealer, unsealer] := makeBrand("Name")
def secret := sealer.seal("My secret")
unsealer.unseal(secret)
# value: "My secret"
```

Rights Amplification The amount of access to a resource can be made dependent on the unsealers a party possesses. The resource provides facets of itself sealed with certain sealers. Only parties in possession of the matching unsealer(s) are able to unseal these facets and use them. This is called rights amplification [1, 7]. Rights amplification can be used to grant access to groups of objects: instead of storing powerful facets to all objects of a group, only a single unsealer has to be stored. Using the unsealer, the powerful facet can be retrieved directly from the resource, making storage unnecessary.

4 Capabilities and Virtual Environments

The object-capability paradigm can be applied to virtual environments on multiple levels. Its use might be limited to that of a user interface metaphor. Security between hosts can be achieved by using network capabilities. A capability-secure scripting layer enables programmers to create objects that exchange capabilities with their environment. Finally, by designing a virtual environment system from the ground up to follow capability security rules, users can be given direct, limited access to system modules.

4.1 An Object-Capability Secure Scene Graph

The central part of most virtual environments is the scene graph. The scene graph is the data structure that represents the different spatial elements which make up the virtual environment. In most networked VEs a client-server structure is used, where the scene graph resides on one or more central servers and is replicated on the clients. In interactive VEs users are usually able to influence the scene graph to some degree.

Especially for cooperative construction, training, or planning applications, a way to set fine-grained access rights for scene elements is needed. To be able to apply the object-capability model of access control to VEs, a prototype client-server application was developed. The client displaying the virtual environment is shown in figure 6. The server as well as the clients of the application are implemented in the E language. The scene graph is stored on the server. Each of its nodes contains visualization data including position, rotation,

Figure 6: Screenshot of the client application

scale, material properties, and mesh data. The nodes of the scene graph are interconnected by read-only facets. These facets can be used to read the properties of the nodes, but not to change them. Effectively, write access to a single node does not give further write access to the rest of the graph.

When the server starts up, it creates an URI that can be used by the clients to open a network reference to the root node of the scene graph. This URI includes a random part, which makes it unguessable. The URI thereby acts as a capability in text form. It can be given to the client users by email or similar communication channels.

As soon as the URI is stored to a text file in the root folder of the client application, the client can use it to open a network reference to the root node. The client is now able to send messages to the root node and exchange further network references. From the root node the client can obtain read-only facets of its children nodes. The client is now able to recursively traverse through the scene graph structure on the server and create local proxies for all nodes. These proxies are responsible for data replication as well as for locally displaying visual representations of their remote counterparts.

4.2 Interaction and Floor Control

The client has now obtained read-only facets of all nodes in the scene graph. It can use these facets to replicate the nodes, but not to interact with them. To do this, further capabilities of the nodes are needed. One way of implementing interaction between user and scene is the system of tools, mediators, and interaction operators [5], where tools controlled by input devices can be used to modify nodes of the scene graph. The user can choose between a number of tools with different purposes. Tools can be used among other things to change the position, material properties, or shape of nodes.

Mediators define a point of application for tools of a specific type. A node can be defined as modifiable by a certain type of tool by attaching a mediator of the corresponding type to it. The actual interaction between tool and mediator is handled by an interaction operator. The lifetime of this object is limited to the duration of the interaction. Its task is to implement the transfer function based on the inputs coming from the tool and forward the results through the mediator into the scene graph.

Usually, situations when multiple tools try to obtain control of a node are solved in a "first come, first served" manner. But in large-scale systems, malicious users could use this to block objects simply by not releasing their tools. To prevent this, policies for "floor control" are needed. These policies are used to decide which of a number of contending users will receive control for a pending

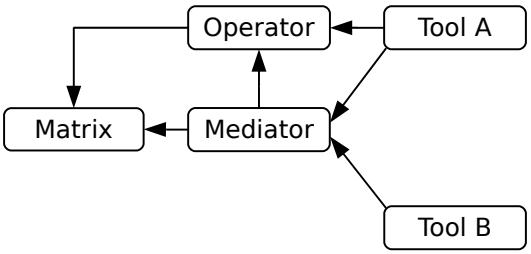

Figure 7: Interaction control using a revocable forwarder. The mediator uses its direct access to the node to construct an interaction operator. By passing this operator to Tool A, it gives it the authority to affect the node indirectly. When Tool B requests access, the mediator can choose to interrupt the ongoing interaction operation and create a new operator for Tool B.

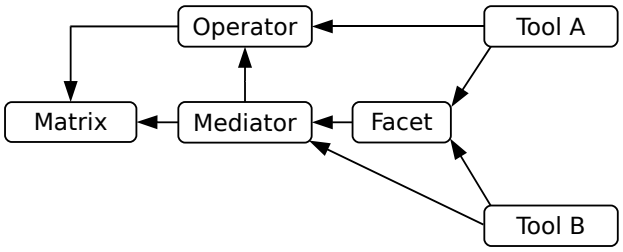

Figure 8: Interaction control in a teacher and students scenario. While a student (Tool A) is only able to use the functionality provided by the facet, the teacher (Tool B) receives direct, unrestricted access to the mediator.

resource based on a number of conditions [11]. In most multi-user applications, policies for floor control are defined system-wide. In object capability systems the scene elements themselves are able to define and enforce their own floor control policies. The system of tools, mediators, and interaction operators can be modified to support this.

When the system of tools, mediators, and interaction operators is executed in a capability-secure environment, interaction operators can be interpreted as capabilities which allow their holder to modify nodes in ways specified by their types. Mediators act as gate-keepers to their nodes, controlling access and enforcing floor control. By wrapping interaction operators in revocable forwarders (see section 3.3) before passing them to tools, the mediators are able to interrupt ongoing interaction operations at any time and cut off access to the node. This is illustrated in figure 7. When a tool requests an interaction operator from a mediator that has already granted access to another tool, the mediator can choose to revoke the access of the current tool and give access to the new tool, thereby transferring floor control. It can also choose to ignore all incoming requests until the current tool finishes its operation.

These relatively simple policies for floor control have their disadvantages. Always granting exclusive access to any requesting tool may result in situations where users are taking turns in stealing access to the node from another. When access is never granted during an interaction operation, users are able to block nodes. One way in that the mediator could deal with these problems is by allowing exclusive access only for a certain amount of time. Once the time has passed, it allows other tools to take over control from the current one.

4.3 Teacher and Students

In training situations, it may be useful to have a user who is "in charge" (i. e. the teacher), whose word carries more weight than others. For instance, the teacher could have the ability to access nodes exclusively even if this interrupts ongoing operations of unprivileged users (i. e. students). Often such requirements are met by assigning roles to the identities of users. In such a system, to obtain teacher priorities, the user has to authenticate himself and prove that he is a member to the teacher group.

In large VEs the relationships between users and objects can be complex, especially when users are allowed to create their own contents and define their own access rules for them. This may result in "role explosion", a situation where large numbers of roles or user groups are created to specify the various relationships. In capability-based environments, the requirements for a student/teacher system can be met at the object level without the need for identities or roles. Teachers differ from students in that they are able to access a more powerful facet of the mediator, which they can use to obtain exclusive access to the node, possibly revoking existing interaction operators held by students (cf. figure 8). A simple way to implement this is to use rights amplification (see section 3.3). The mediators provide a powerful facet of themselves which is sealed with a special sealer. Teachers are defined as users in possession of the unsealer needed to access these facets. Once unsealed, teachers use these facets to obtain access to nodes, thereby overriding students. Although students are also able to access the sealed facets of the mediators, they are not able to use it as they are not in possession of the necessary unsealer.

4.4 Communicating Capabilities

In virtual environments, as in real life, access to an object may not only depend on prior authorization, but also on a number of other conditions. A user that receives the key to a door is only able to use it if he is physically able to reach the lock. By communicating capabilities along different channels, different conditions can be checked before granting access. In our prototype application, capabilities can be communicated in a number of ways:

- **Capabilities for all participants in the scene:** It is possible to attach capabilities to the publicly available read-only facet of a node. All participants of the virtual scene are able to retrieve and use these capabilities. In an ACL-based system, this could be compared to assigning a read-only permission to the "all" user group.
- **Using rights amplification:** By sealing a capability before adding it to the read-only facet of a node, its usage can be restricted to parties in possession of the corresponding unsealer. In an ACL-based system, a similar effect could be achieved by granting a permission to a certain user group.
- **Using drag and drop:** The capabilities of various scene elements that the user has acquired are displayed in the capability list (cf. figure 6). By dragging an entry from this list and dropping it onto the node of a virtual actor, the capability represented by that list entry is given to the actor. The actor can react dynamically to this assignment or store the capability for later use.
- **By collision:** Actors may exchange capabilities based on a collision detection mechanism. Capabilities exchanged by collision can only be used by those actors that are within each other's proximity. Once the collision ends, all exchanged capabilities are immediately revoked.
- **By containment:** The nodes of a scene graph form parent-child relationships, which often correspond to logical relationships. By making capabilities dependent on their position within the scene graph it is possible to use these logical relationships for access control. In our prototype, actors can "contain" other actors. Actors can use doors to move from one container to the next. When an actor uses a door, the node representing it visually is transferred from the child list of its current container to that of the actor targeted by the door. The contained actors can use their container to exchange capabilities with their siblings. The container wraps all exchanged capabilities in revocable forwarders

before passing them on. When an actor leaves its container, the container revokes all capabilities exchanged between this actor and its siblings.

4.5 Combining Capabilities

By making access dependent on several capabilities which are exchanged through different channels, it is possible to enforce terms of service on virtual actors. To be able to use a door, actors have to meet three conditions: They have to be inside one of the rooms linked together by the door, be close enough to touch it, and be in possession of the right key. To enforce this, the `enter` method of the door expects three different capabilities as arguments. One of these capabilities is exchanged by collision. Only if actors touch the door do they receive this capability from the collision manager. The actors can prove that they are in the room leading to the door by supplying a capability that the door gives through the common parent node. The key capability can be given to the actors by drag and drop, or alternatively the door can be made usable by everyone by attaching the key to the door.

4.6 Meta-Information on Capabilities

To be able to react to received capabilities in a reasonable way, actors have to have information on its type and the actor it is a capability of. In our application prototype, capabilities are passed around wrapped in objects of the class `CapInfo`. A `CapInfo` object provides the following information about its capability:

- **Type information** Its type is identified by a simple string value, for instance "Move" or "Buy."
- **Revocation state** For capabilities that are wrapped in revocable forwarders the revocation state is represented by a promise object that is resolved once the capability is revoked. It is possible to register functions with the promise object to be executed once it becomes resolved. This can be used to dynamically react on capability revocation. For non-revocable capabilities this value is not set.
- **Actor identity** The `CapInfo` object includes a reference to the read-only facet of the target node. Recipients can use this reference to determine the actor the capability references. This read-only facet is accessible by all participants of the scene anyway, so including it again in the `CapInfo` does not give any additional authority.
- **Sealing state** If the `CapInfo` object contains a sealed capability, information for identifying the matching unsealer is included. This value is not set if the capability is not sealed.

5 Scenario: Bumper Cars

The following scenario is an example of rights exchange between multiple parties:

> Alice wants to go on a bumper car ride together with her friend Bob. As soon as she has bought a ticket at the ticket booth they enter a car together. Alice puts her ticket into the designated slot in the car. The car starts to drive, and because the steering wheel is mounted between them, both Alice and Bob are able to steer it. When the time is up, the car stops moving and Alice and Bob have to get out.

We implemented this scenario with our prototype. We will now give an overview about which capabilities are exchanged between the parties in which ways. At the outset of the bumper car scenario, there are four parties: Alice, Bob, the ticket salesman, and the bumper car. The ticket salesman and the bumper car are virtual actors programmed by scripts. Alice and Bob are also virtual actors, but they are under the control of real users.

The ticket salesman wants to earn money, so it is in his interest that everybody is able to buy tickets from him. By creating a "Buy" capability and attaching it to his ticket booth where everybody can reach it, he makes sure potential customers are able to buy tickets from him.

Alice moves her actor through the virtual environment. She has an input device that she can use to select or manipulate objects. Once she encounters the ticket booth, she selects it, which causes her client software to inspect it for capabilities that can be used for interaction. Because a capability of the type "Buy" was found, the client presents Alice a pop-up menu with a choice labeled "Buy." If Alice selects this entry, a monetary transaction is initiated. Monetary transactions can be handled with capabilities in a manner that does not require participants to reveal their identities [12]; there are a number of implementations of capability-based virtual money available in E [13, 7], so we will not pursue this topic further. Now that the ticket booth has been paid, it creates a bumper car ticket and gives it to Alice as the return value to the "Buy" message. The ticket is wrapped in a revocable forwarder, which the booth can later on use to invalidate it. Alice's client stores the ticket in her capability list.

Alice navigates to a car. Once she is close enough to touch the car, she receives the capability to enter it. Only users who can supply a valid ticket are able to use this capability. If Alice selects the car, her client inspects it for interaction choices. Because an "Enter" capability is found and the required ticket is also present, a pop-up menu with an "Enter" choice is created. If Alice selects that choice, her client passes the ticket to the car along with the capability to remove Alice's actor from its current location. The car uses this capability to move the actor into its sub-scene. Also, it sends Alice's ticket to the ticket booth. The ticket booth starts a timer for the duration of the ride.

Once inside, Alice receives capabilities for the interior of the car. Most importantly, she is able to use the steering wheel, which means she is able to drive the car. Also, she receives another "Enter" capability. Alice gives this capability to Bob by using drag and drop. Bob is now able to enter the car and access the steering wheel, too.

Once the time is up, the ticket booth invalidates the ticket and sends a message to the car to eject its occupants. Because the ticket is now invalid, Alice and Bob have to pay again before they can go on another ride.

The described scenario shows the dynamic exchange of fine-grained access rights between multiple participants made possible by object-capability security. None of the interactions require identification. The participants dynamically react to changes of access rights. Access rights are granted dependent on conditions of distance, parent-child relations in the scene graph, and prior authorization.

Using ACLs, the dynamic exchange of access rights described here would be much harder to realize. Because subjects are not informed about changes in their permissions in ACL-based systems, dynamically reacting to changes is more difficult to implement.

6 Actor Implementation

The nodes of the scene graph are only responsible for storing and replicating their visual properties. Interaction between a node and the environment is handled by a separate object: the controller. The controller object most often resides on the server, although it is also possible to create the controller on the client side using E's distributed computing model. The controller is responsible for granting and revoking capabilities as well as for dynamically reacting to received capabilities and other input. It is able to exert floor control policies as described in section 4.2.

On the client side, each node is paired with a view object, which is responsible for displaying the node locally. Each view contains a network reference to a "Use-Only" facet of the the node's controller. This reference can be used by the client to communicate with the controller and obtain capabilities. When a node is selected, the client

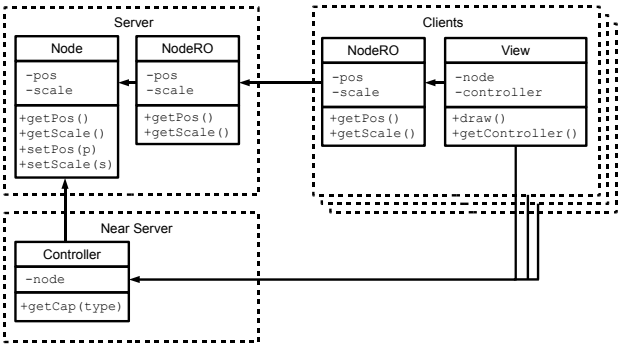

Figure 9: Model, view, and controller of an actor.

sends a message to its controller requesting a "Touchable" capability. If that capability can be retrieved, the client sends it the message `touch()`.

The controller, the node with its proxies on the various clients, and the views attached to them together form a virtual actor that can communicate with its surroundings as well as control its own visual appearance. As shown in figure 9 this architecture is similar to the classic Model-View-Controller meta pattern [8], with the extension that the model is a distributed object spanning multiple systems.

The creator of a node is able to create capabilities of the actor and store them in the controller. The facet of the controller which is distributed to the clients cannot be used to add capabilities to it, only to retrieve them. The following E code shows how to create a virtual actor as well as how to add a `scale` capability to its controller and making it publicly reachable:

```
def addActor(makeNode, makeController) {
  def actorCtrl := makeController("MyCtrl")
  def node := makeNode (
    "MyNode",
    [ "url" => "//server/actor_geom.ms3d",
      "controller" => actorCtrl.useOnly()
    ]
  )
}
rootNode.add(node.readOnly())
def scale := makeScale(node)
actorCtrl.add(scale)
```

To create nodes, access to the node constructor `makeNode` is needed. Only nodes created by this constructor are admitted to the scene graph. To add nodes to the scene, clients need network references to both this constructor and a node that allows for adding child nodes. The arguments to `makeNode` are a name for the node and a list of properties that define its visual appearance. The `url` list entry defines the location of a 3D-mesh file to be loaded as a visual representation of the actor. If it is set, the client loads the indicated mesh file using the HTTP protocol and displays it as the visual representation of the node. The reference passed as the `controller` property will be distributed together with the node to the various clients. Because other users should be able to only retrieve capabilities from the controller, not to add any, a "Use-Only" facet of the controller is given to the node constructor. `makeNode` returns a reference to the newly created node. A facet of the node is created and sent to the root node as the argument to the `add` message. This causes the root node to add the node to its list of children and to distribute it to the all participating clients. Because only a read-only facet of the node is given to the root node, the creator can be sure that only he has the capability to modify its properties. The reference to the node is then used to create a "Scale" capability. The operation `makeScale` creates a facet of the node that can be used to alter its scale. This capability is made publicly available by adding it to the controller of the node.

7 Capabilities in Large-Scale Virtual Worlds

Authentication is a problem especially in decentralized virtual worlds where users move from host to host as they change their virtual location. In the World Wide Web, authentication is handled by the web sites themselves, with the result that users have to create a separate user account for each web service they want to use. Central authentication authorities could be used to manage user identities. This may create a scalability bottleneck as well as a central point of failure. Alternatively, capability security allows users to acquire and store access rights without having to first establish an identity. This creates the possibility for decentralized virtual worlds where users are able to interact in a flexible and anonymous manner.

A common criticism of the capability approach seems to be that by allowing capabilities to be delegated, it is often impossible to determine who is accountable for the way in which the delegated capabilities are used. This can be countered by using the Horton protocol [14], which allows the delegation of capabilities while making the recipient himself accountable for the actions he takes with them. There are many applications for this protocol in virtual worlds: In our bumper car example, Alice could delegate the capability to drive the car to Bob without making herself responsible for the way he uses it. The Horton protocol also shows that it is possible to implement identity-based security systems on top of capability systems, thereby combining the best of both worlds.

In virtual worlds that allow users to execute scripts and add contents, self-replicating objects are a big problem as these can quickly use up all system resources. This is called gray goo attack or fork bomb attack and has emerged in various multi-user dungeons (MUDs) and systems like Second Life. In capability-secure virtual environments it is possible to prevent this kind of attack by using an agoric approach [13]. The creation of objects is seen as a capability which can only be used in exchange for a certain amount of a virtual currency. The cost of creating new objects can be made dependent on the number of objects already present in the system. This way, as the number of objects approaches a system limit, the "price" for new objects rises until nobody is able to afford it.

The use of object capabilities does not have to impose a performance penalty. Many of the performance trade-offs of object-capability security have already been accepted when an object-oriented language was chosen for implementing a VR system. By using "taming systems" such as Joe-E (http://www.joe-e.org), mainstream object-oriented languages can be made capability-secure. It allows existing development tools to be used for creating capability-secure systems. Also, the need to acquire a capability for an object before being able to affect it does not impose the need for an extra network round-trip between client and server. Using E's "promise pipeline" feature [6], the client can queue messages so that they are delivered to the capability immediately.

By treating the nodes of the scene graph themselves as capabilities, parts of the scene can be hidden from unauthorized users or displayed using lower level of detail substitutes. In multi-user construction tasks, confidential parts of the scene could be hidden from unauthorized users until they are introduced to them by an authorized party.

In capability-secure virtual environments it is possible to give user-created scripts direct access to parts of the core system. Allowing scripts restricted access to the collision detection system or to influence the physical dynamics of the world enables a whole new level of programming within the virtual world. It then becomes a platform for creating complex cooperative applications like multi-user CAD systems, games, and team planning scenarios.

8 Conclusions and Future Work

In capability-secure virtual environments, users and virtual actors are able to anonymously exchange fine-grained access rights and dynamically react to granted access rights. Because use of capabilities does not require authentication, central authentication authorities become unnecessary. This allows for the creation of fully decentral web-like virtual worlds.

Especially in large-scale virtual worlds that are not under central control, it is important that virtual actors are themselves able to control the ways in that they can be influenced by the environment. In capability-secure environments, actors can define and enforce floor control policies as well as making access to their functionality dependent on contextual variables such as proximity or containment. By using a collision detector to exchange access rights, access to the methods of a virtual actor can be made dependent on distance. Virtual actors can decide which interactions to allow at a certain distance or which awareness information to reveal. By employing multiple collision volumes with different extent, it is possible to implement the concepts of nimbus and aura [3].

Interaction design is a critical aspect of computer security [15]. We argue that the provision of capabilities is a better metaphor for managing access rights than adding the subject to an ACL. By binding capabilities to user interface elements, assigning permissions can be made transparent to the user.

Our virtual environment application prototype is far from complete. There are several directions in which this work can be extended. The user interface for managing capabilities is only rudimentary. A more complete interface would have to support the creation and assignment of facets as well as revocation of delegated capabilities. Also displaying capabilities in a list on the screen may become confusing if large numbers of actors are present in the scene. Usability can be possibly improved by displaying capabilities in the scene coupled to their targets. The PASION project [16] has developed a gesture interface for group interaction in immersive virtual environments. By using this or similar interfaces to communicate capabilities, access control can be made more intuitive and user-friendly.

In the SPACE model [17] the virtual environment is segmented by boundaries. Users are allowed to traverse boundaries depending on possession of passwords or other attributes. User access to an object is dependent on the paths by which they can reach the object. Using the developed mechanisms for capability assignment, access control systems such as SPACE can be implemented on the scripting level in a secure way.

Upon receiving capabilities from a third party, the recipient cannot be certain that the information contained in the `CapInfo` object is correct. The recipient also cannot be certain that he is not communicating with an impostor. A future implementation might have to provide a service for verifying the information contained in `CapInfo` objects. This means that different tasks require different protocols and capabilities. Is it then sufficient to have a limited common vocabulary of capabilities and protocols to use them? Or should users be able to define new types of capabilities and protocols?

If large-scale virtual worlds are to become platforms for trade and cooperative work, better tools are needed for defining and enforcing access rights as well as security protocols. By using the object-capability model to design and implement virtual environments, safe cooperation between untrusting partners becomes possible and many of the security problems of current virtual worlds can be solved elegantly.

Acknowledgments

We thank Mark Miller, Kevin Reid, and the ERights community for discussion and constructive feedback on earlier drafts.

References

[1] A. K. Jones. *Protection in Programmed Systems*. PhD thesis, Carnegie-Mellon University, Dept. of Computer Science, 1973.

[2] N. Hardy. The Confused Deputy (or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988.

[3] S. Benford and L. Fahlén. A Spatial Model of Interaction in Large Virtual Environments. In *ECSCW'93: Proceedings of the 3rd conference on European Conference on Computer-Supported Cooperative Work*, pages 109–124. Kluwer Academic Publishers, 1993.

[4] H. Tramberend. Avocado: A Distributed Virtual Reality Framework. In *Proceedings IEEE Virtual Reality '99 Conference*, pages 14–21. IEEE, 1999.

[5] H. Tramberend, F. Hasenbrink, and B. Fröhlich. Tools, Mediators, and Interaction Operators: A Concept for Interaction in Virtual Environments. In *3rd Immersive Projection Technology Workshop*, pages 77–79. Center of the Fraunhofer Society Stuttgart IZS, 1999.

[6] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

[7] M. S. Miller, C. Morningstar, and B. Frantz. Capability-based Financial Instruments. In *Proceedings of Financial Cryptography*. Springer Verlag, 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[9] D. D. Redell. *Naming and Protection in Extensible Operating Systems*. PhD thesis, MIT, Project MAC, 1974.

[10] J. H. Morris. Protection in Programming Languages. *Commun. ACM*, 16(1):15–21, 1973.

[11] J. Boyd. Floor Control Policies in Multi-User Applications. In *CHI '93: INTERACT '93 and CHI '93 Conference Companion on Human Factors in Computing Systems*, pages 107–108, New York, NY, USA, 1993. ACM Press.

[12] K. M. Kahn and W. A. Kornfeld. Money as a Concurrent Logic Program. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proc. of the North American Conference 1989 (Volume 1)*, pages 513–535. MIT Press, 1989.

[13] M. S. Miller and K. E. Drexler. Markets and Computation: Agoric Open Systems. In B. A. Huberman, editor, *The Ecology of Computation*, pages 133–176. Elsevier Science Ltd, 1988.

[14] M. S. Miller, J. E. Donnelley, and A. H. Karp. Delegating Responsibility in Digital Systems: Horton's "Who Done It?". In *Proceedings HotSec '07*, 2007.

[15] K. Yee. User Interaction Design for Secure Systems. In *ICICS '02: Proceedings of the 4th International Conference on Information and Communications Security*, pages 278–290. Springer-Verlag, 2002.

[16] T. Pfeiffer and M. E. Latoschik. Interactive Social Displays. In *IPT/EGVE 2007: 10th Immersive Projection Technology Workshop & 13th EG Symposium on Virtual Environments*, pages 41–42. Eurographics, 2007.

[17] A. Bullock and S. Benford. Access Control in Virtual Environments. In *VRST'97: Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 29–35. ACM Press, 1997.