Dynamic Exchange of Capabilities Between Mobile Agents

Jim Miani

CIS 642

Outline

- Supporting Concepts
  - Relevant Java properties
  - Process-level security
  - Terminology
- Paper Abstract
- Protocol Objectives
- Project Requirements
- Implementation Details
- Related Work
- Criticism/Discussion

Relevant Java Features

- Polymorphism
- Inheritance
- Interfaces
- Type Safety (no arbitrary memory references)
- Platform-independence
- Object serialization

Process-level security

- Pre-network era, security typically handled by OS (UNIX or mainframe).
- PC operating systems typically give super-user privileges by default
- Thus security must be enforceable at process/language level.

Terminology

- An agent is a mobile process with it's own context (code and data).
- Access rights are the set of capabilities one agent grants another.
- A capability is a token that identifies an object and contains access rights.

Abstract

- Define a protocol for a secure exchange of software capabilities between processes
- Sample implementation in Java
- Use Interface Definition Language (IDL) to separate protection policy of agent from agent's application code

Why Use Agents?

An agent is an independent process which may travel to several sites in order to complete its task.

Pros	Cons
• Peer-to-peer model • Ideal for distributed computing • Efficient use of network resources • Flexibility • Decentralization	• Security • Security • Security The point is that this is something we would like to do *if and only if we can ensure secure operation.*

Protocol Objectives

- Evolution: Dynamically exchange capabilities
- Decentralization: Agent-level administration of capabilities
- Mutual suspicion: All agents suspicious of other. No imposed hierarchy.
- Modularity: Protection scheme separate from application logic.
- Portability: Language-independent.

Language features

- Again, NOT Java-specific
- No direct memory addressing
- Type-safety
- Dynamic class loading
- Dynamic binding
- Polymorphism (interface definitions)
- Object serialization
- Name server

Project Requirements

- Isolation - Agent confined to granted access privileges
- Access Control - Definition of grant policy
- Authentication - Correctly identify an agent or agent's source.

The Protection Model

- To invoke an object method, agent must have reference to object and capability (i.e. access rights) to use it
- Creator of object typically granted full access to object at instantiation
- Capabilities along with object reference can be passed to other agents by creator

Exchanging Capabilities

- Need to dynamically exchange capabilities
- Need method to control exchange
- Language extensions are poor solution
- Use interfaces (Interface Definition Language)
- Very similar to typical OO inheritance protection schemes, but greater granularity

Mobile Agents

Part II

Jim Miani

CIS 642

Polymorphism†

- Inclusion Polymorphism allows the same identifier to reference objects of different types provided they have a common ancestor class.
- Operation Polymorphism (a.k.a. overloading) determining the method invocation by signature of arguments.
- Parametric Polymorphism uses types as parameters in generic class declarations. This is a feature of ML.

† As defined in Wilkie '93 and Pohl '94

Recap

- Objective - Define a protocol for mobile agents to exchange capabilities in a secure and dynamic fashion.
- Disengage definition of agent grant policy from agent application logic.
- Definition of grant policy should be independent of implementation language.
- Decentralize grant and delivery of capabilities
- Use Interface Definition Language to achieve these goals.

Interface Definitions

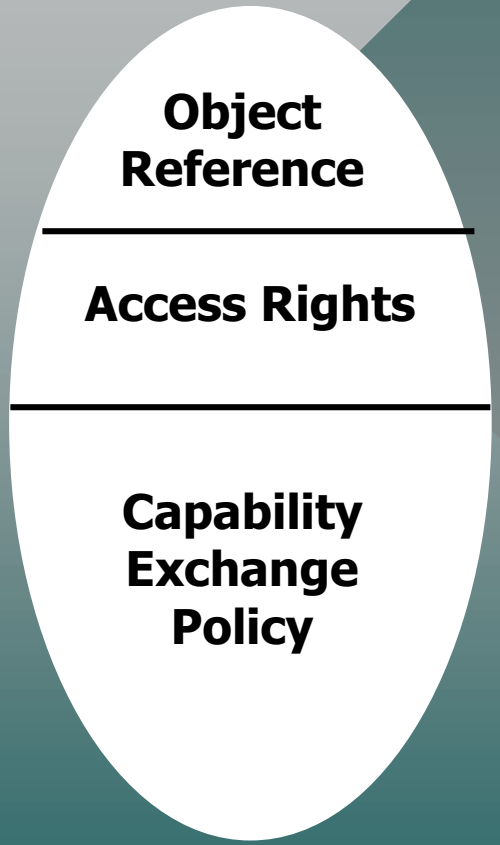
- Object references are passed as parameters
- Problem: method to pass access rights with object reference parameter *without* embedding this information in application logic.
- Solution: use interface definition language
- Interface is described independently of implementation, separating protection definition from application logic.
- Use IDL to create views which define access control policy for a capability.

Views

A view describes:

- The methods authorized by the access rights associated with the capability.
- The capabilities that must be transferred between the caller and callee along with the parameters of the methods authorized by the view.
- Note that views are recursive in nature since a view may pass another view to an invoking agent.
- Since agents are mutually suspicious, both the invoking agent and the invoked agent will define views.

View Structure



Case Study

- Agents want to print without giving distrusted print server write permission on file.
- Print server needs to be able to perform task without giving distrusted agent access to system resources.

Sample Java Interfaces

```
interface Printer_itf{
        void init();
        Job_itf run(Text text);
}
interface Text_itf{
        String read();
        void write(String s);
}
interface Job_itf{
        void stop();
}
```

This is the Java interface definition for a printer application.

These interfaces will be shared between the caller and the callee.

Security issues:

Printer doesn't want client to invoke init method.

Client doesn't want printer to invoke write method.

IDL Syntax

- Keyword *not* indicates method is disallowed for agent receiving object reference
- When an object reference is passed as a parameter in a view, *pass <view>* specifies the view to be passed with the reference.

Sample View

To implement the security restrictions desired, both agents will define views:

Requesting Agent

```
view client implements Printer_itf{
   void init();
   Job_itf run (Text_itf text pass reader);
}
view reader implements Text_itf{
   String read();
   void not write (String s);
}
```

Print Server

```
view server implements Printer_itf
   void not init();
   Job_itf run (Text_itf text);
}
```

Points to note

- Agent protection policy is defined independent of any other agent and any centralized server.
- Policy specification is detached from agent implementation.
- Each view is an additional layer, or "filter" between agents to maintain proper relationships.

Implementation of Filters

- Pre-process IDL to create classes which act as buffers between interacting agents
- Filter class will implement all methods defined within view.
- Filter class will override methods (to throw an exception) forbidden in interface definition.
- Invoking agents will be passed a reference to an instance of the *filter* class.

Java code after IDL pre-process

```
public class reader
    implements Text_itf{
    Text_itf obj;
    public reader(Text_itf o){
        obj = o;
    }
    public string read(){
        return(obj.read);
    }
    public void write (String s){
        Throw an exception!!
    }
}
```

Client

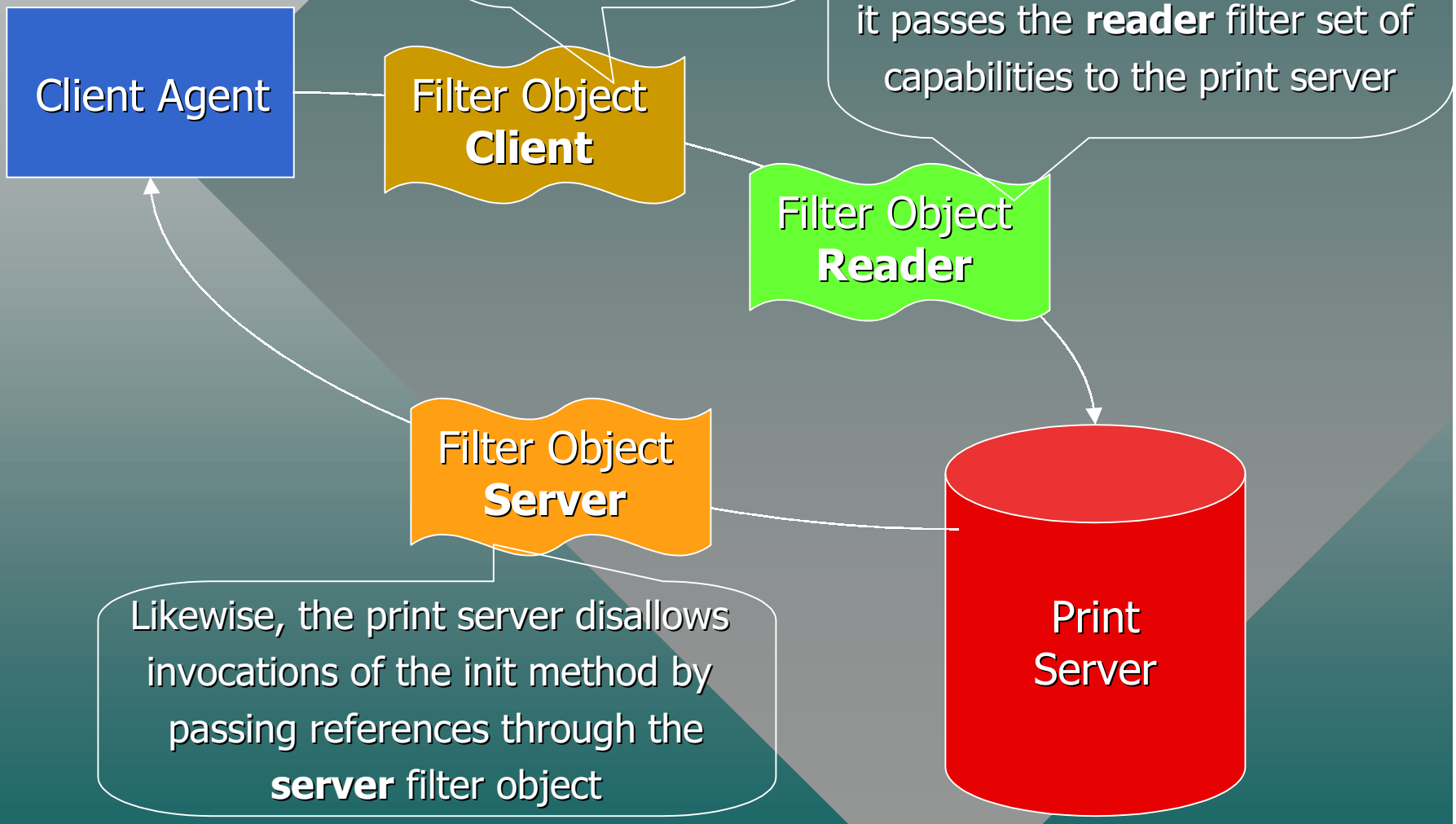
```
public class client
    implements Printer_itf{
    Printer_itf obj;
    public Printer_stub(Printer_itf o){
        obj = o;
    }
    public void init(){
        obj.init();
    }
    public Job_itf run (Text_itf text){
        reader stub = new reader(text);
        return obj.run(stub);
    }
```

Java code after IDL pre-process

Print Server

```
public class server implements Printer_itf{
    Printer_itf obj;
    public Printer_stub(Printer_itf o){
        obj = o;
    }
    public void init(){
        Exception!!
    }
    public Job_itf run(Text_itf text){
        return obj.run(text);
    }
}
```

Illustration



Related Work

- Agent TCL
  - not object-oriented
  - does not allow data sharing
- AgletIBM
  - same downside as TCL
- Telescript
  - Does provide shared objects
  - Provides authentication mechanisms
  - Access control must be managed within program

Criticism/Discussion

- Difficult to assess feasibility of implementation
- Language-independence claim is suspect
- No authentication of agents (w.i.p.)