1.Introduction

Lava is an architecture to facilitate the secure delegation of executable code. It is based on the Java system [20][26] and hence shares the concepts, terminology and mechanisms of Java. Thus, the executable content is a Java *applet*. The mechanism of applet delegation allows one host to send the applet to another host where it will execute. The big challenge is to consider the security of the entity delegating an applet as well as the security of the entity accepting this delegation. Lava provides a flexible and secure framework which can be used as the basis for a variety of distributed applications. The architecture of Lava is simple and open, and it defines the security and operational aspects of the agent delegation process.

1.1 Motivation

The benefits of distributed systems and network centric computing are well known, and these technologies have been around for a while. Agent systems are a technology to partition a system over numerous, co-operating network elements. In this project we are interested in mobile agents i.e. agents which can migrate to and execute on remote hosts. Other technologies like distributed object architectures (DOA) and messaging-oriented middleware (MOM) also provide mechanisms to distribute a system [38]. However, in both of these the agents are static. Lava combines the concepts of delegation, mobile agents and applet security to provide an flexible and secure architecture for distributed applications.

1.1.1 Distributed Network Management

A network management system (NMS) consists of managers (management applications) and managed objects controlled by management agents[1] [6]. A management application can request information from the managed objects through the management agents, make management decisions based on this information and subsequently ask the agents to perform some actions. For low bandwidth environments the traffic generated by management activities may be expensive. Furthermore the response time depends on the speed of the network(s) between the manager and the managed objects. Thus if the manager and managed objects are separated by a large, unreliable network, like the internet, then the delay may be substantial.

In a distributed environment it is possible to have a number of managers each responsible for a different domain/network (e.g. SNMPv2 [6]). Then a management application can delegate responsibility to some other manager who is closer to the managed object. If the management policy changes then the management software has to be re-configured on all these managers. Also the managers now need to communicate among themselves which requires yet another protocol. Thus, while this approach alleviates some of the problems of the centralized approach, it also introduces new constraints for communication between managers and administration of the distributed system.

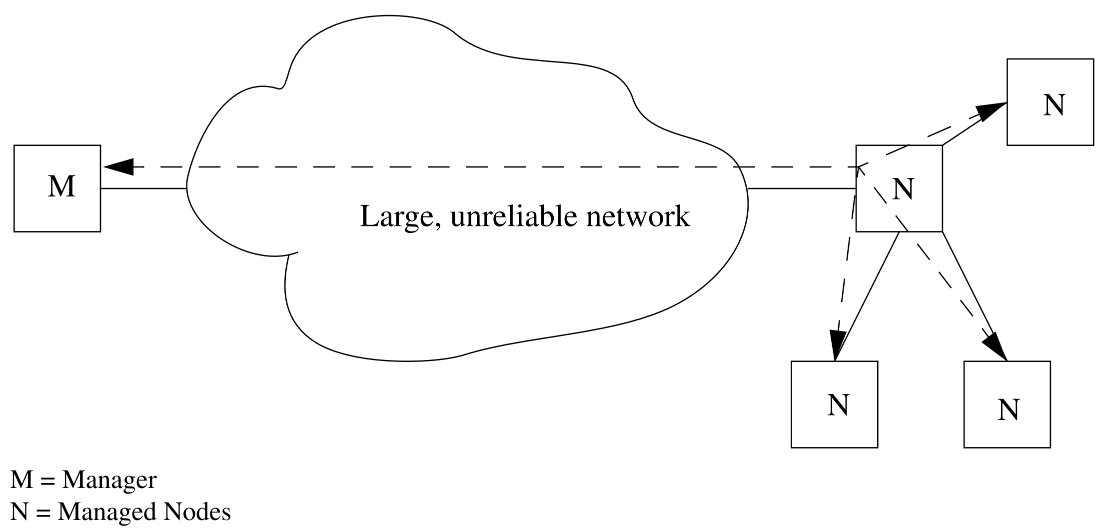


FIGURE 1.1 Network Management by a Remote Manager

---

1. The network management community uses the term "agent" to refer to a server process at the managed object which supports a network management protocol and performs some specific tasks. In order to distinguish this from delegated agents we will refer to the former as "management agent" and the later as "agent" or "Lava agent".

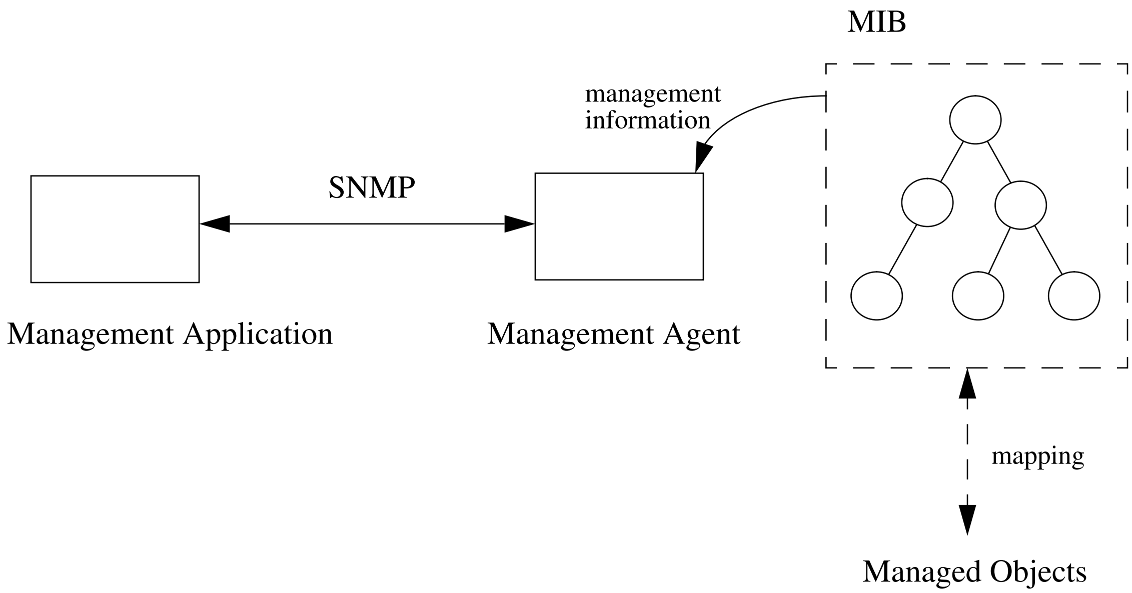


FIGURE 1.2 Network Management with SNMP.

Now consider a scenario where the manager can dynamically send an intelligent, mobile agent to the managed object which will gather information from the managed object, analyze this information and issue suitable directives to the managed hosts. The benefits of such an architecture seem to be obvious... reduced traffic, fast response time and changes in management policy do not require changes to numerous copies of management software. The benefits of the spatial (distance) and temporal (dynamic) distribution of management applications using delegation is presented in [17]. Lava provides an architecture which supports such a network management paradigm. Example applications[1] which would benefit from such a model are NMS for wireless networks and web-based NMS [10].

An SNMP based application to detect a a TCP SYN flood attack has been implemented over Lava (sec. 5.1). This is an example for the type of application which would benefit from the Lava approach.

1. [38] describes the kind of applications which would benefit from using mobile agent technology.

1.1.2 Distributed Intrusion Detection

A Distributed Intrusion Detection System (DIDS) [36] collects local intrusion events from different sites, correlates these events and decides whether the system is under attack. One practical concern in implementing a DIDS is the performance overhead of detecting and transmitting all these local events. For example, if there are N checks in the detection rule set, then N checks have to be performed for *every* incoming packet at *each* of the DIDS nodes. This represents a sizable load on the nodes and on the network. One way to save resources is to implement the detection rules as mobile applets. Then if the global DIDS suspects an attack of type X, it can send only the necessary rules to the nodes through which the attacking packets are coming. This results in isolating the intrusion detection activity to the nodes in the attack path(s) and the rest of the network resources are not wasted.

1.2 Delegation

A common scenario in distributed systems is that an entity A will request another entity B to perform some service on an object using permissions bestowed upon it by A. On its own B may not have the authority to access the object. The mechanism by which A can temporarily authorize B to perform the service is known as delegation [47].

1.2.1 Passive v/s Active delegation

A passive delegation is like a remote procedure call. This means that the delegator only tells the delegate[1] *what* needs to be done. The procedure for performing this task is present at the delegate. Passive delegation is accomplished through message passing. In an active delegation the delegator also tells the delegate *how* a task is to be accomplished. This can be achieved by sending an executable object to the delegate. Process migration and mobile agents are an example of active delegation.This is the kind of delegation in Lava.

1. In this section “delegate” is used both as a verb and as a noun. The meaning should be clear from the context.

1.2.2 Issues in delegation [47]

1. Revocation

   After a delegation is issued the delegator may want to revoke the delegation at a later time. This may be because the delegator has lost trust in the delegate or the service may not be required anymore.

2. Cascading

   The delegate may in turn delegate the task to a third entity. If this is desirable then the delegate must be capable of issuing sub-delegations.

3. Restricting

   The delegator should be able to endow a subset of its authority to the delegate to restrict the actions authorized by a delegation. For example, a delegate may be given read access to only a subset of files. The restriction may also be expressed as a function of the time period when an access is allowed, the number of accesses, etc.

4. Independence

   Once a delegation has been issued, it is desirable for the delegator and delegate to be able to operate independently of each other.

1.3 Problem Definition

In the previous sections we have looked at the motivation for a new architecture for distributed systems and the concept of delegation. Thus we can now define the problem targeted by the Lava project.

An architecture which will facilitate the delegation of a mobile, intelligent agent to a remote host and its execution in a mutually secure fashion.

The following two figures present the role of Lava as a framework for distributed systems and the problem domain of this architecture.

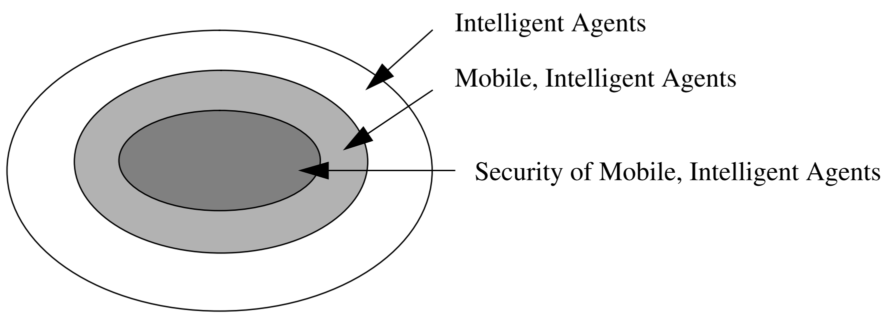


FIGURE 1.3 Problem Domain of Lava

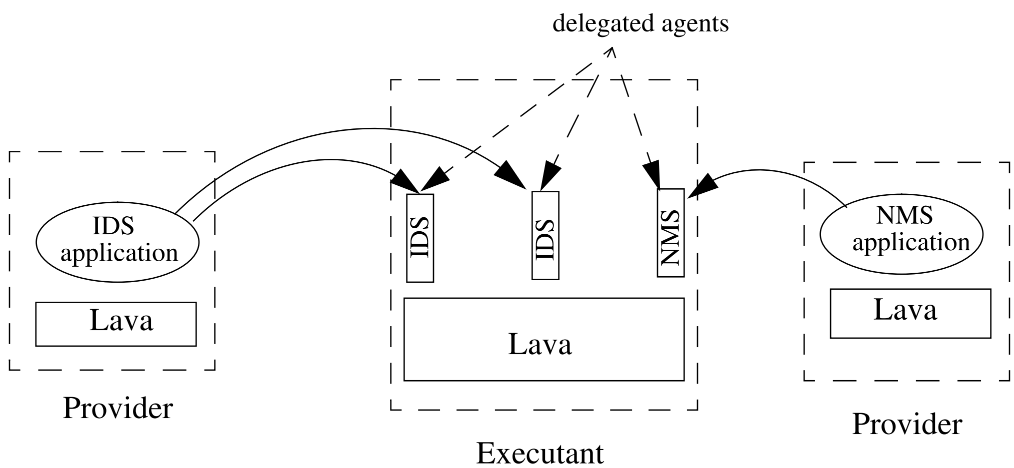


FIGURE 1.4 Lava as a platform for distributed systems.

1.4 Java[1]

1.4.1 Java Overview

Here we will only briefly discuss the features of Java which are directly related to the Lava project. Applets written in Java can be compiled to a machine independent bytecode form, which can then be downloaded by a client and executed locally. The Java language semantics and runtime checks provide some security assurances to the client system and to the downloaded applet [26]. The important (for us) features of Java are platform

1. "Java is simple, object-oriented, distributed, interpreted, robust, secure, architecture- neutral, portable, high-performance, multi-threaded and dynamic language" - JavaSoft.

independence, applet downloading and secure applet execution . On the client side, the downloaded code runs inside a context provided by a browser. The applet's bytecode is downloaded, verified, interpreted and its actions are monitored. These actions are performed by the Java Virtual Machine (JVM) implemented by the browser. Since the bytecode targets a virtual machine it can run on any system which implements this VM. The security features are discussed in (sec. 2.3.1)(sec. 2.3.4).

1.4.2 Java Security Holes

Many security flaws have been discovered in Java [15] and all of them can be fixed by modifying the implementation. However there is no guarantee that the system is truly secure (see sidebar 1.1). There are significant differences between the semantics of the Java language and the bytecode, and these differences have been exploited to thwart the security of the Java system [8]. It seems evident that these attacks indicate flaws not just in the implementation but in the architecture of Java's security system. Since Java security is enforced partially by the compiler and partially by the run time system, we insist on *both* these checks being performed for an applet to be considered safe by the Lava system.

SIDEBAR 1.1 Formal notation for Java

A team of researchers at Princeton claim in [8] that the language cannot be considered provably secure since it has neither a formal semantics nor a formal description of its type system. [12] moderates the claims made by the Princeton team. The JSRM [13] acts as a starting point for further formal modeling efforts of Java semantics[a]. [9] looks at Java's type system in the context of secure dynamic linking and [11] presents a proof of the soundness of the type system for a subset of Java.

a. This is from the discussion at JavaSoft Forum 1.1 (http://java.sun.com/forum/securityForum.html)

1.4.3 Java in Lava

The Java system provides language features and runtime mechanisms to download executable content across a network and securely execute it locally. Thus it was chosen as the base system for Lava. However the Java system could not be used as-is because of two main issues:

1. In Java the transfer mechanism is “downloading” i.e. the transfer is client initiated and the applet executes on the client. In Lava the transfer mechanism is “delegation” i.e. the transfer is initiated by a delegator and the agent is downloaded by and executes on the delegate. Here, the delegator is the client for the services provided by the delegate and the delegate is the client for the downloading process. The terms client and server become quite fuzzy, so we introduced the terminology in (sec. 1.5.1).

2. The Java security architecture utilizes a sandbox[1] model which does not provide the flexibility required by the Lava applications. The sandbox model treats all applets uniformly and the security policy is usually very restrictive [41]. Current browsers do not provide any mechanism to easily customize the security policy.

   For example, Advent used Java to implement an SNMP MIB browser [1]. However, because of the restrictive security policy of current browsers, this Advent product had to provide a back-door to bypass the security mechanism. In [5] a few security concerns have been identified regarding Advent’s approach. In general, it is unsafe to bypass the security mechanism.

   In the network management example presented in (sec. 1.1.1), an agent delegated by a manager needs to have different access rights from an agent delegated by any non-privileged user. Furthermore, the trust invested in a delegator may change and correspondingly the access rights available to that delegator’s agents may need to be changed too

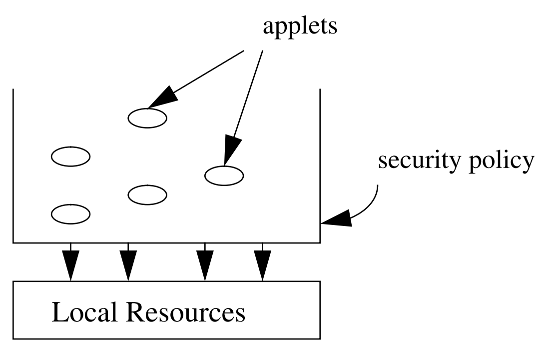


FIGURE 1.5 Conceptual view of Java’s model for access control

1. a box filled with sand for children to play in

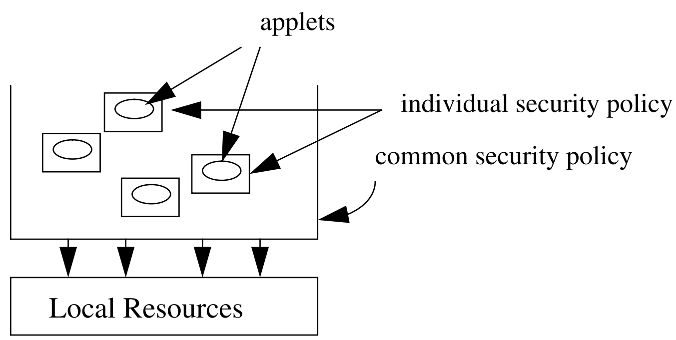


FIGURE 1.6 Conceptual view of Lava's model for access control

Thus we proposed a new architecture based on the Java system which would provide secure delegation and a provider-based security architecture with support for easily customizable security policies.

1.5 Lava Overview

1.5.1 Secure Applet Delegation

In Lava, an applet represents a mobile, intelligent agent. The mechanism by which a user can send an applet to a remote host for execution is called applet delegation.

In the Lava documents we have used the following terminology: (Fig 1.7)

Provider: The person or process who is requesting a delegation. The agent will execute with the authority of this provider. This could be an individual (e.g. Alice) or a role (e.g. remote network manager).

Executant: The host on which the delegated agent will execute. E.g. router

Lava Agent: Executable content which can be delegated by a provider to do some work at the executant. For now we can consider this to be an applet. E.g. mobile management agent.

Originator: The compiling service where the agent/applet bytecode was generated. E.g.software certification authority.

Secure delegation means that the different entities in the delegation process should be protected from each other. As in any trusted communication, the communicating entities (provider and executant) need to be protected, so that if one of them is compromised the potential damage to the other is limited. Secure channels are needed to provide some assurances about the source and integrity of the transmissions. Finally, since the delegation in Lava involves transfer of executable content, we also have to consider the threat of a malicious agent. This requires runtime monitoring and appropriate safeguards to ensure that the agent cannot misbehave i.e. violate the security of the executant. The security requirements have been presented in detail in (sec. 2.2.1).

1.5.2 Design Goals

1. Compatibility with the existing and future Java systems.

   Java is an evolving system and as the needs of the user community grow there are bound to be changes in the language and policies. In order for Lava to be compatible with future releases of Java it was necessary not to make any changes to the existing core classes, the semantics of the Java language or the bytecode. The target systems for Lava are all systems which support Java [40]. These will include at least the Solaris, Linux, and WindowsNT platforms.

2. Uniform security mechanism for agents and applets.

   Lava addresses the security concerns for the delegation of agents. However the need for a flexible and secure architecture for downloaded applets is also apparent [3][18][23]. The Lava architecture uses a uniform security model for agents and applets. Thus downloaded applets too can use the flexibility of the capability model. Furthermore, authentication is supported for both applets and agents.

3. Keep delegation and downloading independent.

This derives from the previous two considerations. The downloading and delegation processes target different application models and hence the security policy for one should not affect the other. Thus while Lava has a common architecture for applets and agents, it supports independent policies for their execution.

4. Security from downloaded content

   The additional structures needed for the delegation mechanism should not weaken the integrity of the existing security mechanism nor open up new avenues of attack by downloaded applets. Our aim is to provide a more flexible architecture while maintaining the security. Namely, it still depends on the safety of the language, the byte-code verifier and separation of name spaces provided by the Java system. Additionally Lava uses signed applets and compiler verification to enforce the security requirements of the Java system.

5. Transmission security

   The transmission of information across a network has many security issues such as privacy, authentication, non-repudiation etc. [39], which need to be enforced depending on the level of security required by the application. These can be enforced through encryption and authentication mechanisms. Lava uses the concept of a *secure* channel which is a communication channel for which these facilities are available. The specific algorithms and techniques will not be part of the Lava standard but the existence of a secure channel is required by the design.

6. Lava administration

   Lava is intended to be a simple system which will not require much administration. However it should facilitate the establishment of trust relations, policy configuration, record keeping, password management and auditing.

7. Simplicity and Usability

   The targeted users of Lava are the developers of distributed applications who will use Lava as a base architecture. The usage should not differ widely from that of Java. The architecture itself is meant to be lightweight and it will not impose too many constraints on the users. The administrator at the executant is also considered a user. However a person sitting at the machine is not considered a user and the delegation should be transparent to him.

1.5.3 Lava Architecture Overview

The design of Lava is based on two mechanisms. The primary one is that of *agent delegation*. This is concerned with the interactions between the Provider, Executant and Agent. A secondary but essential mechanism is *compiler authentication* provided by the Trusted Compiling Service (TCS), which defines the relation between the Originator and the Executant.

1.5.3.1 Delegation

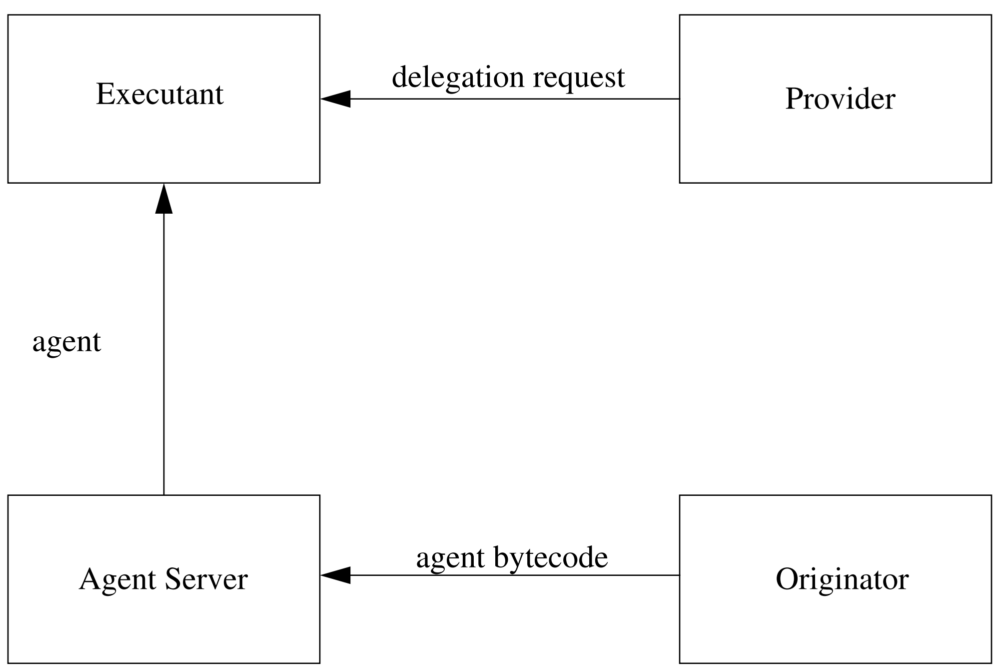


FIGURE 1.7 Agent Delegation in Lava

The main players in the delegation model are the Provider and the Executant. The provider issues a delegation by sending a message to the executant which uniquely specifies the agent which should be executed. The delegation request contains the location of the agent to be delegated, permissions granted to the agent by the provider and authentication and freshness information. If the executant decides to accept the delegation it will download the spcified agent from the Agent Server. This is just a Web server where the agent bytecode is physically located. The originator is the source generating the agent. A “trusted” originator is associated with a Trusted Compiling Service which digitally signs the code it generates.

1.5.3.2 Trusted Compiling Service (TCS)

A “trusted compiler” is a compiler such that the byte-codes produced by it are safe from the compiler point of view. The compiler will perform extensive safety checking before bytecode generation. The client submits source files for compilation. These are compiled at the server and signed by the TCS. The signature or compiler certificate identifies the TCS and the compiler it is using. Eventually the host which will execute the agent/applet can check that the bytecode was generated by a trusted compiler. It is important to note that the “trusted” attribute associated with a compiler may change over time, as implementation bugs are discovered and fixed and enhancements are made to the language.

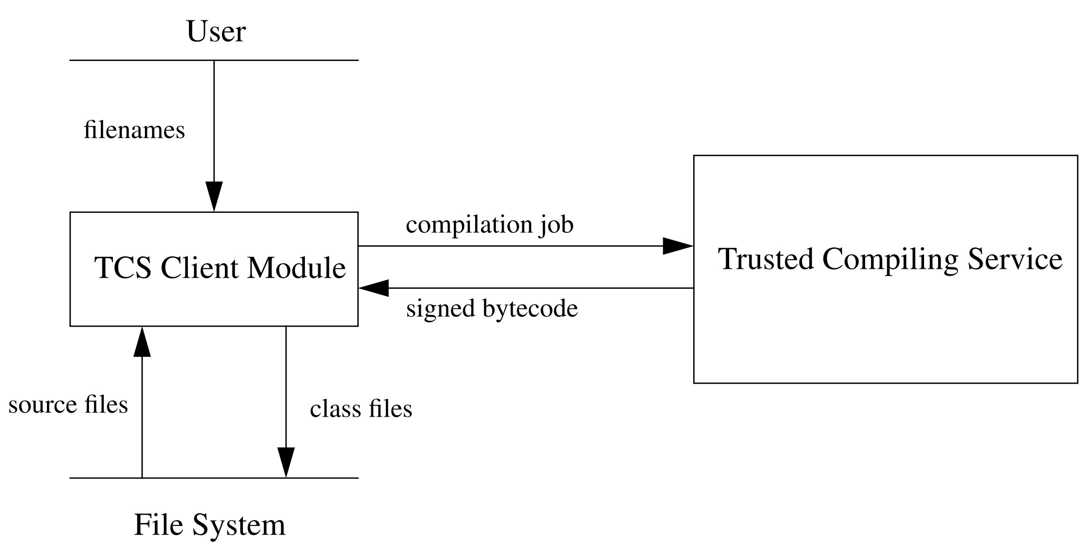


FIGURE 1.8 Trusted Compiling Service

1.6 Related Works

1. [21] defines a facility to allow arbitrary users and objects to dispatch and retract mobile agents. It provides a framework for agent transfer and is independent of any specific transfer mechanism or agent systems. Agents are CORBA objects which can execute on remote hosts. It also provides a mechanism for security of the host environment.
2. IBM's Aglet Workbench [22][48] is a continuation of the framework described in the previous paragraph. Aglets are Java objects which can migrate from one host on the Internet to another and continue execution on the new host. The primary characteristic of an aglet is that it consists of executable content as well as state information. It uses object serialization mechanisms like RMI or CORBA to accomplish this transfer of state. The Java Aglet API defines methods for creation, dispatching, message handling, retraction, cloning and disposing of the aglet.
3. [3] proposes a security model for controlling the execution of downloaded content on the internet. It is proposed for a variety of content types such as Java applets, OLE components etc. A stamp is associated with each content containing the security credentials of the content creator and the resource requirements of the content. The access rights granted to the content depend on this and are a subset of the rights of the downloading principal. A capability list is used to maintain these access rights.
4. [23] defines an architecture that provides general mechanisms for flexible control of downloaded content. It defines general mechanisms for solving the problems of content message authentication, domain derivation, domain composition and domain enforcement. These mechanisms identify where security policy is needed to flexibly control content.
5. The idea of delegation for network management originated from the MbD project [17]. In [52] the idea of delegation was used for efficient communication in mobile computing. Recently, the area of active networking [30] provides many applications which may benefit from Lava technology.

1.7 Organization of the Thesis

This introductory chapter takes a first look at the Lava architecture, its genesis and the issues involved in its design. The following chapters describe the architecture in detail and further explore the issues in detail.

Chapter 2 deals exclusively with the security aspects of Lava. It defines the security model in (sec. 2.3) and the policy structure for this model in (sec. 2.5). The trust relations between the different entities participating in the delegation process are given in (sec. 2.4.2).

Chapter 3 describes the design of Lava. This evolves from the system requirements in (sec. 1.5.2) and the security requirements in (sec. 2.2.1). The design is organized around the processes of agent delegation (sec. 3.1) and trusted compiling (sec. 3.2). First we present the subsystems and their interactions and then the classes within each of these subsystems. The effort has been to provide a simple, object oriented design. The dynamic behavior has been described through scenarios and dynamic models.

Chapter 4 describes the prototype implementation of the design. The prototype implements a subset of the features of Lava. (sec. 4.1) presents the assumptions and additional requirements imposed for the prototype development. The actual classes used and their descriptions constitute the bulk of this chapter. The API for the Lava prototype is given in Appendix III.

Chapter 5 concludes with a brief description of a project to build a realistic application over Lava. This is still work in progress. This is the first version of the Lava system and there are a number of issues and improvements which need to be considered in the future (sec. 5.3). Some of the issues discussed in this section deal with the design and some with implementation.

Since unfamiliar jargon may often cloud the important fundamental ideas, Appendix II contains a list of terms and definitions.

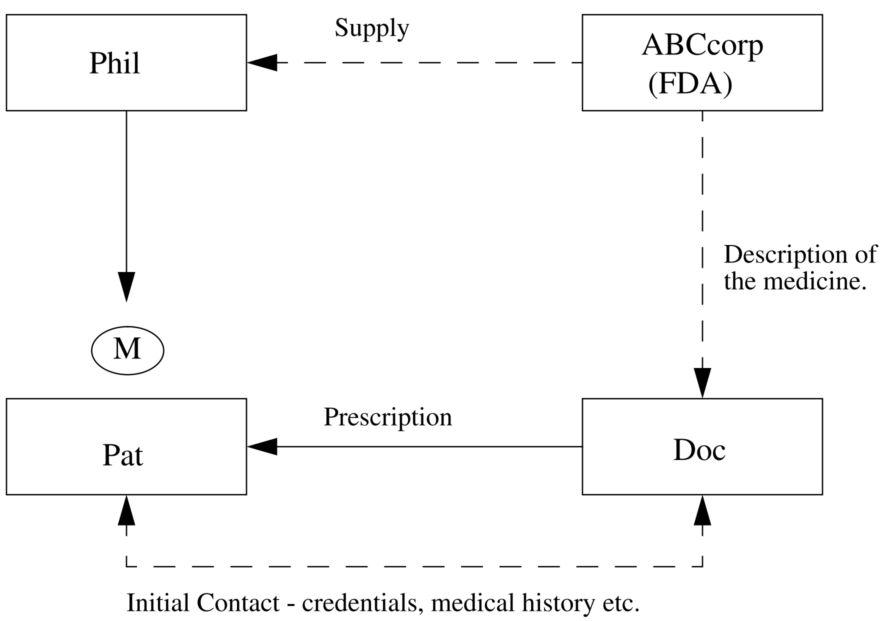
Appendix I contains a preliminary user document for someone to use the Lava prototype and a developer's document to serve as a checklist of some issues which need to be kept in mind while developing an application over Lava. These are to be used in conjunction with relevant parts of the thesis and are in no way complete or final.

2. Security Architecture

2.1 Doctor-Patient paradigm

2.1.1 Scenario

Let us consider a real life scenario which will illustrate the security concerns of Lava. Consider the situation where a doctor has sent a prescription to a patient. The patient gets the prescribed medicine from a pharmacy. Now, the patient wants to have certain guarantees about the effects of the medicine. (Fig. 2.1) shows the different entities involved and their interactions. Before we examine the security concerns of this model in any detail we need to ensure that this abstraction provides an adequate representation of Lava's architecture (Fig. 1.7).



Doc = doctor, Pat = patient, Phil = pharmacist, ABCcorp = Pharmaceutical company, M = medicine, FDA = Food and Drug Administration

FIGURE 2.1 Doctor-Patient paradigm

2.1.2 Correspondence to the Lava architecture

The doctor ‘Doc’ corresponds to the provider, patient ‘Pat’ to the executant, medicine ‘M’ to the Lava agent, pharmacy ‘Phil’ to the agent server and pharmaceutical company ‘ABCcorp’ to the originator. A comparison of (Fig. 1.7) and (Fig. 2.1) and the flow of events in (sec. 1.5.3.1) and (sec. 2.1.1) shows the similarity of the two architectures. The prescription *uniquely* specifies the medicine which is what the delegation request does for the agent, in the Lava architecture. A subtle difference is that the delegation request specifies the location of the agent while the prescription does not specify the pharmacy. However this will not affect our security analysis since the overall outcome is similar i.e. in the Doctor-Patient paradigm, the medicine has been uniquely identified and the patient does not need to go to a trusted pharmacy, and in Lava, the agent is uniquely specified and again the agent server does not need to be trusted. A mismatch between the patient and executant correspondence is the complexity of the interaction of medicine with the human body and the patient’s knowledge of these effects. Thus we make the assumption that *the effects produced by a medicine are limited to a finite domain of easily detectable events* and that *the patient can detect these events.* Another discrepancy is that the communication between the patient and doctor is usually direct while between provider and executant is over an insecure network. Thus we assume that *the doctor sends the prescription through the U.S. postal services*.

2.1.3 Analysis

Pat needs to be aware of and trust Doc’s credentials before even going for a check-up. After the check up she receives the prescription in the mail. Then she needs to ensure that the prescription is indeed sent by Doc. Pat can now go to any pharmacy and get the medicine. She should check that the medicine is the prescribed one, is sealed, unexpired and FDA[1] approved. If any one of these conditions are not satisfied she will reject the medicine. After taking the medication, Pat can monitor it’s effects. Doc needs to give her a list of the *expected effects* along with the prescription, so that she can compare them with

1. Food and Drug Administration. A governing body which certifies the “safety” of foods and drugs.

the observed effects. Finally she will report back to the doctor after some time. Another concern is that of the patient's *privacy*. Pat may not want anyone to know the content of her communication with the Doc.

2.1.4 Threat Model

Let us now introduce Eva, who wants to try and cause some harm to Pat.

1. Eva can intercept and modify or replace the communique between Doc and Pat. This can be thwarted easily by using a tamper evident seal and signing the letters to verify their origin.
2. Eva can pose as a doctor but Pat will not trust her since she will not have the required credentials. Suppose Eva is a doctor, even then Pat will not go to Eva unless she has a trust association with Eva. Finally, suppose that Eva is Pat's ENT specialist and hence has a trusted association. However, if she tries to prescribe medication which affects the heart then Pat should detect it and recognize Eva as a possible attacker.
3. Suppose Eva compromises the pharmacy. However when she tries to give Pat some fake medicine, it will not pass the checks described in (sec. 2.1.3).
4. Eva decides to produce her own medicine and supply it to the Phil. However without proper approval, Pat will not buy that medicine. In fact Doc would run his own checks on the credentials of the medicine manufacturer before prescribing the medicine.
5. Eva can try and retransmit Doc's prescription so that Pat will take more of the medicine than necessary. Hence the prescription needs to be dated.
6. Eva can break into Doc's office and try and steal Pat's medical records. Doc needs to have sufficient office security (e.g. locks, alarms, guard etc.) to counter this threat.

The requirements and threats discussed in the preceding sections illustrate the concerns of Lava. Consequently, Lava's security architecture derives from this paradigm.

2.2 Scope

The security of a system depends on the security of each of its components and the interactions between them. The Lava system addresses the security issues directly related to and arising out of the agent delegation process. Since Lava is based upon the Java system its scope is limited by the scope of the Java system. For example, an agent in Lava is a Java applet and the extent and granularity of the Lava security manager checks are limited by the JVM. The scope of the Lava security architecture is expressed by the services it expects from other components (assumptions) and by the facilities it provides to other components (requirements).

2.2.1 Requirements

The security architecture of Lava has to consider the threats to the provider delegating the agent as well as to the executant on which the agent resides. Thus the term “client” refers to both the provider as well as the executant. If one of them is malicious or compromised then the security mechanism should be able to contain the potential damage to the other. The agent may also be malicious and both the provider and executant should be protected against it.

1. Each of the entities should have individual security mechanisms to provide some measure of security from outsiders. This is popularly known as a Trusted Computing Base (TCB). Lava assumes the existence and reliability of the TCB (sec. 2.2.2).

2. Verification of the source, integrity and freshness of the agent.

   Verify that the agent was generated from a trust-worthy source and hasn’t been modified.

3. Verification of the source , integrity and freshness of the delegation request

   The executant must be able to verify that a delegation request is indeed from the provider from which it purports to come and that it has not been tampered with on the way.

4. Flexible architecture to support individualized policies.

The permissions an agent obtains on the executant machine depends on the provider who authorized it. Furthermore the provider too has its own security constraints. This means that the effective security policy depends on the Executant-Provider-Agent combination. The architecture should be flexible enough to allow such customization of policy.

5. Control the actions of the agent.

   Once the agent has been downloaded to the executant machine, the runtime system should be able to detect and filter all attempts by the agent to access local resources.

6. Non-interference

   An executant might have a number of agents and applets running on it simultaneously. These should be shielded from each other.

7. Reliability of the runtime security mechanism.

   The runtime should provide a secure, restricted environment for the agent to execute in. The runtime security along with the safety features of the language and requirements 1 and 2 should ensure that an agent can not bypass the security mechanism.

8. Security monitoring and reporting.

   If security violations do occur then these should be detected and reported to a managing authority. This may result in a change in the trust-worthiness of some of the principals.

9. [14] discusses the security concerns of mobile agent systems. The primary issue is that of state alteration[1]. However Java applets and hence Lava agents, are *stateless*. Thus we do not need to consider the class of security concerns arising out of migration of state.

These requirements arise out of a consideration of the interactions between the entities in the applet delegation process and can be categorized according to these interactions.

1. As a mobile agent migrates from one host to another, changes may occur in its state and these may cause a security hazard for the new host it has migrated to. This is applicable only for stateful agents.

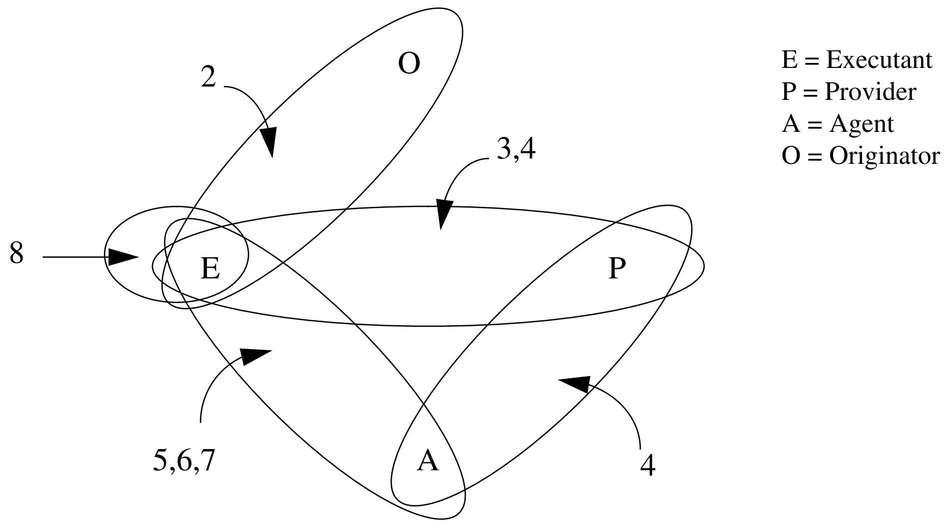


FIGURE 2.2 Mapping security requirements to entity interactions

2.2.2 Assumptions

1. Language safety

   The safety features of the language should assure that downloaded code cannot directly access the local system. We depend on the safety features provided by Java (sec. 2.3.1).

2. Security of client and system software.

   Security of passwords, configuration files and trusted software is a prerequisite for the security of any system including Lava. This is a system level security issue.

3. Secure channels

   We assume that underlying protocols provide secure channels using appropriate encryption and authentication mechanisms (sec. 2.3.3).

4. The executable content can access the local system only through a well defined set of commands (system interface).

5. Sound judgement of site administrator.

The site administrator at the executant needs to decide which providers and compilers can be trusted and the access rights to be granted to each provider. The provider needs to decide which executant to be delegated to and the permissions to be granted to the delegated agent. These are all application dependent issues.

2.3 Security Model

Lava has a multi-level security model.

- Language level security
- Transmission security
- Admission Control
- Runtime security
- Resource access monitoring

2.3.1 Language Level security

We depend on the security of the Java language.

> “ The important features of the language from a security standpoint are the use of access control for variables and methods within classes, the safety of the type system, the lack of pointers as a language data type, the use of garbage collection (automatic memory deallocation), and the use of packages with distinct name spaces.“ [4]

While this ensures that a program written in Java will not harm the target system, an attacker could use a modified Java compiler or some other method (e.g. Jasmin [29], AppletMagic [46]) to construct an applet which could violate the security of the target system. This means that the Java runtime must also perform a number of checks in order to maintain the security of the system.

2.3.2 Transmission security

This is an important part of any secure distributed system. Lava itself provides authentication of the originator of an agent and the provider in a delegation. However it builds this on top of secure channels for which it depends on lower level protocols like Kerberos [39][45] or SSL [31]. Secure-HTTP [33] could be a candidate protocol for providing security services between the agent server and the executant.

2.3.3 Admission Control

When a provider sends a delegation request to an executant, the executant may accept the delegation or may deny it. Possible criteria for denying a delegation request are:

- The executant distrusts the provider
- The executant does not have sufficient resources
- The resources required by the agent are not available (by looking at the intention list).[1]
- The provider is not permitted to access the resources required by the agent.[1]

2.3.4 Runtime Security

This is the security provided by the Java runtime.

> “ Aside from simple format checks, the bytecode verifier checks:
>
> - that it doesn’t forge pointers
> - that it doesn’t violate access restrictions
> - that it accesses objects as what they are
> - that it calls methods with appropriate arguments of the appropriate type
> - that there are no stack overflows

1. This option has been rejected (sec. 3.1.1).

> Along with checking the validity of the bytecode, the ClassLoader has the responsibility of creating a name space for downloaded code, and resolving the names of classes referenced by the downloaded code. " [4]

However there have been numerous attacks on the Java system which have shown that if the compiler level safety is violated then the bytecode verifier cannot always compensate for it [8]. Lava uses a Trusted Compiling Service protocol and signed applets to guarantee that compile time checks have been performed.

2.3.5 Resource Access Monitoring

Lava has a flexible, capability-based, bipartite security manager which monitors an agent's resource accesses.

Flexible: The protection domains for agents are dynamically generated. Lava can support a number of policies and it is easy to switch and configure these policies.

Capability-based: It uses a capability list structure [34] to store the access rights of an agent. Thus each delegated agent can have different permissions.

Bipartite: The specific access rights to be granted to an agent are decided in part by the executant on which the agent will run and in part by the provider on whose authority the agent will execute.

2.4 Lava Security Architecture

The basis of Lava's security is a set of trust relations between the various principals involved in the delegation and a model to grant authority based on these relations. A trust relation defines the degree of trust between two entities. Once a trust relation is established an entity may obtain special permissions to access another entity's resources. This is supported by Lava's capability based security model. First let us look at some of the threats and issues which must be considered by Lava, then we'll present the trust relations and capability model in more detail.

2.4.1 Threats / Security Considerations

The security of a system is dependent, to different degrees, upon the security of the individual components and the trust associations between them. We need to consider the effects of the failure of the individual components and associations on the rest of the system. Many of the threats enumerated here derive from the Doctor-Patient scenario (sec. 2.1.4). These have been referred to via examples wherever applicable.

1. Delegator attack

   An attacker may try to delegate an agent which will access the executant's private resources. The executant should be able to restrict the access rights of this agent. Furthermore an executant has different levels of trust in different providers. Thus an access by an agent under the authority of one provider may be legal while the same access under the authority of another provider may be illegal. E.g. (sec. 2.1.4-2)

2. Evil agent or Evil agent server.

   An agent may try to perform some illegal accesses in the background. This is an attack on the executant as well as the provider. The provider should be able to restrict the actions performed by the agent under its authority. Furthermore if the agent does succeed in its evil intentions then this violation should be detected.

   If the agent server is compromised then an attacker may be able to modify the agent or its signature. E.g. (sec. 2.1.4-3)

3. Evil Originator

   The originator can use a modified compiler to avoid compile time checks and open up ways by which the agent can bypass the security mechanism of the executant. Some mechanism is required to verify that the agent was compiled using a safe compiler. E.g. (sec. 2.1.4-4). The security of Lava depends on the integrity of the Trusted Compiling Service. If this is violated then the security of the system may be temporarily broken.

4. Privacy

   An attacker may modify the agent while it is being downloaded or he may eavesdrop on the communication between the provider and the executant. E.g. (sec. 2.1.4-1)

5. Evil executant

   If the executant is bad then it can send back false or misleading information to the provider. It can also access a delegated agent's private information. Another possibility is that if the executant's runtime VM has bugs then the results may not be dependable.

6. Replay

   An attacker may store and replay a delegation request. Consider a delegation issued by a network manager to a router for an agent which will flush the routing tables. An attacker replaying this request could cause the network to fail. E.g. (sec. 2.1.4-5)

7. Agent Interaction

   Java keeps applets from different locations isolated by maintaining different name spaces. In Lava, agents belonging to different *delegations* need to be isolated from each other. Since different delegations may use the same agents we cannot directly use the name spaces used by Java. Furthermore, an agent may have the permission to use system resources and make persistent changes to the state of the executant which means that agents can indirectly affect each other.

8. Result verification

   The results that an agent generates open up a possible avenue for attacking the provider by returning false information. However I consider this a higher level problem since its resolution requires knowledge of the security requirements of the application doing the delegation.

2.4.2 Trust Relations

Lava defines protocols to establish trust relations between

- the Executant and the Provider
- the Executant and the Originator

- the Provider and the Agent

2.4.2.1 Executant(E) and Provider(P)

The establishment of this trust relation requires lower level security protocols to provide a secure communication channel between the principals.

- There are *multiple levels of trust* between P and E viz.

1. P is unknown[1]

   Then the default access control policy (sec. 2.5.1) is used.

2. P is known and mistrusted[2]

   A delegation issued by P is flagged so that some additional protective measures can be installed. For example, the admission control policy may reject an agent delegated by a bad provider. This serves as a notification service.

3. P is known and trusted

   An agent delegated by P will execute under the policy enforced by the associated capability.

- The *communication protocol* consists of 3 messages:

1. Delegation Request
2. Revocation
3. Reporting

Delegation Request

- P identifies E and a secure channel is set up between the two machines.
- P sends a delegation request to E

  `[Delegate, Pid, password, Nonce(P), URL`$_A$`, IntentionList]`

1. P is known by E if E has a capability associated with P. Otherwise it is unknown.
2. E can designate a provider as bad or mistrusted.

- E decides whether to accept or reject the delegation.

  `[Reject, URL`$_A$ `Nonce(P+1)]`

- If E decides to accept the delegation, then it downloads the agent and checks its signature. If the integrity of the signature cannot be verified or if the executant does not have a trust relation with the Originator / TCS then the delegation fails and the Provider is notified of this.

  `[Failure, message, Nonce(P+1)]`

- An accept message is sent back and the agent is then allowed to run within the constraints imposed jointly by the local administrator and by the remote provider.

  `[Accept, Nonce(P+1), RuntimeVersion]`

- The provider checks whether the RuntimeVersion is trusted and if it is, then it can believe the results returned by the agent.

In each message the first field is the message type identifier e.g. `Delegate`. `Pid` identifies the provider. The `password` is a shared secret between the provider and the executant. This does not need to be unique to that P-E combination, but if the provider uses the same password on more than one executant then there is a greater threat from a compromised executant. The `Nonce` ensures freshness [39]. `URL`$_A$ gives the location of the agent to be delegated. The `Intention List` specifies the expected behavior of the agent.

Revocation

- P sends an ABORT directive to E.

  `[Abort, Pid, password, URL`$_A$`, Nonce (P+2)]`

  E verifies the password and kills the agents delegated by P from the specified URL.

Reporting

1. P can ask for a report.

- P sends a REPORT directive to E.

```
[Report, Pid, password, URL_A,Nonce(P+2)]
```

- E verifies the password, fetches the results from the agent and sends them back to P.

2. E can send a report.

- E gets the results from the agent and sends them back to P as a report. This can be initiated by the agent.

An important issue for consideration is that the access rights granted to a provider may change at anytime. This may be a result of the administrative policy which is independent of Lava's delegation mechanism. Thus the access rights may change even during a delegation.

2.4.2.2 Executant(E) and Originator(O)

As mentioned earlier, the security of the Java system strongly depends on the integrity and correctness of the compiler used to generate the bytecode. A trustworthy Originator represents a trustworthy compiler version. The trustworthiness of a particular compiler version may change over time as bugs are discovered and fixed. The originator has an association with a Trusted Compiling Service (TCS). This generates a compiler certificate along with the bytecode and attaches it to the end of each class file[1].

```
CompilerCert = [TCSid, E_TCS(Version, TimeStamp, MAC(applet))]
```

Thus the downloaded applet consists of bytecode segment(s) + CompilerCert(s)

The message authentication code (MAC) can be generated using any message digest algorithm like MD5, SHA etc. [39]. The receiver of the applet should know which algorithm is being used. An asymmetric encryption scheme like RSA [39] could be used for scalability.

The receiver of this applet/agent will perform the following checks:

1. Check if the TCS is trusted. If it is then decrypt the CompilerCert.

1. A bytecode representation of a Class.

2. Check the timestamp. This may or may not be used by the client.
3. Check the compiler version.
4. Generate the MAC for the applet and verify the result.

If the applet passes all four checks then the executant can allow it to run.

2.4.2.3 Provider (P) and Agent (A)

The trust relation between the Provider and the Agent is not a direct communication between the two but instead is built on the relation between the Executant and the Provider. When P delegates an agent A, it does not know whether the code associated with the URL will do what it claims to do. So when P requests a delegation it also sends an IntentionList which is a profile of what P expects A to do. P then depends on the security of E to guarantee that the agent works according to this profile. Thus Lava does not provide any measures to assure that the agent is generating the correct results. However it provides both E and P with constructs to limit the damage that A can cause if it is pretending to be some other agent.

2.4.3 Events and Actions

This section summarizes the types of events or messages that a participating host can receive and the actions the security mechanism must perform.

TABLE 2.1 Events and Actions - Executant

Message	Actions
Delegation message	Verify integrity of message. Verify provider password.
Download class file	Check if the originator is known/trusted Verify integrity of downloaded segment Verify the signature on the class. Check if the compiler used is trust-worthy.

TABLE 2.2 Events and Actions - Provider

Message	Actions
Report	Verify source of the report Check if executant has a trust-worthy runtime (Lava version).

2.4.4 Capability-based Security Model

A capability associates actions with objects. The objects in our model are system resources such as files, directories, ports, systems procedures, java packages etc. The actions are the permissions to access these resources.

The executant associates providers with capabilities. The capability for a particular provider defines its access rights on the executant machine. These permissions are decided by the local administrator depending on the extent to which the provider is trusted by that executant. When an agent is delegated by a provider it inherits the authority of that provider and its access rights are defined by the corresponding capability. During agent execution, the Lava runtime system intercepts all local resource accesses and filters them through the capability for the agent attempting the access. If the access is denied by the capability then it throws a security exception.

This model allows a fine-grained, provider oriented, flexible security policy to be enforced. Furthermore the policies can be easily configured and modified.

The capability specification has the following format:[1]

```
capability ::= BEGIN_CAPABILITY + permission_list +
END_CAPABILITY

permission_list ::= permission_tuple + permission_list
```

1. The terms in upper case are terminal symbols.

```
permission_tuple ::= NULL | tag + resource-rights_pair

tag ::= resource_type | trust_level

resource-rights_pair ::= resource_name + access_rights

resource_type ::= FILE | ACCEPT_CONNECTION | CONNECT_TO |
EXEC | PROPERTY

trust_level = BAD
```

The resource_name and access_rights will depend on the tag (resource_type).

TABLE 2.3 Capability Specification

Tag	**Resource Name**	**Access Rights**
FILE	file name	READ | WRITE
ACCEPT_CONNECTION	host-port pair	NULL[a]
CONNECT_TO	host-port pair	NULL
EXEC	command name	NULL
PROPERTY	java property name	NULL
BAD	NULL	true | false

a. NULL = Absent

Design decision

The capability specification *does not use negative access rights.* The rationale for this is the expressive power of the capability will not be increased by adding negative rights. However the efficiency may degrade since an extra field will have to be checked. Initially a capability starts with the minimal access rights and these can be increased by explicitly allowing specific resource accesses. If we need a policy 'p' which says *"do not allow subject 'sub' to perform operation 'op' on object 'obj'*[1]" then this can be expressed by not mentioning the resource 'obj' in the capability specification for 'sub' with the access right containing 'op'.

1. `allow_p(sub,obj,op) = false` (sec. 2.5.1)

2.4.5 Failure Scenarios

The Lava system consists of a number of interacting entities. In this section we describe the effects of these entities being compromised and the safeguards provided by the Lava architecture to protect the other entities from the compromised one.

1. Agent Server gets compromised

   This threat is described in (sec. 2.4.1). The provider based security policy and the provider specified intention list together provide a restricted domain within which the agent must execute. The message digest and encrypted signature together prevent an attacker from modifying the agent or its signature. However the evil agent could still do some limited amount of damage within the scope of the permissions granted to it by the executant and provider. Thus in a critical application the safety of the agent server does become an important requirement for a safe application.

2. Provider gets compromised

   This can be disastrous if the provider is highly trusted. If someone gets the administrator's password then they can make any change to the system. This is similar to the password problem in any authentication system. Thus a compromised provider can cause an evil agent to be delegated to an executant and enjoy a high degree of trust. The auditing and monitoring mechanism can help detect the violation after it has occurred but there is no real time protection in Lava for such a situation.

3. Executant gets compromised

   This could mean that a delegated agent might be replaced or its results and operation corrupted. The final effect is that the results returned by it may be incorrect. Lava does not provide any result verification facilities. It is up to the application to decide whether to use the results. An assumption of the delegation process is that the delegating entity will delegate to a trusted executant.

2.5 Security Policy

"*A security policy can be defined as a finite set of rules which delimit the accesses that can be made to objects*." - A. Jones and W. Wulf [2]

Lava's security policy is defined by the Security Manager [4][51] and the Capability classes.

2.5.1 Policy Structure

The Security Manager module defines a global security policy which is imposed upon all applets and agents. A provider specific policy is imposed if the global policy allows it. If the provider is not recognized a default policy is applied. Furthermore the provider can also specify constraints under which the agent will execute (sec. 2.6.2). Thus the actual access rights are a function of the delegation instance. The authentication policy allows applets to be signed or unsigned while *all* agents need to be signed by a TCS.

Let the boolean valued function `allow_p (sub,obj,op)` represent the query "*Can the subject* `'sub'` *legally perform the operation* `'op'` *on object* `'obj'` *under policy* `'p'` *?*". In our case the subjects are applets/agents, objects are system resources and operations are the kind of access which can be done on the object. Lava defines the following policies:

- $P_{overall}$ = This is the overall Lava policy and it is composed of the P_{global}, $P_{provider}$ and $P_{intention\_list}$ policies.
- P_{global} = Global Lava policy
- $P_{provider}$ = Provider-specific policy at the executant.
- $P_{intention\text{-}list}$ = The constraints which the provider would like to place on the agent.

Then,

$$\texttt{allow\_P}_{\texttt{overall}}\ \texttt{(sub,obj,op)} = \texttt{allow\_P}_{\texttt{global}}\ \texttt{(sub,obj,op)}\quad \vee$$

$(\texttt{allow\_P}_{\texttt{provider}}\texttt{(sub,obj,op)}\ \wedge$

$\texttt{allow\_P}_{\texttt{intention-list}}\texttt{(sub,obj,op))}$

where V and Λ are boolean OR and AND operators respectively

See (sec. 4.3) for more details on the Lava policy.

2.5.2 System Resources

The system resources available to an applet/agent via Security Manager checks are enumerated below. These are the same as in the Java system [20]

1. Files - read or write
2. Ports - listen, accept or connect.
3. System properties - getting or setting.
4. Dynamic Libraries - linking
5. System calls - exec
6. Java packages - access or define
7. Graphics display
8. Threads

The following resources are not checked by the Security Manager. These are protected by the language specification and/or the runtime system.

1. Random memory - cannot access memory directly
2. Input devices - access key-strokes and mouse clicks only when the applets window has been selected.
3. Output devices - cannot directly access output devices.

The public API of the security manager class for Lava (Appendix III) lists all the checks which are carried out. A subset of these resource accesses can be permitted by a capability specification (sec. 2.4.4).

2.6 Notes

2.6.1 Agent Interaction

One of the security features of Java is that applets from different locations could not interact with each other and for Lava too we wish to maintain this security property. In Lava, agents can access system resources and effect persistent changes on the executant. Thus it may be possible for agents from different hosts to interact and indirectly access resources which are not directly accessible by them.

Such an attack possibility could arise out of user error (site-administrator and provider). Therefore the security of the system is very dependent on the individual security policies on the executant.

2.6.2 Intention List

This construct enables the provider to restrict the actions which an agent may perform on his/her behalf. It describes the *expected* profile of the agent being delegated.

For example, suppose a provider P with administrative privileges on the executant E wants to delegate an agent A which will get some information from an SNMP agent running on some host S close to the executant host. If the agent thus delegated is a trojan horse which reads the password file and communicates these results to some other host X, the security of E and the trust of P will be violated. This attack can not be prevented by E's security policy alone since P has administrative rights which will allow A to use any local resource (say). In such a case P can specify, through the intention list, that it expects A to only open a connection to the S. Thus when A tries to access the password file or connect to X, the access will be denied and a security violation will be signalled.

3. System Design

This chapter details the design of the Lava system. The design goals are given in (sec. 1.5.2). A high level view of the Lava architecture is given in (sec. 1.5.3). The rest of this chapter continues from there. The design is presented in two parts viz. the agent delegation process and the trusted compiling service. Each section contains the design considerations specific to its role.

3.1 Delegation

3.1.1 Design Considerations

1. (sec. 1.2.2) discusses the issues to be considered for the delegation process. These will be handled in Lava as follows:

   (a) Revocation

   The provider should be able to ask the executant to halt the execution of the delegated agent. The Provider - Executant protocol (sec. 2.4.2.1) has a revocation message which directs the executant to abort the delegation and halt the execution of the delegated agent. We introduce the concept of ownership of delegations here. Only the provider owning a delegation is allowed to abort it. The owner of a delegation is the provider which issued the delegation request.

   (b) Cascading

   The Lava architecture does not explicitly allow cascaded delegations. However this can be achieved indirectly by the application running over Lava. For example if a delegated agent wanted to issue a delegation then it simply has to send a delegation request to the target executant with the requisite parameters. In order to do this the agent needs to know the identification information for the provider.

   (c) Restricting

The *Intention List* part of the delegation request (sec. 2.4.2.1) is a construct by which the provider can restrict the permissions which can be granted to the delegated agent on the executant.

(d) Independence

In general the provider and executant processes operate independently and if one of them crashes then the other can still continue execution.

However, the level of interaction is an issue which is within the control of the application using Lava. In the network management example (sec. 1.1.1), the Lava solution would help reduce the communication overhead. Hence in such a case a heavy interaction between the provider and executant would void the benefits of using Lava.

2. An agent is basically an applet and it does not need to know anything about whether it is invoked through delegation or downloading or who its provider (if any) is. Thus all communication between the provider and the delegated agent must be via some communication interface on the executant which permits a limited level of interaction between the provider and the agent.

3. (sec. 2.3.3) discusses the issue of admission control for agents and proposes some criteria that may be used to accept or reject a delegation. One possibility was to check whether the resource accesses given in the intention list were permitted for the provider and reject the delegation if this check failed. In retrospect we believe that such an approach would unduly restrict the flexibility of the system. Agents are intelligent entities and may be programmed to use alternative resources if those which they desire are not permitted. For instance, if an agent wants to use port 4050 for communicating with some other agent and if this port is not usable for a particular executant host then the agent may be able to select another port to perform its work. Thus, in order for this agent to function, only a subset of its intended resources need to be accessible. Thus the approach in Lava is to merge the intention list with the capability associated with the provider, on the executant, and allow the agent to execute under this joint policy (sec. 2.5.1).

3.1.2 Agent Architecture

The structure of a Lava agent is very similar to that of a Java applet. Additionally it makes some internal state visible to the external world.

3.1.2.1 Class Description

1. Agent

   This represents the executable content which is delegated and executed remotely by the Lava architecture. It has an "*is a*" relation to a Java applet since Lava is based on Java. Additionally, it has a data structure to store the results of its execution and a way to make these results available to its context.

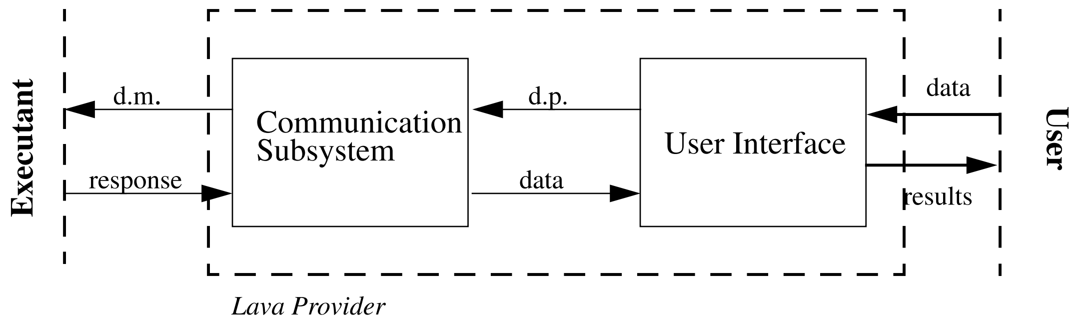
3.1.2.2 HTML Tag

The executant needs a way to distinguish between applets and agents. For this we have proposed a new HTML tag. Essentially it is identical to an `applet` tag [16] except that the `applet` keyword is replaced by the `agent` keyword. The format of the agent tag is given below.

```
<AGENT
     [CODEBASE = agent-url]
     CODE = agent-filename
     WIDTH = pixel-width
     HEIGHT = pixel-height
     [ALT = alternate-text]
     [NAME = agent-name]
     [ALIGN = alignment]
     [VSPACE = vertical-pixel-space]
     [HSPACE = horizontal-pixel-space]>
[<PARAM NAME = parameter VALUE = value>]
```

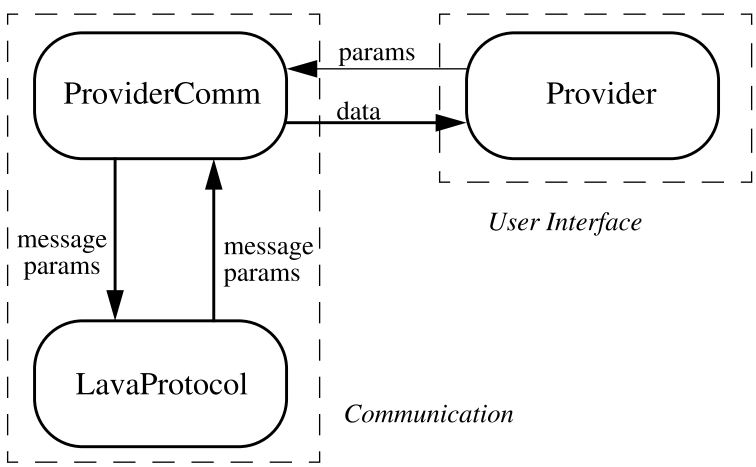
```
....

</AGENT>
```

3.1.3 Provider Architecture



d.p. = delegation parameters
d.m. = delegation message
results = results of the delegation (formatted)
response = reply from the executant

FIGURE 3.1 Provider Subsystems



params : delegation parameters
message params : A messageparameters (fields, type) derived from the delegation parameters
data : the results of the delegation returned from the executant

FIGURE 3.2 Provider Classes

3.1.3.1 Subsystems

1. Communication Subsystem

   Handles the communication between the Provider and the Executant. It receives delegation parameters from the delegator and constructs delegation messages from them.

2. User Interface

   Provides an interface through which the delegator can make delegation requests and receive the results of a delegation.

3.1.3.2 Class Descriptions

1. The Provider is a light-weight class which represents the delegating authority. It communicates with the executant using the Communication module. It gets the user id, password, agent URL and permissions from the User or high level application and sends the information to the executant. Then it waits for the executant to send back some results and relays them to the user.

2. The ProviderComm class is the communication module which understands the delegation specific issues of the communication such as the meaning of the delegation parameters and the fields of a delegation message. The communication protocol between the provider and executant consists of messages to request a delegation, revoke an ongoing delegation and ask for a report from the agent.

3. The LavaProtocol class encapsulates protocol details such as message formats and constants. It hides the details of the communication channel.

3.1.4 Executant Architecture

3.1.4.1 Subsystems

1. Communication subsystem

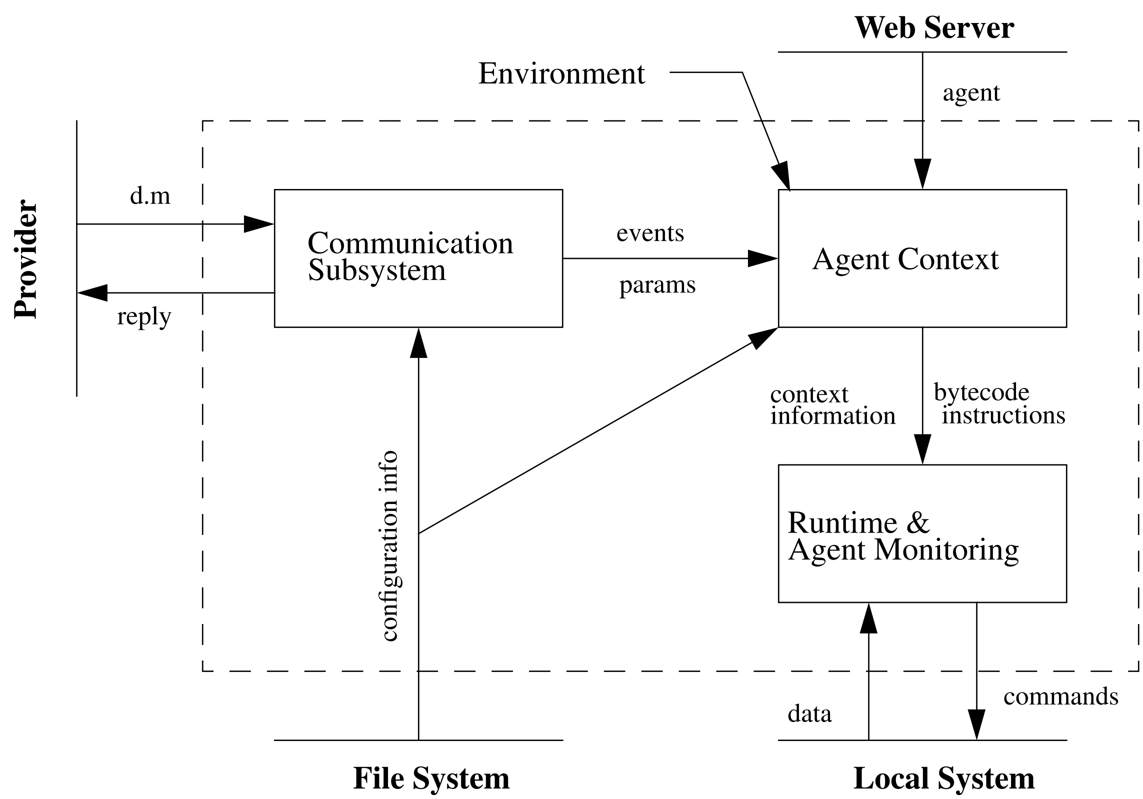
   Handles the communication between the Provider and the Executant. It receives a delegation request, verifies its validity and sends the location of the agent and identity of the provider to the Agent Context subsystem.

2. Agent Context subsystem

   Downloads the agent and associates a context with the agent. The context includes a name space, protection domain and an interface to the external world which contains a visual element.

3. Runtime and Security subsystem

   The runtime consists of the virtual machine on which the agent executes. The security subsystem monitors and controls the agent's resource accesses.



d.m. = delegation message from the provider
reply = agent report and status messages from the executant
data = access rights information and other data to be passed to the agent.
commands = system calls to access local resources.
events = directives to the agent context to start, stop etc. the agent.
params = delegation parameters
bytecode instructions = the instructions of the agent program
context information = info required to filter the agent's resource accesses
agent = the delegated entity, consisting of signed class files
environment = executant or Lava administration processes.

FIGURE 3.3 High Level Executant Architecture

3.1.4.2 Class Descriptions

The delegation activity on the executant consists of two distinct processes viz. agent delegation and agent execution.

Agent Delegation

The Provider sends a delegation request to the Executant. The executant processes the request, downloads and verifies the agent and finally starts the agent. It may need to send back a report generated by the agent.

The class descriptions are given below in the *order of invocation.*

1. AgentDaemon
   This is a continuously running server that waits on a port for delegation requests. When it receives a request it creates an AgentComm object to handle the communication. On start-up it initializes the data structures needed for the security checks.

2. AgentComm

   This provides the communication service between the provider and executant. The delegation request and commands are interpreted and appropriate events are sent to the AgentViewer. Admission control for delegation messages (sec. 2.3.3) is done here. This also establishes a channel to send back results to the provider.

3. AgentViewer

   This parses the HTML document, creates AgentViewerPanels in which the agents will be displayed (one per agent tag). It sends events to the AgentViewerPanel to start, initialize, stop and dispose the agent. This provides an interface through which the executant can interact with the agent.

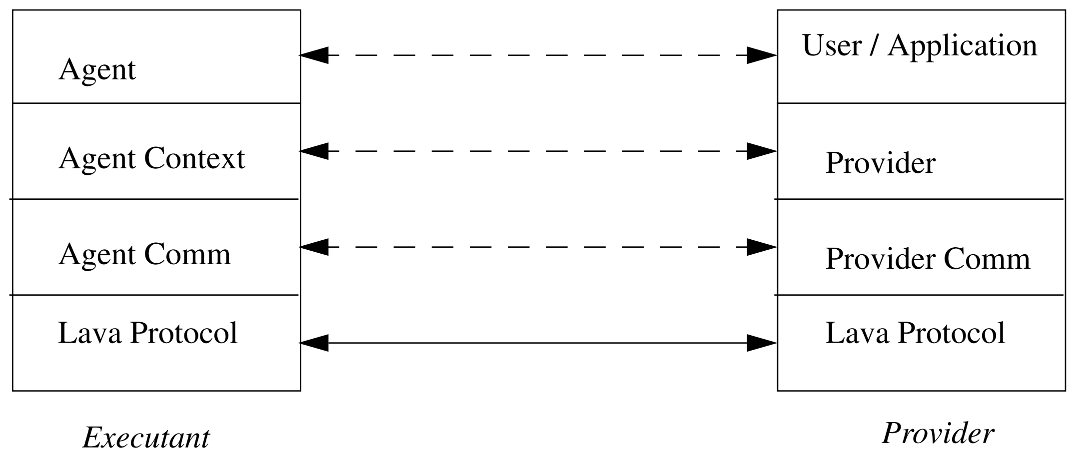


FIGURE 3.4 Layered communication architecture

4. AgentViewerPanel

   This handles events for starting, initializing, stopping the agent and reporting the agent's results. It creates an AgentClassLoader which actually downloads the agent from the specified URL. It also provides the visual part of the agent context.

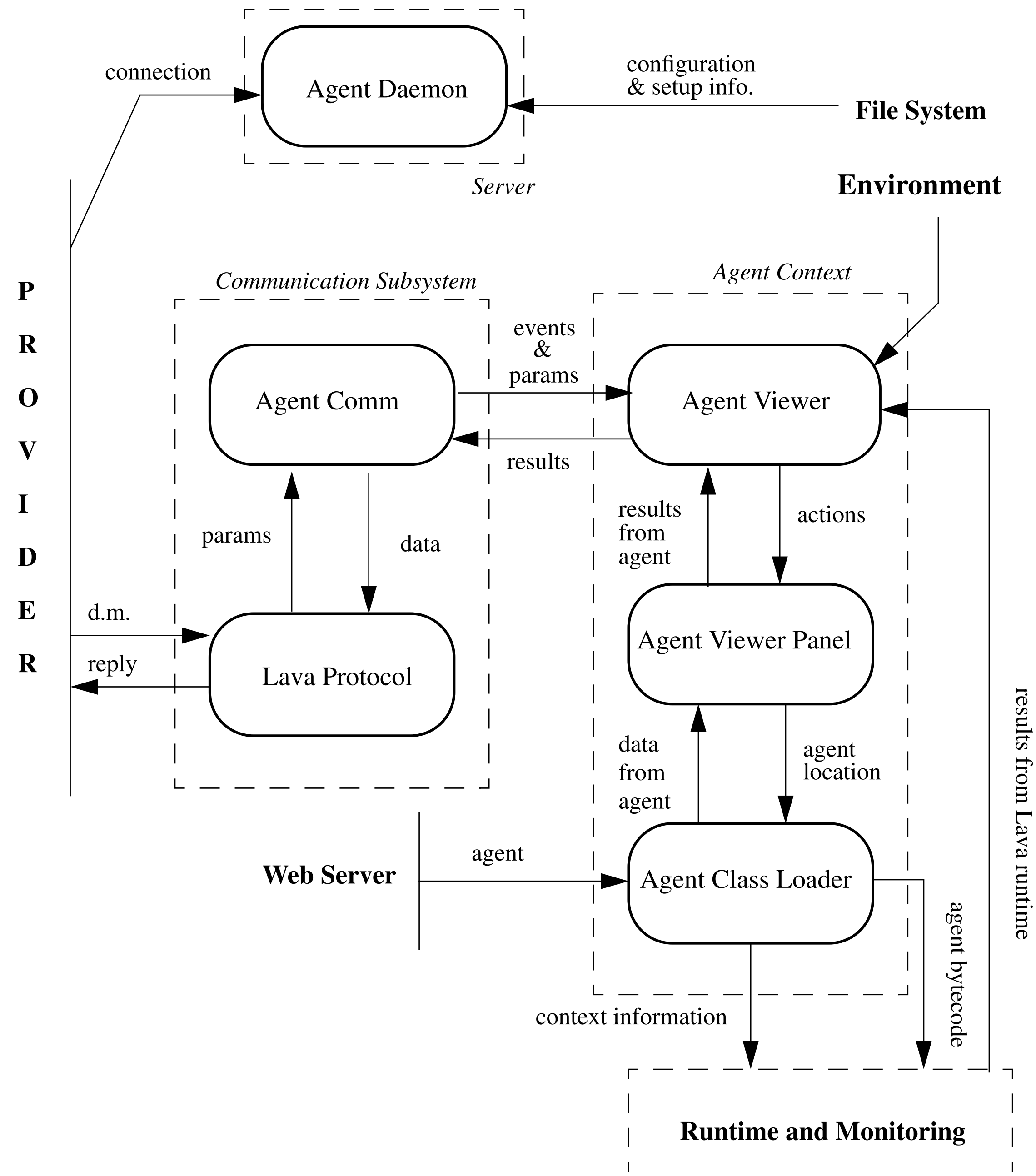
5. AgentClassLoader

   This downloads agent bytecode, verifies the signature and constructs the agent classes from it. It creates a name space containing the agent.

6. Delegation[1]

   This represents the delegation instance on the executant. It encapsulates the provider id, intention list and the communication stream on which the results of the delegation are to be sent.

1. These classes are used by the executant classes described above. They are not in the order of invocation.



params = delegation parameters obtained from the delegation messages.

actions = agent operations like start, stop, report etc. to be performed in response to external requests and commands.

agent location = The URL of the agent

unsigned classes = generated from the signed class after authentication.

FIGURE 3.5 Agent Delegation

Agent Execution

The agent executes on the virtual machine. The security manager monitors and filters its actions.

The class descriptions are given below *in the order of invocation*.

7. AgentSecurity

   This class represents the security manager. It filters all the attempts made by the agent to access local resources. It will utilize a uniform model to support both applet and agent security policies (sec. 2.4.4).

8. Capability

   This class contains the access rights information for a Provider on that executant. The format of a capability specification is given in (sec. 2.4.4). This implements the individualized access control policy.

We will use the Java Virtual Machine [26] as is. It executes the bytecode safely and when a resource access attempt is made by the agent it invokes the security manager.

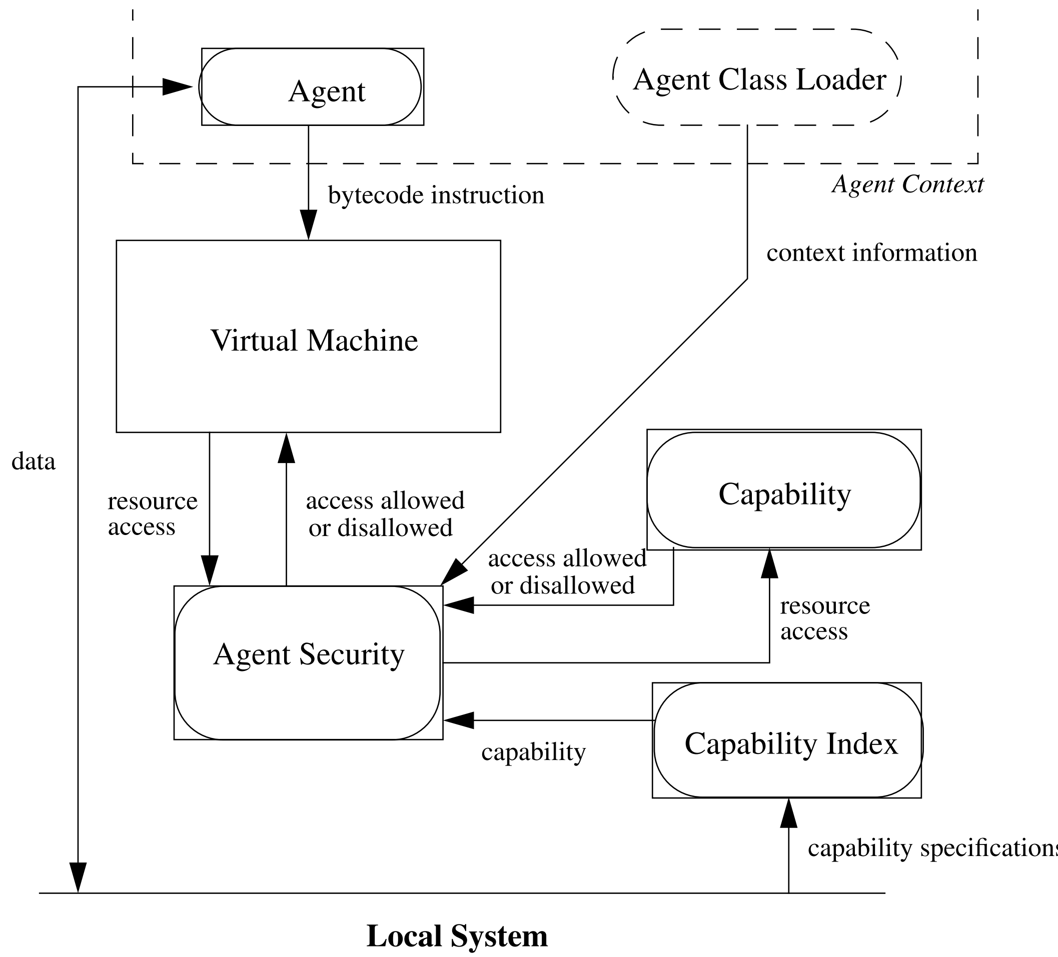


FIGURE 3.6 Agent Execution

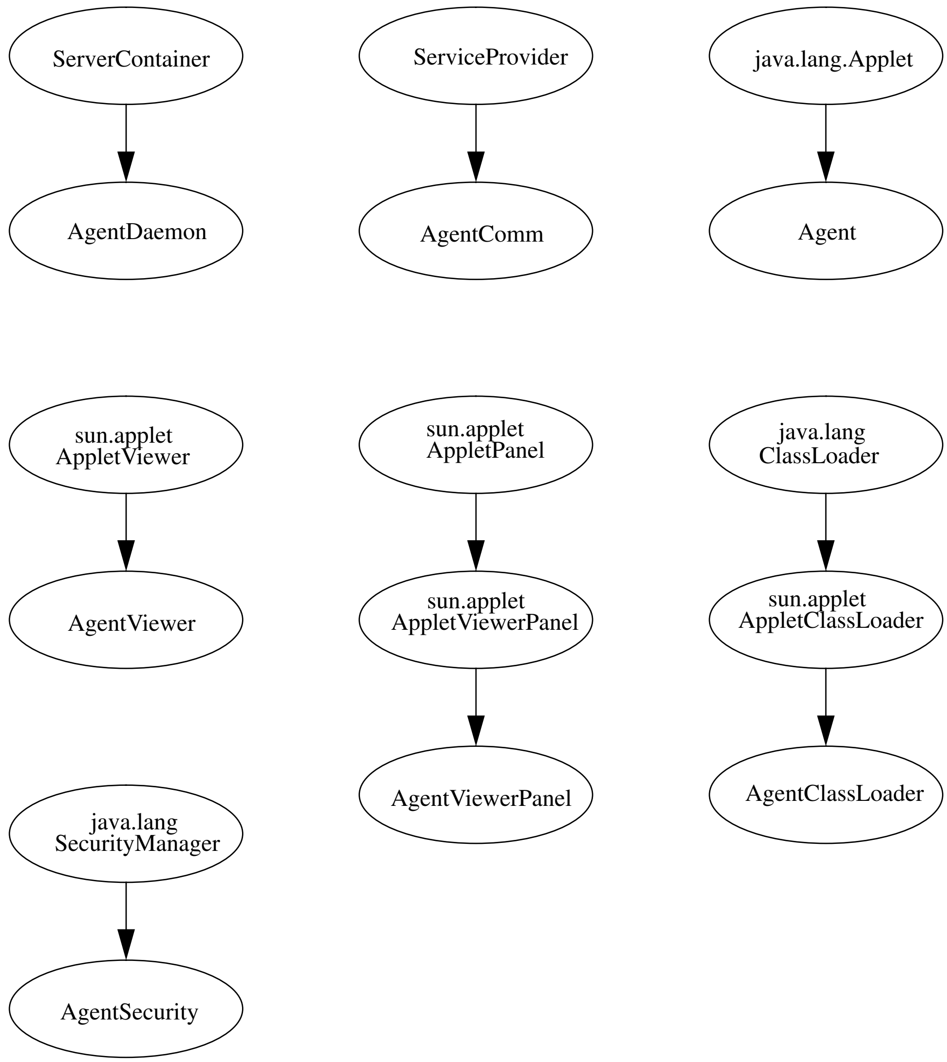


FIGURE 3.7 Class Hierarchy for Delegation classes

3.1.5 Scenarios

The first scenario describes the steps in the delegation process while the other scenarios deal with specific cases arising with multiple agent delegations.

1. A successful delegation

   The provider sends a delegation request to an executant. The request contains the provider's id, password, URL of the agent and its intention list. The executant accepts the request and a communication channel is established among these two entities. An AgentComm instance is created to handle all communication through this channel. In the runtime environment on the executant, each agent is contained within a distinct agent context (AgentViewerPanel). Note that the same bytecode delegated by different providers are considered to be *distinct* agents. This separation is achieved by the AgentClassLoader. The agent executes on the virtual machine. Whenever an instruction tries to access a local resource, the runtime traps it and invokes the SecurityManager (common for all applets & agents). The security manager filters the access through the capability associated with that delegation and if the capability allows it, then the access is permitted, else a security violation is signalled. The capability mentioned here is formed by merging the permissions of the provider's capability on the executant (from the CapabilityIndex) and the intention list sent by the provider. This is stored in the agent context and made available to the runtime when required. The delegation can be terminated by the *provider*, by the *executant* or *automatically* when the agent has completed its work. During the execution of the agent the provider can ask for reports from the agent. The AgentViewerPanel gets the results from the agent and sends them back to the provider.

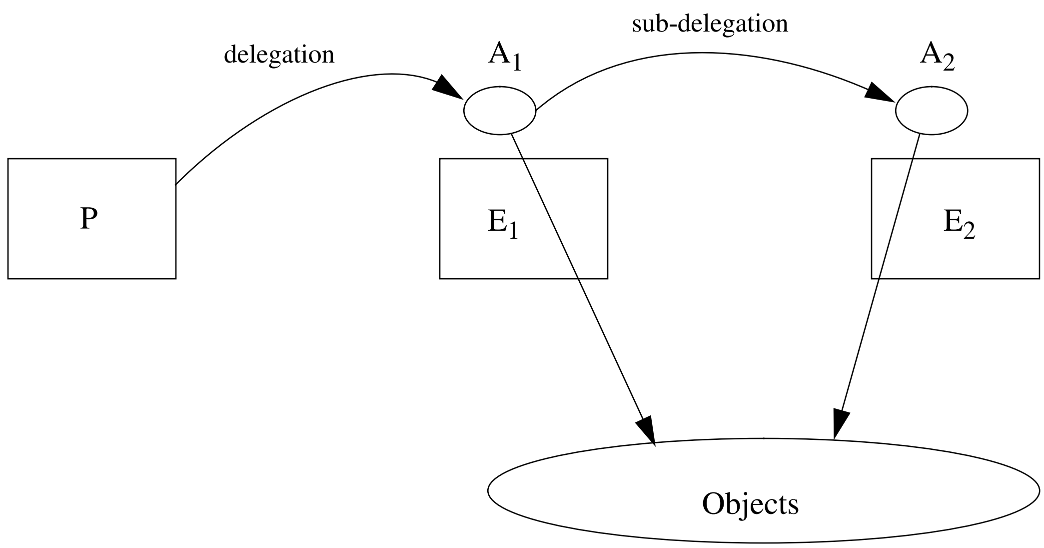
2. Suppose provider P delegates an agent A[1] on an executant E. Then another provider Q delegates the same agent A on the same executant. The two agents must be executed with separated name spaces i.e. no shared address space.

3. Suppose provider P delegates an agent A on an executant E. Then it delegates A on E again. These two agents will belong to the same name space and can share data. If these two delegations have different access rights because of different intention lists then the later one's permissions will be used. Thus a provider can change the permissions bestowed on an agent it owns without having to recall it. Note that the first agent's classes are used for the subsequent identical delegation. A delegation is identified by the provider and the location of the agents.

4. If P delegates two different agents, A and B, on E then they share the same name space iff both are from the same location otherwise they are kept in separate name spaces.

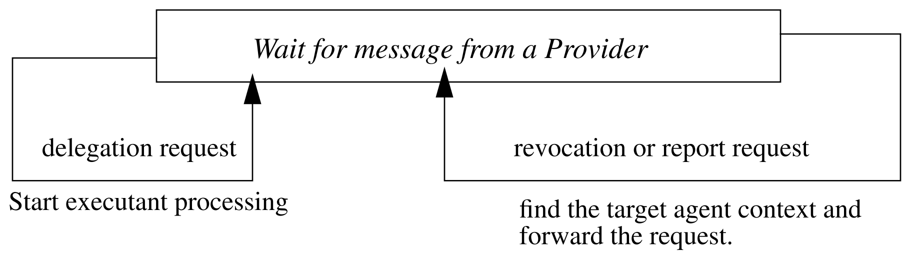
1. the agent is identified by its name and location

5. Consider a delegated agent which monitors activities on a subnet. For this it may want to issue sub-delegations to other executants. Fig. 3.8 shows how cascading can be achieved by an application running on Lava. The provider P delegates agent A1 on executant E1. A1 can access the objects with the permissions of P. A1 then delegates A2 on E2 with the authority of P. A subtle but very important distinction is that the delegating entity is A1, not E1. Hence A2 can access the objects with the permissions of P.

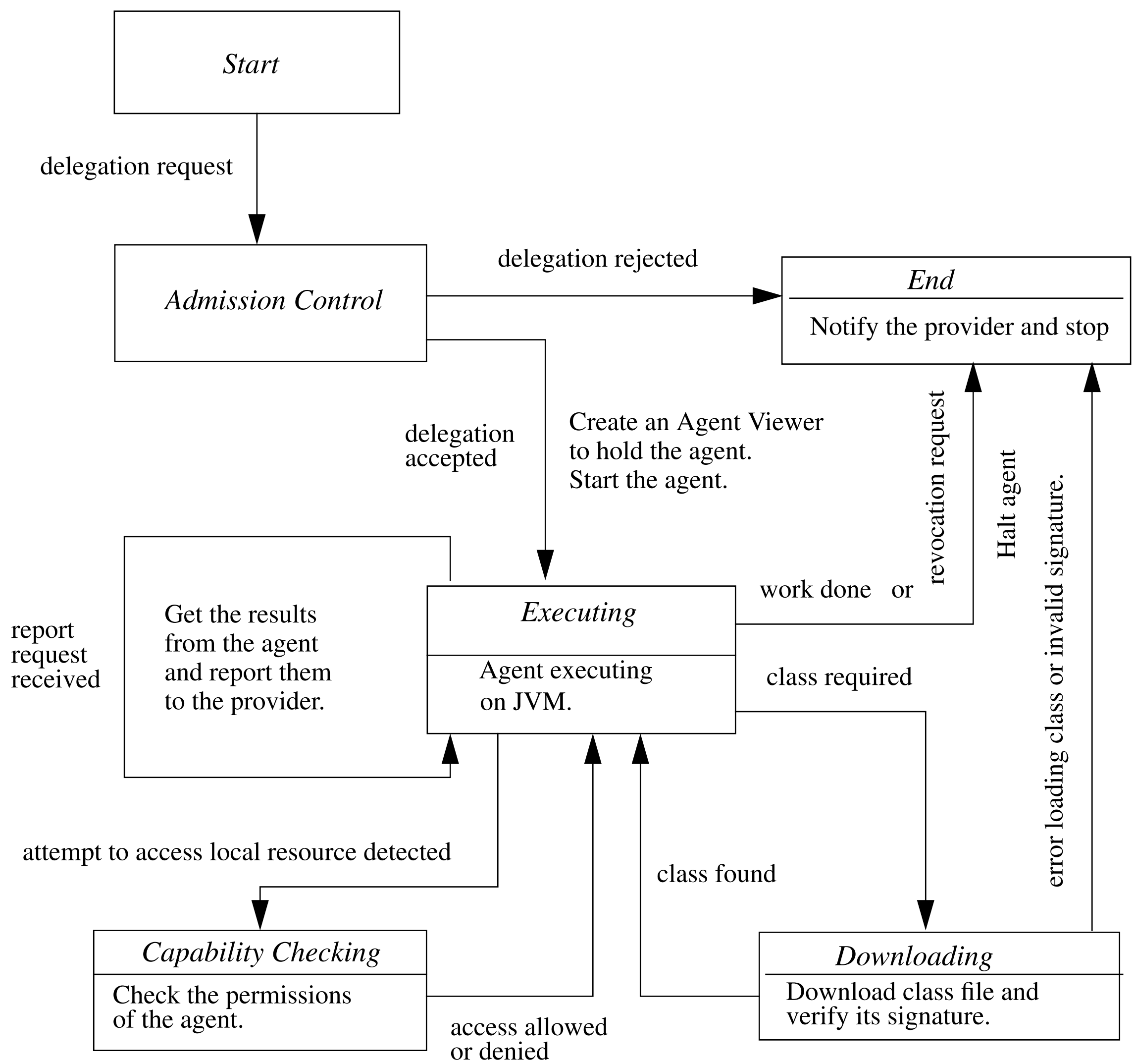


P = Provider
E_1, E_2 = Executants
A_1, A_2 = Lava Agents
Objects = abstractions of system resources

FIGURE 3.8 Cascading delegation in Lava



(a) Executant Server

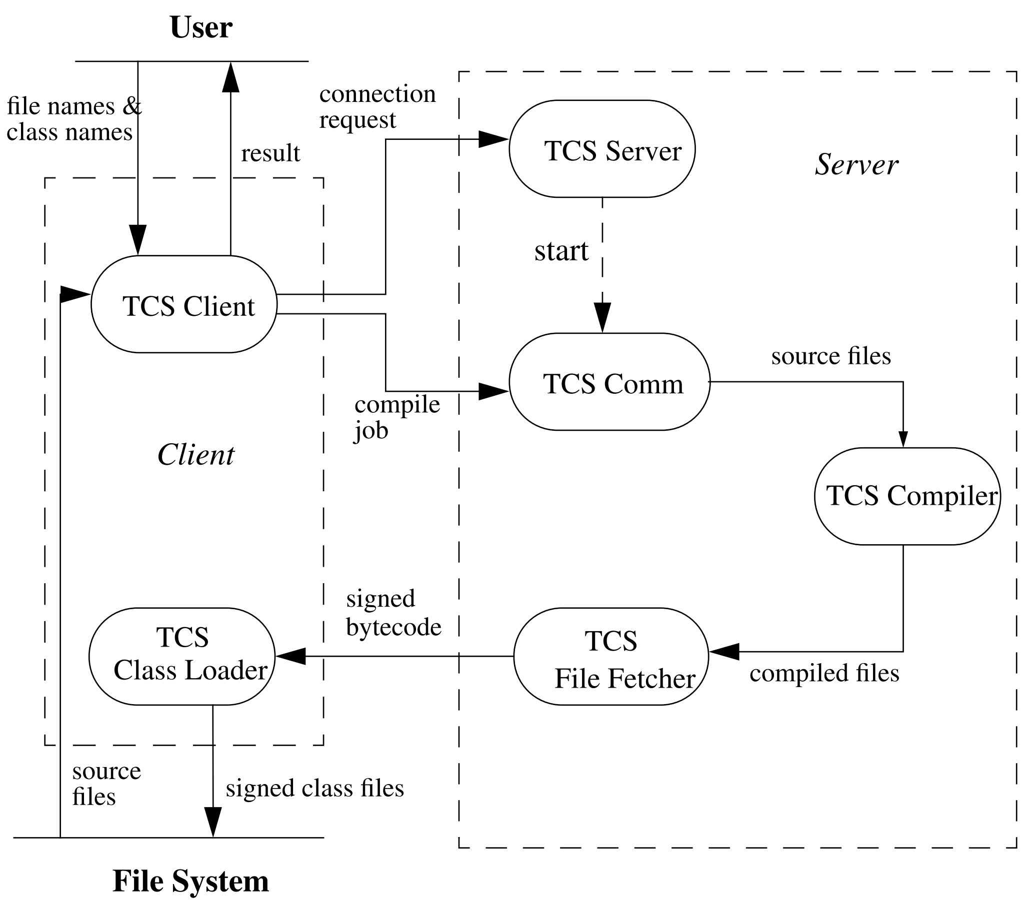


(b) Executant Processing

FIGURE 3.9 Dynamic Model - Executant

3.2 Trusted Compiling Service (TCS)

As discussed earlier (sec. 2.2.2), Lava depends on the safety of the Java language system and in order to maintain this security we need the assurance that the agent has been compiled by a trustworthy compiler. The TCS provides this assurance.



signed class files = Java `.class` files with a compiler certificate at the end
source files = Java `.java` files[1]
compiled files = Java `.class` files
file names = Names of the source files which are to be compiled in a job.
class names = Names of the classes to be downloaded.

1. These may also be source files in some other language (sec. 3.2.1)

FIGURE 3.10 Trusted Compiling Service

3.2.1 Design Considerations

1. The signature (compiler certificate) appended to the class files needs to be secure to protect against tampering. This could be accomplished using some message digest scheme.

2. The communication needs to be authenticated so that the client knows that the compiled code is coming from the TCS server which it trusts. We assume this will be handled by the secure channel protocol. Note that the encryption of the compiled code is not required.

3. We envision that the TCS Servers will be well known and standardized compilers will be used. The compilers are not restricted to only Java compilers. Bytecode can be generated from other languages too [46].

4. Scalability and key distribution issues have not been considered in this design.

3.2.2 TCS Server

The TCS server is well known and trustworthy to the extent that it uses a safe compiler and the signatures generated by it are secure. It consists of the following classes:

1. TCSServer: This is a continuously available server. It accepts connections from clients and spawns TCSComm instances to handle each of these connections.

2. TCSComm: This handles the communication between the server and the client. It interprets the compilation request, receives the source files from the client and invokes the TCSCompiler to compile them.

3. TCSCompiler: This compiles the source files and generates the class files.

4. TCSFileFetcher: This fetches the compiled bytecode file and sends them to the client. A compiler certificate (sec. 2.4.2.2) is attached to these class files before transmission.

3.2.3 TCS Client

The client uses the TCS to get the source code compiled and certified.

5. TCSClient: It queries the user for the names of the source files to be compiled, sends these files to the TCS server and invokes the TCSClassLoader to download the compiled files.
6. TCSClassLoader: This downloads `.class` files from the remote TCS server and saves them to the local file system.

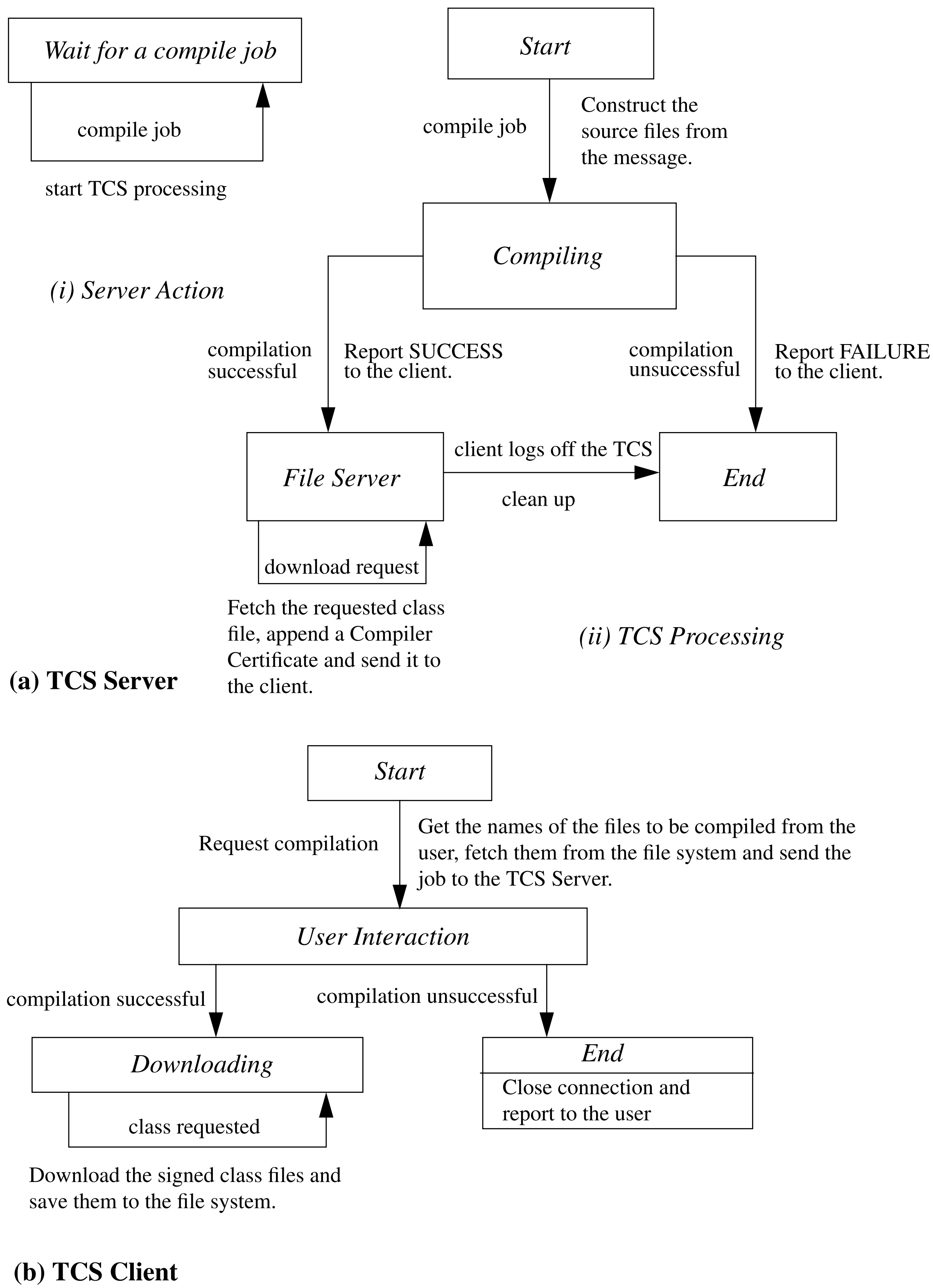


FIGURE 3.11 TCS Dynamic Model

3.3 Lava Administration

Lava's administrative requirements are mentioned in (sec. 1.5.2). The administrative functions will include provider side functions, which will be used by the distributed application built over Lava, and executant side functions targeted at the site administrator.

Executant

- Log files to keep a record of all delegations and their status.
- *Effect list* to record the actual effects of a delegated agent.
- API for configuring secure channels.
- API for configuring the provider specific policies.
- API for tracing through log files and effect lists.
- Mechanism to detect the violation of Lava's security policy.

Provider

- API to set up and maintain access classes and their associated capabilities.

3.4 Design Verification

This section describes how all the design goals from (sec. 1.5.2) have been satisfied. (sec. 3.1.1) describes how the delegation issues in (sec. 1.2) have been handled.

1. Compatibility with the existing and future Java systems.
   No changes are proposed to the core classes. This can be seen from the class descriptions. A number of classes from Sun's JDK will be re-used in the prototype to cut development time. This does not affect the compatibility. However the final implementation of Lava should use its own implementation of the these classes.

   We believe that this is an important factor since tying the design of Lava too closely to the implementation of Java would inhibit the growth of both systems. Some of the other recent proposals for flexible security management in Java violate this requirement.

One recent proposal [3] advocates a capability based security model for Java applets which is conceptually very similar to that of Lava. However it requires changes to be made to the Java `java.lang.Thread` class in order to facilitate the dynamic enforcement of applet capabilities.

Another recent work [23] uses digital signatures and scope privileges to provide a flexible security policy for Java. However it annotates the stack frames generated by Java method calls.

2. Uniform security mechanism for agents and applets.

   The Security Manager uses a capability-based model which allows provider specific policies to be imposed on agents. The default capability policy is used for applets (sec. 2.3.5). A Lava agent is an applet and hence we propose a uniform mechanism to handle both.

3. Keep delegation and downloading independent.

   Lava uses Java's downloading mechanism to fetch the agent from the web server. The architecture for delegation consists of three subsystems as described in (sec. 3.1.4.1). The architecture for downloading consists of the Agent Context and Runtime subsystems. Furthermore the delegation process includes the concept of a provider, which is absent in downloading, and the security policy depends on the provider. If the provider is not found then the default policy is used. This means that in order for applets and agents to co-exist they must have the same default policy. This conforms to our design since an agent is an applet (sec. 3.1.2.1). Thus applet downloading can co-exist with the delegation process.

   Lava supports signed agents and applets. Hence a signature recognition and verification procedure should be invoked during downloading. For backward compatibility unsigned applets are also supported. The algorithm and protocol for handling signed applets and agents are independent and may be different.

4. Security

Chapter 2 describes the security architecture. (sec. 2.4.1) describes the threats and security considerations of the Lava system. (sec. 2.4.5) describes Lava's safeguards against various failures and attacks. The following table lists the features of Lava which satisfy the requirements from (sec. 2.2.1).

TABLE 3.4 Design verification for security requirements

Requirement	**Section #**	**Lava features**
Client Security	2.3.1, 2.3.2, 2.3.4	All
Verification of the source, integrity and freshness of the agent.	2.3.2.2, 3.2	TCS and compiler certificate
Verification of the source, integrity and freshness of the delegation request	2.1.2 2.4.2.1	Secure channels Nonce
Control the actions of the agent.	2.2., 2.3.4	Security Manager, Admission Control, Language Security
Non-interference	2.2.1, 2.5.1, 3.1.4	Name spaces maintained by the Class Loaders
Reliability of the security mechanism.	2.2.1	Java's security and TCS
Flexible architecture to support individualized policies.	2.3.4, 2.4	Capability-based model
Security monitoring and reporting.	3.3	Lava Administration

5. Transmission Security

   Transmission security is considered as a service provided by lower level security protocols which is used by Lava (sec. 2.3.2). An important issue is the extent to which the design and implementation of the Lava classes will depend on the specific protocol used. All the details of establishing a secure communication will be handled by the LavaProtocol class. For example, to use Kerberos the `LavaProtocol` class will need to explicitly use the API of the security service i.e. GSS-API to provide secure channels. In order to configure the secure channels, Lava will require knowledge of the specific security mechanism being employed. This is a part of Lava's administrative functions and has not been considered in the design.

6. Lava Administration

We have considered the issues involved in the administration and management of Lava. However, this functionality has not been included in the design and thus, in this respect it is incomplete. In order to include the management modules, another design iteration will be required. However, this addition will not make a difference to the architecture because the effect of the management functions will be to change the degree of trust reposed in the providers and this has already been considered in the Lava architecture.

4. Prototype

This chapter presents the prototype implementation of Lava. This is a *proof of concept prototype*, which implements only what we consider to be the key functionality of the Lava architecture. The focus is on providing the capability-based security manager and the constructs to configure individual policies. A rudimentary support for signed applets and compiler verification is included because we consider this an essential requirement to support our assumption that the Java language is safe. Transmission security issues have not been implemented because we feel that these are well known techniques and can be easily integrated without any change to the architecture.

4.1 Introduction

4.1.1 Implementation Goals

1. No changes to the existing Java system.

   The Java language and virtual machine are being used as is. The only change to the class file format is the compiler certificate appended to the end of each class file. For compatibility our system supports signed as well as unsigned applets. However even for just downloading we believe that some kind of applet originator authentication is needed and will soon be used by the major vendors. The meaning and format of the signature for an applet may be different from that of a Lava agent. Thus Lava should support independent protocols for agent and applet authentication.

2. Reuse existing Sun classes.

   The basic mechanisms needed for the Agent Context and Runtime Monitoring subsystems were already present in Sun's `appletviewer` provided with the JDK. Since this is not meant to be a commercial product, we have tried to reuse these classes to reduce the development time. At the same time we have tried to avoid making any modifications to Sun's code as far as possible. All changes made are documented later on in this chapter.

3. Implement enough of the Lava architecture to demonstrate its functionality.

   A provider should be able to delegate an agent to an executant. The executant should associate some capability list with the provider and execute the agent under the restrictions placed on it by this capability and the intention list specified by the provider.

4. The downloading and delegation processes should be able to co-exist.

   This means that the same runtime should be used for running applets as well as agents, and Lava can support independent policies for downloaded applets and delegated agents

4.1.2 Implementation decisions

These are additional requirements needed to create a working prototype.

1. The executant is a host which participates in one or more delegations. Ideally this should be transparent to anyone using this host machine. However instead of executing agents in the background we chose, for the purpose of demonstration, to provide a display and a menu for a user on the executant to interact with the agent. The executant can send events to the agent through the options provided by the AgentViewer.

2. The design only requires the Provider class to provide an interface between the user and the executant. The specific application which will use the Lava architecture will use this class. However in order to facilitate a demonstration of Lava's features without requiring the development of the application, the prototype Provider includes a graphical user interface.

3. The access rights associated with a provider can be changed at anytime (by modifying the corresponding capability specification). This change is reflected only in subsequent delegations and not in ongoing ones.

4. On the provider side we have grouped common sets of permissions into access classes. The delegator then needs to specify the access class for the delegation and the corresponding permissions will be sent in the intention list. As in the above case, changes to the access classes will be visible only in subsequent invocations of the Provider.

5. The TCS compiler will use the java compiler (`javac`) provided with Sun's JDK. This writes the compiled files to the file system from where they will be read by the TCS server, signed and sent back to the client. Eventually the compiler should pipe the class files to the TCS File Fetcher directly to eliminate using the file system on the TCS Server.

4.1.3 Environment

The development environment is Solaris2.4 and Sun's JDK 1.02. Target environments are all Java compatible systems.

4.2 Class Descriptions

All the classes developed for the Lava prototype implementation are described below. This section first describes the delegation classes and then the TCS classes.

A typical class description has the format given below. The data members and method names are mostly generic. The aim was to describe the functionality of the class. See appendix III for the actual members.

```
packageName.ClassName

- inherits from SuperClassName

- implements InterfaceName

- data members

- methods
```

The Sun classes have not been described here.

4.2.1 Delegation

4.2.1.1 Agent

This class represents the delegation agent.

1. lava.core.Agent

   - inherits from java.applet.Applet

   - data members

   (a) a Hashtable to hold the results to be reported back to the Provider. These results will be sent back to the provider and interpreted by the delegating application.

   - methods

   (a) Add a key-value pair to the results hashtable.

   (b) Return the results accumulated in the hashtable

4.2.1.2 Provider

The Provider has 3 components:

- The GUI gets the following delegation parameters from the user/delegator.

  (a) User ID: To identify the delegator to the executant

  (b) Password: To verify the identity of the delegator

  (c) URL of the HTML page containing the agent to be delegated

  (d) Access class for the agent: To specify the access rights that the agent may be allowed to use on the delegator's behalf.

  (e) action/event: This can be one of the following:-

  - send a delegation request
  - abort a delegation
  - get a report from a delegated agent

An Intention List is constructed by looking up the capability associated with the selected access class. All this information is then sent to the ProviderComm.

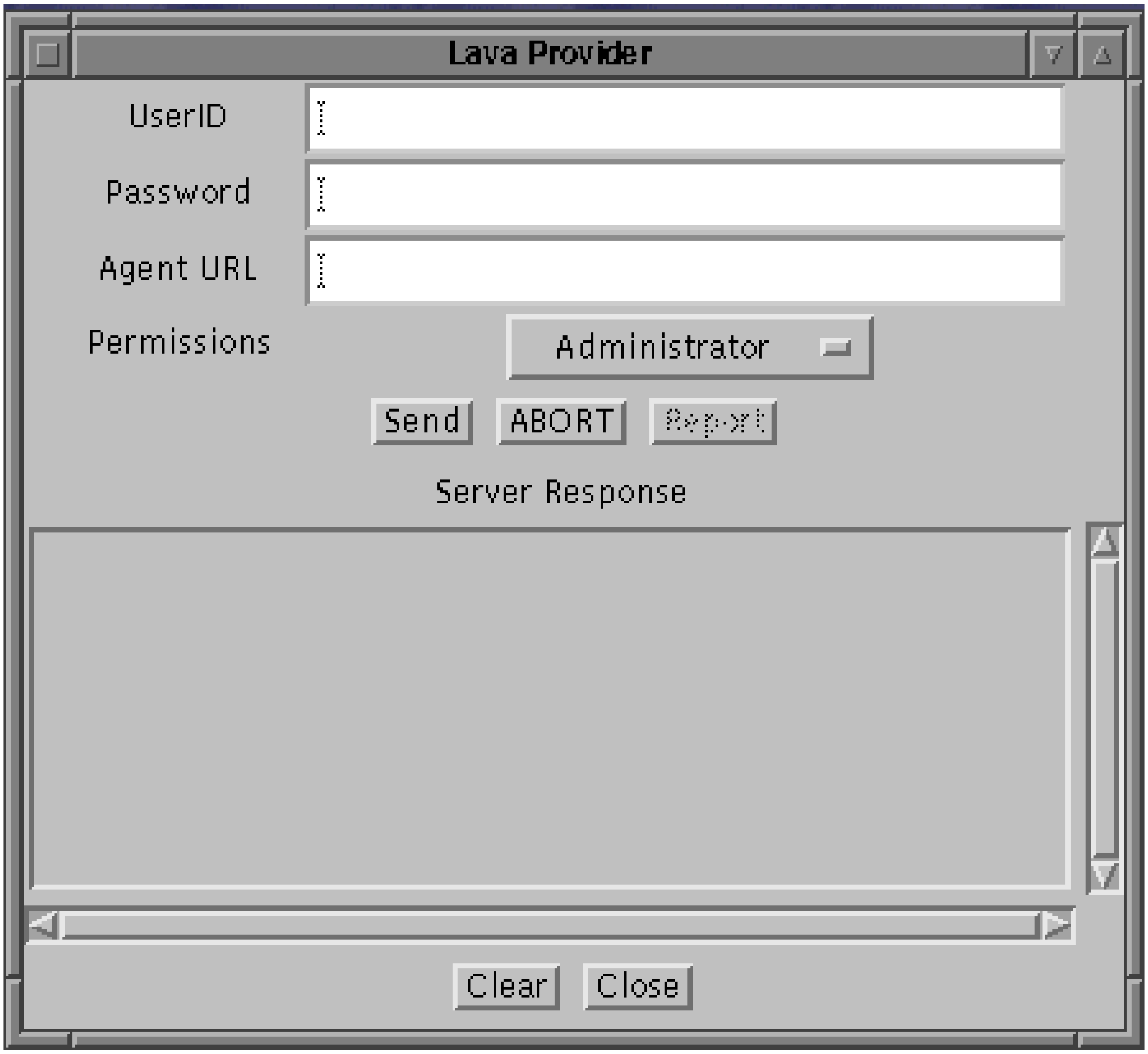


FIGURE 4.1 Provider GUI (Prototype)

- The ProviderComm encapsulates these parameters into a message for transmission to the executant. We have abstracted out the security issues in this transmission. Thus the delegation request is transmitted in plain-text and has the following format:

  `DELEGATE%%<userid>%%<password>%%<url>%%<intention list>%%`

  An intention list has the same format and specification as a Capability (sec. 2.4.4).

The revocation message is transmitted in plain-text and has the following format:

`ABORT%%<userid>%%<password>%%<url>%%`

The report request is transmitted in plain-text and has the following format:

`REPORT%%<userid>%%<password>%%<url>%%`

- The receiver thread receives the replies from the executant and sends the information to the UI. The processing of the responses is application specific, hence for the prototype we chose only to display the responses.

1. lava.agent.Provider

   - inherits from java.awt.Frame

   - data members

   (a) configuration files to create access classes and their corresponding capabilities.

   (b) GUI components for the user interface.

   (c) cache to hold the capabilities for the access classes.

   - methods

   (a) setup: Initializes the communication component represented by a ProviderComm object, the access classes and the GUI.

   (b) start: Displays the GUI and starts the receiver thread to get the results of the delegation.

   (c) handleEvent: Handles GUI events. If the “send” button is pressed then it fetches the user id, password, agent location and access class parameters and sends them to the communication module where they are formatted into a delegation request message and transmitted to the executant. Similarly for the “abort” and “report” buttons. The “clear” and “close” buttons cause the GUI to be cleared and closed respectively.

   (d) makeIntentionList: Get the capability for an access class. This is the intention list transmitted in the delegation request.

2. lava.agent.ProviderComm

   - implements java.lang.Runnable interface

- data members

(a) the executant host to which the agent is to be delegated

(a) socket connection and I/O streams to communicate with the executant.

(a) the provider using this communication.

- methods

(a) Set-up the communication channel to the executant.

(b) Construct a delegation request from the supplied parameters and send it to the executant.

(c) Construct a revocation request from the supplied parameters and send it to the executant.

(d) Construct a report request from the supplied parameters and send it to the executant.

(e) Receive the executant's responses and forward them to the provider.

3. lava.net.LavaProtocol

   - data members

   (a) Constants for the Lava delegation protocol such as the field delimiter, header identifiers for the message types and message terminator.

   - methods

   (a) construct a message from the supplied parameters.

   (b) parse a message and return the parameters.

This protocol contains the following 5 messages:

- issue a delegation
- revoke a delegation
- request a report
- send a report
- error.

4.2.1.3 Executant

Here too transmission security is assumed. Furthermore the password is also not checked since implementing a lookup to a password file is trivial. Note that such a verification mechanism is required by the Lava specification.

1. lava.agent.AgentDaemon

   - inherits from lava.net.ServerContainer

   - data members

   (a) list of trusted compilers

   - methods

   (a) initialization function: Reads a configuration file containing the trusted compiler versions and stores it in a list.

   (b) server function: Listens on a port for connections. When a connection request comes in it creates an AgentComm object to handle the incoming delegation request.

   (c) Check if a compiler version is trusted. Any compiler not in the list of trusted compilers is treated as unsafe and agents compiled by unsafe compilers are rejected by the executant.

2. sun.applet.AgentComm

   - implements ServiceProvider

   - data members

   (a) socket, input and output streams for communicating with the provider.

   (b) store is a hashtable to store the delegation parameters.

   - methods

   (a) initialization: Set up the communication channel

   (b) run: Accepts and handles messages from the executant.

   (c) communication: Read in the delegation request and parse it to obtain the userid, agent URL and intention list. A Capability object is constructed from the intention list. These are stored as name-value pairs (USER,<userid>), (PASSWD,<password>), (LOCATION,<url>) and (PERMIT,<capability>).

(d) Decide whether to accept or reject the delegation request. At this point all delegation requests are accepted. However according to the specification this function will have to verify the provider and decide whether to accept the delegation based on the criteria for admission control e.g. reject the delegation if the provider is marked as “bad” or if the executant does not have sufficient resources.

(e) Process the request and perform the appropriate action. For a delegation request, it spawns an AgentViewer to download and run the Agent. For a revocation, it stops the corresponding delegated agent and closes the viewer. Finally, for a report request, it directs the agent to send back its results to its provider.

(f) spawnAgentViewer: Encapsulate the delegation parameters in a Delegation object and parse the HTML page containing the agent(s) to be downloaded. AgentViewers are created for each of these agents.

(g) authenticate: Check the validity of the password supplied by the provider and throw a security exception if the check fails. This does nothing right now.

3. sun.applet.AgentViewer

   - inherits from sun.applet.AppletViewer for convenience.

   - data members

   (a) an AgentViewerPanel in which the agent will be contained.

   - methods

   (a) parse: Parses the HTML document to get the name and parameters for the agent.

   (b) display: Displays a window containing the AgentViewerPanel and a menu with 5 options viz. report,tag,info,properties and close. Selecting any of these causes an event to be sent to the panel which handles these events.

4. sun.applet.AgentViewerPanel

   - inherits from sun.applet.AppletViewerPanel for convenience

   - data members

   (a) provider representing this delegation instance.

   (b) the attributes of the agent are stored in a hashtable.

(c) the document URL

(d) the base URL

- methods

(a) loadApplet: Create an AgentClassLoader to download the agent and then stick it in a frame for displaying.

(b) processEvent: The panel containing the agent acts as an interface between the agent and the executant. It receives various commands as events which it enqueues and then handles in sequence. These events are to init, load, start, stop, destroy the agent, to relay a report from the Agent to the Provider and to display information about the agent.

5. sun.applet.AgentClassLoader

- inherits from sun.applet.AppletClassLoader for convenience

- data members

(a) base URL

(b) provider

- methods

(a) loadClass: Downloads bytecode from the base URL and constructs classes.

(b) verification: Separates the signature from the class and extracts its components. It generates the MAC for the class and checks it against the transmitted MAC. Then it checks if the compiler version is trusted. For the prototype the Compiler Certificate attached to a class file is not encrypted and does not have the TCSid field.

6. sun.applet.AppletSecurity

The sun.applet.AppletSecurity manager has been modified to perform the security checks for applets as well as agents. This is because we use the same mechanism for both applets and agents.

- inherits java.lang.SecurityManager

- methods

(a) initialization

i) creates a CapabilityIndex which is initialized with <provider, capability> pairs. First <provider, capability location> pairs are read in from a configuration file. The format of the "capability location" is <filename>#<offset> where the capability specification for the provider is located. Then Capability objects are created from these capability specifications and stored in the CapabilityIndex.

ii) sets the global networking policy to the appletviewer.security.mode property. This may be NETWORK_UNRESTRICTED, NETWORK_NONE or NETWORK_HOST.

(b) security functions: Implement a global policy for access control which is common for all downloaded content.

i) Creating top-level windows.

ii) Access a package.

iii) Define classes in a package.

iv) Set up a networking-related object factory.

For all other resource accesses it gets the Capability for the agent and checks if the particular access is permitted. Look at the security policy implementation (sec. 4.3) for more details.

7. lava.agent.Capability

- data members

(a) dirList: hashtable containing files and READ/WRITE permissions.

(b) portList: ports on which the applet is allowed to listen.

(c) toHostList: remote host:ports to which the applet can connect to.

(d) fromHostList: remote hosts from which the applet can accept connections.

(e) propertyList: all the properties which the applet is allowed to read.

(f) canExec: flag to indicate if the agent is allowed to exec local processes.

(g) provider: The provider with which this capability is associated.

(h) mistrusted: is the provider mistrusted or bad? This is not being checked for because its implementation is trivial and the decision of labelling a provider as "bad" is application dependent. Eventually this will be checked before downloading the agent.

(i) urlList: List of applet locations which are "trusted". *Not Used.*

- methods

(a) creating the capability: The capability specification is read from a file or a stream and the corresponding data members are initialized.

(b) set defaults: This is used to enforce the default applet/agent policy.

(c) merge: Merges two capability objects. For resource access permissions the merge selects the most restrictive permission set among the capabilities being merged, for every resource. In general it is similar to an intersection of the permission sets of both the capabilities. e.g. for a file the resulting capability has a READ access to it only if both merging capabilities have READ access to it. However if any of the capabilities are marked "bad" then the resulting capability is also marked "bad".

(d) Checks to filter access resources on the client system. Refer to API (Appendix III) for a complete list of checks. The following methods have not been implemented completely: `checkExec(String)`, `checkLink(String)`, `checkRead(FileDescriptor)`, `checkWrite(FileDescriptor)` and `checkConnect(String,String,int)`.

8. lava.agent.CapabilityIndex

   This has an index of all the recognized providers and a cache of the all their Capabilities on the executant.

9. lava.agent.Delegation.

   This represents an instance of the delegation process.

   - data members

   (a) the id[1] of the provider

1. pronounced "i-dee"; abbreviation of identifier.

(b) an output stream to send back messages to the provider

(c) permit: a Capability object which represents the joint permissions granted by the provider and the executant to the agent.

- methods

(a) Get the Capability associated with that provider from the CapabilityList and merge it with the Capability of the provider specified intention list. This gives the effective permissions for the delegated agent.

4.2.2 Trusted Compiling Service

The TCS implementation is very close to the original design and was developed without any changes to the existing java or sun packages.

Server side classes

1. lava.tcs.TCSServer

   - inherits from ServerContainer

2. lava.tcs.TCSComm

   - implements ServiceProvider

   - data members

   (a) client socket

   (b) I/O streams to communicate with the TCS client.

   - methods

   (a) Invoke a TCSCompiler to get and compile the Java source files. If the compilation is successful then construct the compiler certificate and invoke the TCSFileFetcher to send back the compiled classes.

   (b) Send a status message back to the client.

3. lava.tcs.TCSCompiler

   - data members

(a) target dir: directory in which the compilation will be performed

(b) targetDirList: A hashtable of target directories and their availability. A target directory is locked before use and freed after use.

- methods

(a) Get the files to be compiled from the TCS client. Select a free directory to perform the compilation or create a new one if none are available. Lock this directory. Place the source files into it and compile it. The prototype simply invokes the compiler provided with the Sun JDK. If the compilation fails then send the compilation errors back to the client.

(b) make the certificate: Append the current time and the version number of the compiler used to the bytecode and then run the MD5 hashing algorithm to construct the compiler certificate.

```
CompilerCert = [Version, TimeStamp, MD5(applet)]
```

The prototype uses a simplified version of the compiler certificate given in the design specification (sec. 2.4.2.2) to authenticate the code.

4. lava.tcs.TCSFileFetcher

   - data members

   (a) input stream to receive data from the TCS client.

   (b) error stream to send back status/error messages to the client.

   (c) a TCSOutputStream to send the class files to the client.

   - methods

   (a) Get the class names from the client, fetch the corresponding files from the temporary directory used for compilation, transmit them back to the client, delete the files and finally free the target dir for reuse.

5. lava.tcs.TCSOutputStream

   - inherits from java.io.DataOutputStream

   - methods

(a) write: Generate a compiler certificate for the byte segment, append this to the end of the segment and transmit it.

Client side classes

6. lava.tcs.TCSClient

   - data members

   (a) host where the server is running

   (b) port to connect to on the server host

   (c) I/O streams to communicate with the TCS server.

   (d) a TCSClassLoader to download the compiled bytecode from the server, construct classes from it and save the class files to the file system.

   (e) a list of source (.java) files to be compiled.

   (f) the name of the main class to be downloaded. There may be more than one main class generated by a compilation job. These will be handled one at a time.

   - methods

   (a) get the java source files: The user is prompted for the.java files which are to be compiled together.

   (b) send the files to the server: Transmit the files separated by EOF delimiters, to the TCS server.

   (c) download: If the compilation is successful invoke the loader to download the classes.

7. lava.tcs.TCSClassLoader

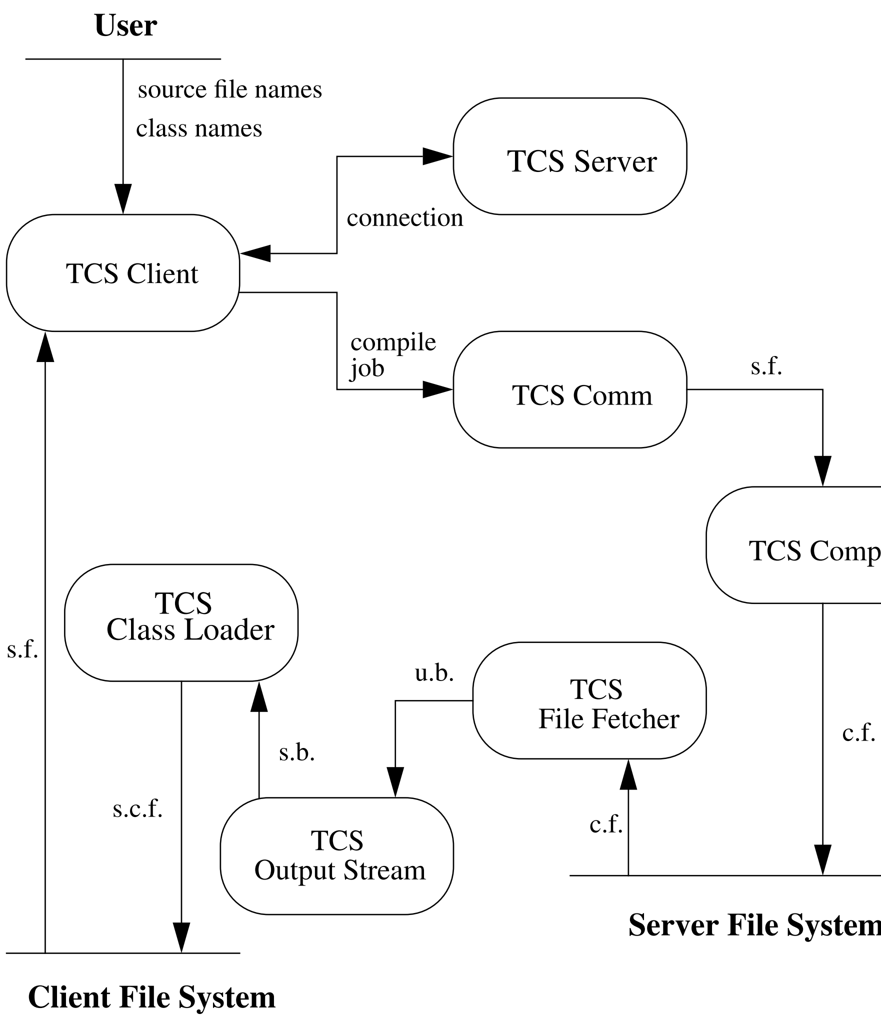
   - inherits from lava.system.NetworkClassLoader

   - data members

   (a) I/O channel to the client.

   - methods

(a) Load a class file from across the network and save it to the file system along with its digital signature. Construct a class object.[1]



s.f. = source (.java) files which need to be compiled
compile job = compilation request containing the names and contents of the.java files.
c.f. = class files generated from the submitted sources.
u.b. = unsigned bytecode; the content of the class files
s.b = signed bytecode
s.c.f. = signed class files

FIGURE 4.2 TCS mechanism in the Lava prototype

1. The class construction involves resolving the references to other classes made in the current class and recursively downloading them if needed

4.2.3 Other classes

1. lava.net.ServerContainer

   - data members

   (a) port on which the server accepts connections

   (b) service class: The service provider which will handle accepted connections.

   - methods

   (a) This class provides the functionality of the driver thread of a typical server. It waits for connections at a port and when a connection request comes in it creates an object of the requisite service provider class.

2. lava.net.ServiceProvider

   This interface provides a template for service providers which can be plugged into a ServerContainer.

   - methods

   (a) init: Initialize the service provider.

   (b) results: The results generated by the service provider

3. lava.util.Set

   An unordered collection of elements without duplicates.

   - methods

   (a) union

   (b) intersection

4. lava.util.MultiHashtable

   A hashtable with multiple values associated with each key.

5. lava.util.TwoKeyHashtable

   A hashtable which uses two keys to identify a value.

6. lava.net.NetworkClassLoader

   A class loader that downloads classes across the network using sockets.

7. lava.util.md

   Implements the MD4 and MD5 algorithms. This was taken from Harry Mantakos' JOTP[1] applet at http://www.cs.umd.edu/users/harry/jotp/src.html, and used as is for the prototype.

4.3 Security Policy

This section describes the security policy implemented by the Lava architecture. It has the structure proposed in (sec. 2.5.1).

4.3.1 Global Policy

1. The network mode can be NONE,HOST,UNRESTRICTED. The selection is applicable to all applets & agents.

- NONE: cannot connect to any host
- HOST: can connect to "trusted" hosts which are specified in a configuration file. These permissions are specific for each applet provider and may differ on different executants.
- UNRESTRICTED: can connect to any other host on the internet. This is again a global choice like NONE and hence individual permissions don't need to be checked.

2. Agents are not allowed to create class loaders.
3. Top level windows cannot be created by agents. This means that all windows created by an agent will have a warning sign.
4. An agent cannot define classes in any package which is marked as "restricted" i.e. a system property `package.restrict.definition.<packagename>` exists and has the value `true`.
5. Agents are not allowed to exit the virtual machine.
6. Agents are not allowed to manipulate threads or thread groups outside their applet thread groups.

1. Java One Time Password calculator.

4.3.2 Provider-based policy

Default policy

1. cannot access the file system.
2. can connect back to its server but not to any other machines.
3. cannot listen on ports.
4. cannot exec system calls or link dynamic libraries.
5. can access a property only if its twin `<property>.applet` exists and is set to `true`.
6. is not “mistrusted”

These are the minimum permissions given to any downloaded agent. If the provider is not recognized by the executant then a default capability is assigned which enforces this policy.

Capability specified policy

1. Read/Write permissions for files and directories have to be specified using the FILE tag in the configuration file.
2. Permissions to listen on a port, accept connections from remote hosts and connect to remote hosts are specified with the LISTEN, ACCEPT and CONNECT tags respectively. An agent can listen for connections only on specified ports. An agent can accept connections originating from specified host-port pairs. An agent can open a connection to any port on the machine from which it is downloaded and any port on specified remote hosts. The CONNECT tag also has a field for the port number to which a connection is allowed but for now this is ignored by the security policy.
3. Can only `exec` specified system commands. For the prototype this is an *all or nothing* choice.
4. Can only link to specified dynamic libraries. For the prototype this is *all or nothing* and is considered the same as `exec`’ing a system call.

5. Can access applet properties (see default policy above) and the properties specified with the PROPERTY tag.

6. The BAD tag marks a provider as "bad" (sec. 2.4.2.1). This has *not been implemented* in the prototype.

4.3.3 Intention List (IL)

The intention list is a construct for the Provider to indicate what he/she expects the delegated agent to do.

- The specification of the IL is the same as a Capability specification, which makes it easier to combine access rights specified by the executant and the provider.

- The IL is specified independent of the permissions of the provider on a particular client i.e. the IL does not have to satisfy any subset relation.

- For convenience, typical sets of access rights are categorized into access classes such as administrator, file manager, network manager, privileged user, ordinary user etc. These can be set and configured using the configuration files (Appendix Ia). This grouping is transparent to the executant which may also use its own access classes independently.

4.4 Changes to Sun's classes

All these classes were from Sun's JDK 1.02.

1. sun.applet.AppletClassLoader

   (a) Added a `provider` data member which the AgentClassLoader inherits. This is `null` for applets. This field represents the entity authorizing the applet's execution. Thus it provides a placeholder for the support of *trusted applets*.

   (b) Changed the access modifier of `loadClass(URL url)` method from `private` to `protected` so that its sub-class, AgentClassLoader can over-ride it.

   (c) Added checks for signed classes. If a class is signed it ignores the class part for now. However this provides support for signed classes.

   (d) added an access method for the provider viz. `getProvider()`

(e) added a `checkVersion()` method to check if the applet (if signed) was compiled by a trustworthy compiler.

2. sun.applet.AppletPanel

   Added a `processEvent()` method to perform the action for the received event. The original sun.applet.AppletPanel did this processing in the `run()` method itself. In order to use inheritance properly it was necessary to improve the modularity.

3. sun.applet.AppletViewer

   Before trying to create and install a security manager it checks whether a security manager is already installed.

4. sun.applet.AppletSecurity

   This has been completely changed from Sun's implementation, to implement the capability-based security manager. This system supports both downloading of applets and delegation of agents and hence it was necessary to have a common manager to monitor both. The AppletSecurity in Lava differs conceptually from the AppletSecurity in `appletviewer` in the following ways:

   The `appletviewer`'s AppletSecurity class is monolithic. It contains the specification of the access policy and is also responsible for enforcing them. In Lava this class provides a framework for a flexible monitoring system and the policies are implemented by capabilities. It enforces the global policies but provider policies are enforced by a capability which is called by the manager. The name is misleading and probably should be changed to *LavaSecurity.*

   This class corresponds to the AgentSecurity class of the design specification (sec. 3.1.4.2).

5. Conclusion

5.1 SNMP over Lava

This section describes an on-going project to implement an SNMP based application over the Lava framework.

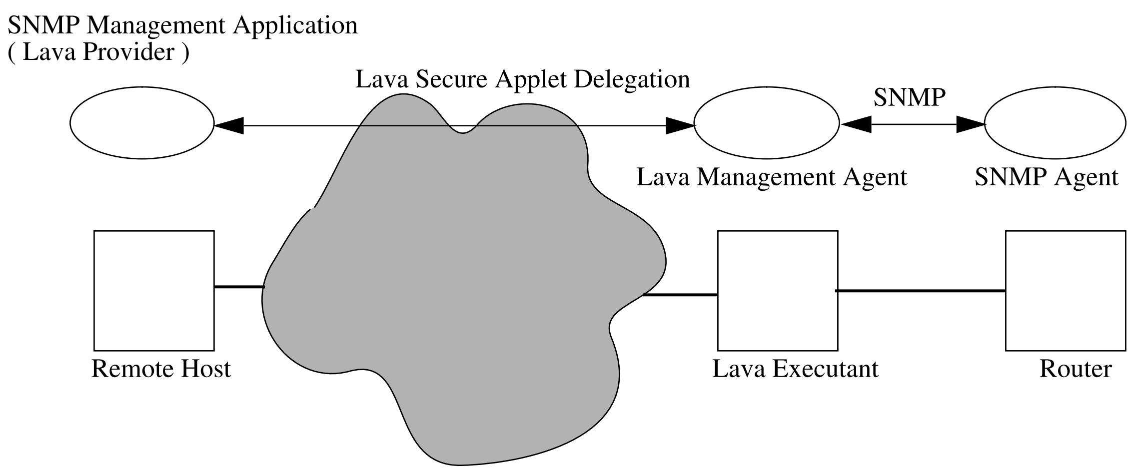


FIGURE 5.1 SNMP over Lava

As shown in the figure above, the management application delegates the Lava agent to the Lava executant which then communicates with an SNMP agent on a router. The Lava agent will monitor the activities of the router(s) and the results generated by it are used by the management application.

At the time of writing this document we had developed and tested a Lava agent which would periodically walk the router's routing table and try and detect a TCP SYN flood attack [7].

The detection mechanism essentially consists of periodically checking the number of half-open TCP connections at a router. The router is considered to be under attack if this number is above some statistical threshold. Now, if a single remote manager was responsible for managing a large number of routers then it would need to get the status of all TCP connections on each of the routers just to get one set of readings. This would need to be done many times to collect enough data for a decision. Here the management application or the network bandwidth could be a bottleneck. In the Lava approach, mobile agents are delegated to each of the routers. Each of these can independently keep checking the MIB[1] on its own router and collect statistics in real time. Thus the detection is faster and more efficient.

5.2 Summary

The objective of this project was to propose an architecture which would support the delegation of mobile agents in a secure fashion within the context of Sun's Java technology. Java is rapidly proving to be a key technology for internet applications and this is the primary reason for selecting it as the basis for Lava. The specific reasons for the technical suitability of Java have been discussed in (sec. 1.4). The attempt has been to identify the core functionality needed by all distributed applications which use delegation and provide these in the Lava framework. Lava provides an open and secure architecture which could benefit certain types of distributed applications discussed in (sec. 1.1). Thus a number of issues like algorithms for transmission security, result verification, key distribution and feasibility studies for specific applications have not been considered in much detail.

Chapter 2 exclusively deals with the security aspects of the Lava architecture. The salient features of the security architecture are flexibility, granularity of monitoring, layering and trust relations. An important factor affecting the security architecture is that Lava is based on the Java system and correspondingly Lava security uses the mechanisms provided by the Java system for ensuring client safety. Lava uses a capability based, bipartite monitoring system which filters all attempts made by downloaded content for local resources with a policy which may be different for each delegation instance. Thus the same agent can be delegated by different providers having different permissions or the

1. Management Information Base

same provider may delegate many agents with differing permissions. Finally the security of Lava is usable for downloading as well as delegation.

Lava has an object oriented design which is described in chapter 3. The design has been verified against the various requirements identified in the previous two chapters. In order to demonstrate the feasibility of the Lava architecture we have implemented a prototype which is described in chapter 4. As mentioned earlier, this is a proof of concept prototype, whose purpose is to provide a working framework over which an example distributed application can be implemented. The current prototype allows an agent to be delegated to an executant and executed there under a policy which is jointly determined by the trust reposed by the executant in the provider and by the provider in the agent.

In conclusion, the architecture proposed in this document does provide a suitable framework for building distributed applications.

It supports migration of mobile agents and allows the provider and executant to independently specify the permissions to be granted to an agent. The architecture then transparently merges the two specifications to decide on the effective permissions granted to a delegated agent. This decoupling makes it easier to update agents at the web server and policies at the provider and executant.

Lava uses a capability-based security manager to support a flexible and customizable security policy. The use of individualized capabilities to filter resource accesses allow delegated agents to do much more *real* work than Java applets could. However with added power comes added responsibility on the author of the agent, to ensure that the agent does not unnecessarily tie up resources or expose confidential information to untrusted entities.

Distributed systems' security has a large number of issues which need to be resolved. The most common ones are those associated with secure transmission. Lava proposes using an existing solution like Kerberos to resolve these issues. More difficult to solve are the problems of denial-of-service attacks and result verification since these require knowledge of the specific application, in order to detect an attack. Result verification in Lava would require a guarantee that the agent does what it is expected to, the execution was not tampered with and the transmission of the results is secure. The trust relation and secure channel between the executant and provider can provide the later two assurances. However in order to guarantee the first requirement we require a trust relation between the agent originator and the provider, which is outside the scope of Lava. In (sec. 5.3) we have proposed a way to alleviate the problem of some denial of service attacks using access control policies based on usage count and time.

This has not yet been proved to be a feasible architecture for distributed applications primarily because no analysis or experimentation has been done to conclusively show that a Lava based distributed application will indeed perform better than the corresponding non-Lava application. In (sec. 1.1) we identify the kinds of applications which would benefit from an agent delegation mechanism. However the prototype and design analysis have proved that secure delegation of agents is possible and a variable control mechanism can be effectively used to monitor agents and maintain the safety of all participants in the delegation process.

5.3 Future Work

The Lava system is in a preliminary stage and does not address a number of issues which need attention in order to eventually make it an industrial strength system.

1. Scalability

   In order for the design to scale well the performance of the delegation system must be measured and benchmarked. The current design allows an unlimited number of agents on an executant, which in turn means that the consumption of local resources can grow unbounded. This will eventually cause the system to crash. There is a method to reject delegations but the criteria for this is not specified in the Lava design.

2. Completeness

   The current design does not precisely define the interfacing between the transmission security services and Lava. The administration and management interfaces and modules are also unspecified. The establishment procedure for trust relations is assumed but it still needs to be described for a complete specification. The security of

Lava depends upon the safety of the language compiler. It assumes that the TCS cannot be violated. In order to justify such an assumption we need to define the procedure for designating a compiling service as *trusted* and the measures for maintaining it's security.

Some changes to the protocols in Lava may be required. For example, the compiler certificate may need an extra field to specify which message digest algorithm is being used.

3. Time / Availability based access control

   An application may want to allow an agent access to a particular file only for a limited period of time or may have an upper limit on the number of ports which an agent is allowed to use. Lava does not provide any mechanism to restrict resource access on the basis of usage or availability. This is an important feature and it can be added to the Lava architecture by changing the format of the capability specification to include such information and modifying the design of the Capability class to include timing and frequency information in the state and to use this to allow or disallow a resource access attempt. This is targeted towards denial of service type attacks.

4. Admission Control
   (sec. 2.3.3) proposes some possible criteria for admission control in Lava. However the design does not define any protocol for this. (sec. 3.1.1) presents a design decision which provides flexibility to agent designers but at the same time increases the programming complexity. A generic solution would be to assign *priorities* to resource requirements.

   For example, an agent could declare that it cannot function without read access to directory ‘xxx’. In this case the executant can check whether the directory will be accessible to the agent before starting its execution. This moves the complexity from the agent code to the specification of the access rights.

   Based on this and the issue of time/availability based access control (see 3), I believe that the capability specification needs a *richer semantics* which will include priorities as well as the changes proposed in 3.

5. Agent initiated asynchronous communication

In the Lava architecture all communication between the provider and executant is handled by the agent context. Thus in order for the agent to send a message to the provider it must ask the executant to forward it. The current design does not support this feature. Thus the provider must periodically poll the executant for results generated by its agent.

The agent context needs to implement an interface which the agent can use to send back reports voluntarily.

The prototype does implement an asynchronous communication whereby the executant can get a report from the agent send it back to the provider. However this is not client initiated.

6. State Information

   Currently a delegated agent is just a piece of executable code. It does not carry over any state and can not interrupt and re-start itself. Such an autonomous agent is defined by the Aglets Workbench specification. In order to support stateful, intelligent agents, Lava would need to use a mechanism like object serialization to transfer state information and a runtime mechanism to continue an agent's execution using the transferred state. Thus Lava could then support agents having varying degrees of intelligence and independence. At this stage Lava can co-exist with the aglet system i.e. an aglet may use Lava to share its work among different hosts, or an aglet type system can be built on top of Lava.

7. Speed

   At this stage Lava is not proposed as an end-to-end solution. A Lava agent will usually use the services of an application specific agent residing on the target (e.g. SNMP agent) and then process the results. The main reason for this is that bytecode interpretation is "slow". With system (e.g. JavaOS [42]) and machine (e.g. Java chip [49]) level support for Java and JIT compilers the speed problem could be solved. Then it would be feasible to use Lava as an end-to-end framework. Another reason for the type of arrangement in (Fig. 5.1) is that Lava must interoperate with legacy systems i.e. since SNMP agents are already available on routers it makes sense to use them.

8. Prototype verification

A reasonably realistic distributed application needs to be implemented over the Lava prototype. This is the next phase of the project. The traffic generated by the application will be compared to the traffic which would be generated by a non-delegated version of the application, to verify the benefits of the agent delegation approach.

9. Efficiency

   We need to profile the performance of the system and detect any inefficiencies in the design or implementation.

   The interpreted nature of Java bytecode causes a performance penalty. There are a number of mechanisms to improve the execution speed of Java code and these could be used for Lava.

10. Effect of new Java releases

    Sun Microsystems released the JDK 1.1 in April 1997. This includes a numerous new features and APIs [44]. We have not tried to execute the prototype using this new version. However the documentation and release claim that the JDK 1.1 is fully compatible with JDK 1.02, which means the Lava architecture can be used with the new version too.

    However, Lava can make use of a number of the new features. The Java Security API [43] allows developers to easily and securely include security functionality in their applets and applications. This includes cryptography with digital signatures, encryption, and authentication. This could be used for the compiler authentication in Lava.

    At the recent JavaOne conference in San Francisco, April 1997, Sun proposed some extensions to the Java sandbox protection model. The extensions essentially provide a separation of policy from mechanism and specify the policies in a separate persistent format as ascii files [19]. This is almost identical to the capability specification used in Lava. Thus our architecture dovetails with the future security model of Java.

11. Lava as an end-to-end system

Lava takes a least common denominator approach to the requirements of distributed systems. Consider an application which needs to access a database on the executant. In this case, the resources the Lava agents will access will be database objects. Here, the access control policies of Lava are not very applicable.

In order to use Lava as an end to end system, it needs to be extensible so that application dependent interfaces can be built on the executant to provide application specific access control using Lava's mechanisms. The advantage of this is that all applications will have a secure platform with a flexible, individualized access control system. (Fig. 5.2) shows the envisioned architecture. The Lava Extension sublayer provides an interface to the Lava components on the executant. The NMS extension module provides NMS specific monitoring. The provider side of the application remains the same.

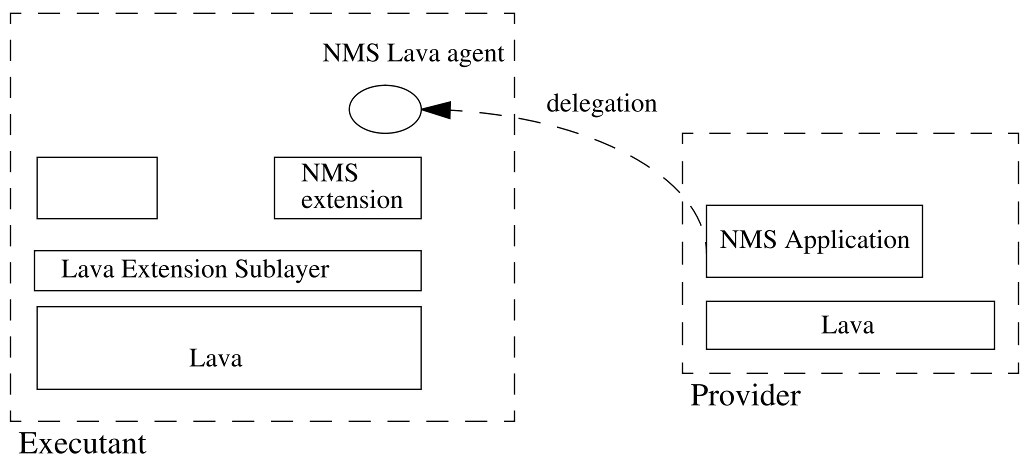


FIGURE 5.2 Lava as an extensible, end-to-end system

Bibliography

[1] Advent, *Java SNMP*. March 1996.

http://www.adventnet.com/snmp_api.html.

[2] Amoroso, Edward G, *Fundamentals of Computer Security Technology.* Prentice Hall, 1994.

[3] Anand R., Islam N. and Rao J., *A Capability-based Security Model for using Internet Content.* To be assigned an IBM Technical Document number, 1996.

[4] Bank Joseph A., *Java Security.* MIT, December 1995.

http://www-swiss.ai.mit.edu/~jbank/javapaper/javapaper.html

[5] Cai H. and Yuill J., *The Functionality and Design of an SNMP MIB Browser*, Network Security Project Report, North Carolina State University 1996.

[6] Comer Douglas, *Internetworking with TCP/IP Vol.1 - Principles, Protocols and Architecture*. 3rd edition, Prentice Hall 1995.

[7] Computer Emergency Response Team (CERT), *TCP SYN Flooding and IP Spoofing Attacks*. CA-96:21. Carnegie Mellon University, September 1996.

[8] Dean Drew, Felten Edward and Wallach Dan, *Java Security: From HotJava to Netscape and Beyond*. Proceedings of the IEEE Symposium on Security and Privacy. May 1996.

[9] Dean Drew, *The Security of Static Typing with Dynamic Linking*. Fourth ACM Conference on Computer and Communications Security. Princeton, April 1997.

[10] Deri Luca, *Surfin' Network Resources across the Web*. IBM Research Laboratory

[11] Drossopoulou S. and Eisenbach S. *Java is Type Safe - Probably.* 11th European Conference on Object Oriented Programming, June 1997.

http://www-ala.doc.ic.ac.uk/~scd/JavaW.html

[12] Duvos Mike, *Re: The Joy of Java.* Post to CypherPunks news group, April 26, 1996.

http://infinity.nus.sg/cypherpunks/dir.archive-96.04.25-96.05.01/0088.html

[13] Erdos Marlena, Hartman Bret, Mueller Marianne, *Security Reference Model for the Java Developer's Kit 1.0.2.* Sun Microsystems. November 1996.

[14] Farmer W., Guttman J. and Swarup V., *Security for Mobile Agents: Authentication and State Appraisal*. 4th European Symposium on Research in Computer Security, Rome, September 1996.

[15] Fisher John, *Java and JavaScript Vulnerabilities*. CIAC Notes 96-01. CIAC - DOE, March 1996.

[16] Flanagan David, *Java in a Nutshell*. O'Reilly and Associates 1996.

[17] Goldszmidt German S., *Distributed Management by Delegation*. Ph.D. Thesis. Columbia University 1996.

[18] Gong Li, *New Security Directions for Java*. Proceedings of IEEE COMPCON, Feb 1997

[19] Gong Li, *Java Security: Present and the Near Future*. Presentation at JavaOne. Sun Microsystems, April 1997.

Slides at http://java.sun.com/javaone/sessions/slides/TT03/index.html

[20] Gosling, Joy and Steele, *The Java^{TM} Language Specification*. Addison-Wesley 1997

[21] IBM, *Mobile Agent Facility Specification*. OMG TC Document cf/96-08-01

[22] IBM, *Aglet Workbench*.

http://www.trl.ibm.co.jp/aglets

[23] IBM, *Flexible Control of Downloaded Executable Content.* Personal Communication, March 1997.

[24] Kramer Douglas, *The Java^{TM} Platform: A White Paper.* Sun Microsystems, May 1996.

http://www.javasoft.com:80/docs/whitePaper.Platform/CreditsPage.doc.html

[25] Liao and Qian, *Active Capability: An Application Specific Security and Protection Model.* Technical Report, Univ. of Illinois at Urbana-Champaign 1996.

[26] Lindholm Tim and Yellin Frank, *The Java^{TM} Virtual Machine Specification*. Addison-Wesley 1997

[27] Linehan Mark, *Comparison of Network Level Security Protocols*. IBM T.J. Watson Research Center. June 1994

[28] Madany Peter, *JavaOS^{tm}: A Standalone Java Environment - White Paper*. Sun Microsystems.

[29] Meyer Jon, *Jasmin Home Page*. New York University, March 1997.
http://mrl.nyu.edu/meyer/jvm/jasmin.html

[30] Minden (Univ. of Kansas), Sincoskie (Bell Communications Research), Smith (Univ. of Pennsylvania), Tennenhouse (MIT), Wetherall (MIT), *A Survey of Active Network Research*.

[31] Netscape, *Secure Sockets Layer.*
http://home.netscape.com/info/security-doc.html

[32] Rose Marshall T., *The Simple Book: An Introduction to Management of TCP/IP based Internets*. Prentice Hall 1991.

[33] Rescorla E. and Schiffman A., *The Secure HyperText Transfer Protocol*. Internet Draft, March 1997 (expires September 1997)

[34] Silbershatz Abraham and Galvin Peter, *Operating System Concepts.* 4th edition, Addison-Wesley 1994.

[35] Sloman Morris and Yialelis Nicholas, *A Security Framework Supporting Domain Based Access Control in Distributed Systems*. IEEE 1996

[36] Snapp, Steven R. et al, *DIDS (Distributed Intrusion Detection System) - Motivation, Architecture and an Early Prototype.* 14th National Computer Security Conference, October 1991.

[37] Sommers Bret, *Remote Objects for Java*, Java World, June 1996

[38] Sommers Bret, *Agents: Not Just for Bond Anymore*, Java World, April 1997.

[39] Stallings William, *Network and Internetwork Security - Principals and Practice.* Prentice Hall 1995.

[40] Sun Microsystems, *Java FAQ*.
http://www.javasoft.com:80/products/jdk/faq.html

[41] Sun Microsystems, *Security FAQ*.
http://java.sun.com/sfaq

[42] Sun Microsystems, *JavaOS 1.0 Ships to Licensees*. Press Release, March 5, 1997.
http://www.javasoft.com/pr/1997/march/pr970305-01.html.

[43] Sun Microsystems, *Java Security API Overview.* February 1997.
http://www.javasoft.com:80/products/jdk/1.1/docs/guide/security/JavaSecurityOverview.html

[44] Sun Microsystems, *JDK 1.1.1 New Feature Summary.*
http://www.javasoft.com:80/products/jdk/1.1/docs/relnotes/features.html

[45] Systems Software Research Group, *JGSS (Java Generic Security Services)*. University of Illinois at Urbana-Champagne.
http://choices.cs.uiuc.edu/Security/JGSS/jgss.html.

[46] Taft S., *Programming the Internet in Ada 95*. Submitted to Ada Europe '96. Intermetrics Inc., Cambridge, MA.
http://www.appletmagic.com/

[47] Tock Theron, *An Extensible Framework for Authentication and Delegation.* Master's Thesis, University of Illinois at Urbana Champagne 1994.

[48] Venners Bill, *The Architecture of Aglets*. Java World, April 1997.

[49] Wayner Peter, *Sun Gambles on Java Chips*. Byte, November 1996.
http://www.byte.com/art/9611/sec6/art2.htm

[50] Yellin Frank, *Low Level Security in Java.* WWW4 Conference, December, 1995.

http://www.javasoft.com/sfaq/verifier.html

[51] Yellin Frank, *The Java Application Programming Interface: Volumes 1 and 2*. Addison-Wesley, 1997.

[52] Zenel Bruce, *A Proxy Based Filtering Mechanism for the Mobile Environment*. Technical Report CUCS-0xx-95 Thesis Proposal. Columbia University, 1995.