Towards the Design of an Internet Operating System

POSITION PAPER

Chrislain Razafimahefa
Ciarán Bryce

Abstract

This position paper describes a design approach for a kernel that supports Internet applications. The kernel must run on any device, from a PDA to a workstation, and be able to participate in a range of applications, from ubiquitous computing to electronic commerce. The core features of this kernel include security, disconnected operation, coordination with other kernels and compactness. We argue for a programming language approach: a single language is used to program applications that is dotted with concepts for security, disconnected operation, etc., and the kernel is structured as a virtual machine for this language.

1 Introduction

The Internet is evolving to include wireless and mobile devices. Users will want access to Internet services from anywhere and at anytime, and the move towards these devices will sustain the exponential growth of the Internet. Estimates suggest that there will be 1 billion mobile telephones in circulation by the year 2003, and that in Europe alone, 1 in 3 Internet accesses will be made using mobile phones. Third generation wireless protocols like UMTS [14] will permit the exchange of multi-media data between phones. Thus, Internet applications and services will have to adapt to the wireless evolution.

Another trend that is revolutionizing networks are personal device assistants (PDAs) with short-range wireless communication facilities. These allow for ad hoc [16] and spontaneous [5] networks. A spontaneous network is formed by groups of people in the same geographical location; the network changes as soon as someone arrives or leaves. These networks are based on technologies like Bluetooth [13], and are characterized by large bandwidth fluctuations and highly dynamic configurations. Estimates suggest that there will be 670 million Bluetooth enabled devices by 2003, so the potential for such networks is large, though still untapped.

Another trend that will affect the structure of the Internet is the growth of ubiquitous computing [32]. 98% of all processors on the planet are not in traditional desktop computers, but are embedded into everyday devices such as vehicles as well as home and electrical appliances. In the near future, these devices will be interconnected via the Internet. Being highly interconnected, embedded devices can collectively respond to events in their environment without human intervention. The devices will be programmable to yield better control of their environment.

*This work is supported by the Swiss National Science Foundation, under the grant FNRS 20-53399.98 "Secure Mobile Objects".
The changes that we have described at the network infrastructure level will also trigger changes in the programming models used for distributed applications. Today, applications running over the Internet are typically based on the client-server paradigm: a server continuously runs on one node and clients on other nodes request services offered by the server. This corresponds to a pull information flow model and is quite static. The configuration between the server and the client is known beforehand and does not change during operation. The most popular example of this in an Internet application infrastructure is the World-Wide Web [6].

The static nature of client-server computing will be replaced by more dynamic configurations, that is more and more Internet applications will run over many nodes and involve the coordination of tasks and data from several nodes and administrative domains. One reason is that devices will create a push information flow model, with information flowing from nodes to clients even without them explicitly asking for it. For instance, while traveling in a car that enters into a new region, information regarding the region such as the names of closest hotels, the next gas station or the map of the next city could be automatically pushed onto an embedded device on the car. Similarly, a user who enters into a market place carrying a PDA will receive offers from sellers in the market place that he must sieve through.

Another important issue in the programming model is the possibility of disconnection. Tolerating disconnection can necessitate the dynamic displacement of programs (i.e., mobile code) and data to different sites. The increased need for timeliness guarantees for services will see an increase in server replication [10], and the continuing growth of electronic commerce will see the use of work-flow like applications on an Internet scale [20].

The emergence of mobility, both of devices and programs, opens new possibilities for the Internet, but also raises many concerns. For instance, as recent viruses such as Melissa and I Love You illustrate, mobile code can contain viruses that attack their host platform. Viruses still seem to find weaknesses in state-of-the-art security mechanisms. Further, wireless communications are subject to relatively high transmission error rates, bandwidth fluctuations and disconnections. Allowing users to work off-line, following disconnection, is an important challenge. It means that code and data required to continue working must be available after disconnection, and that data consistency must be catered for after reconnection.

We have started work on the design and implementation of an Internet Operating System kernel called Lana (for “Language for Advanced Network Architectures”). The goal of this kernel is to run applications, ranging from supervisor processes for embedded devices, to electronic commerce applications. Our goal in this paper is to overview the design choices of the kernel. The key feature of our design is that it follows a programming language-based approach. This means that we add abstractions to the underlying programming language – a watered down version of Java in our case [3] – that cater for security, coordination of networked kernels and disconnected operation. The kernel then becomes a virtual machine for executing programs of this language.

In Section 2 of this paper, we overview the design of the kernel by qualifying how it offers support for security, coordination and disconnected operation. Our work on this project has already started, and will be used in several on-going projects; we describe these in Section 3.
2 Application Programming Issues

We believe that there are three key issues for an InternetOS to support: security, coordination and disconnected operation. Each of these issues are looked at individually in the following paragraphs. We then look at the issue of compactness - since the Lana kernel must run on small devices, and we terminate the section with a summary of the language.

2.1 The Programming Model

*Lana* is an object based language providing the notions of object, class and inheritance. The system allows for multiple *Programs* to run concurrently. *Programs* can communicate by exchanging messages or indirectly by sharing *Objects*.

Since programs are mobile, and because the Internet is subject to high transmission error rates and long delays, we have opted for an asynchronous model as a means for interaction between programs. This model is based on *events*. Programs and the system can asynchronously throw an event at another program. Events can replace traditional method calls, though regular method calls can still be made between objects in the same program. Events are also used for signaling an exception or an error. Each program present in the system defines a set of *EventHandler*s. An *EventHandler* resembles a method in Java, except that it is executed as soon as the corresponding event is read from the object’s event queue, e.g., Figure 1.

The intuition behind the event model comes from the following observation: in a context of a widely distributed network such as the Internet or when using wireless protocols, reliability and bounded communication times are not granted. Consequently, a rule that should guide the design is that programs should not become entangled. Thus, our model does not allow a program thread to visit objects of another program. Instead, it must use the event model to communicate with the other program. This presents some advantages. For instance when a program moves, it can move as it is: it does not have to check whether other programs’ threads are currently running in its objects. It also prohibits a program from stopping one of its thread running in another program’s object, thus leaving that object in an unstable state.

With regards to distribution, two important programming issues are network awareness and resource localization. Clearly on the Internet where the delays might be long and independent administrative regions exist, a program needs to be aware of its current location [30]. *Lana* provides a class *Location* whose instances represent physical nodes, and between which programs and objects move.

Resource localization can either be transparent to the programmer, which means that the system is totally in charge of keeping track of objects, or else the programmer is in charge of defining his own protocol for locating objects. We have chosen an intermediary approach, based on a best effort strategy. When a program leaves a node, its objects are replaced by *shadow objects*. The role of these objects is only to provide the next destination of the objects they represent. This can be exploited by requesting clients to learn the new location of the moved service. Another use of the best effort strategy is to help objects locate replicae made of them. That is, in order to tolerate disconnections and latency problems objects can be replicated.
and then later merged for information consistency (see Section 2.4).

2.2 Security

2.2.1 Isolation and resource management

Code and program mobility play an important role on the Internet since it permits application functionality to be dynamically and quickly deployed to user sites. However, code mobility also poses security risks: the host platform must protect itself from damages that mobile code might engender to the host and to other running programs.

As with any OS, an InternetOS must provide protection domains. The goal of a protection domain is to isolate a program from other running programs. Each program present on the system runs in its own protection domain. A program can be a “system” program, a user’s program or a program coming from a remote host. Of course, a program can fall into all of these categories. A kernel might for instance might down-load a compiler or code verifier from a remote site; these are typically classified as “system” programs.

Isolation is the property that a program running in a protection domain cannot observe or damage resources possessed by a program running in another domain. This is required to protect against confidentiality and integrity attacks. The OS must provide communication channels between domains. It is the role of the security policy to define and enforce access rights for protection domains for these channels.

Protection domain isolation is traditionally implemented using hardware support. In an OS like NT and UNIX, a process is the abstraction used to delimit a protection domain; isolation is enforced through hardware based address spaces, and controlled communication channels are provided in the form of sockets. The advent of portable languages such as Java has led to renewed interest in software or virtual machines which emulate the behavior of a single machine type on a range of hardware and OS platforms. Virtual machines also emulate protection in

Figure 1 A program based on events and with event handler methods for treating them.
software. In Java for instance, each program can be isolated from others using class-loaders [21]. In this approach, each application possesses its own class-loader object which is responsible for finding and linking classes for that application. Each application can thus have its own linked version of a class. In Java, the classes used by one program possess a distinct type to those of other programs, meaning that an attempt by a program to gain access to an object in another program provokes a type violation failure in the program.

Current research [31, 21, 11] puts a lot of emphasis on the design and implementation of software-enforced protection domains because they provide many advantages over hardware-enforced mechanisms. They allow for finer-grain separation – the granularity is at the object level rather than at the program level – and they also allow for faster implementations - since mechanisms like type checking and program analysis at compile time can reduce the runtime cost of ensuring isolation. Further, the cost of a context switch which forces a processor to alternate between the memory pages of different programs is not paid. In any case, on small systems such as the Palm Pilot where the OS or the hardware do not provide multiprocessing capabilities, a software-based solution is mandatory.

Apart from isolation, another important issue an OS has to deal with is resource management [4]. A protection domain must not consume more resources than it is allowed to and it must not prevent another protection domain from gaining access to the resources that it needs. Such an attack is known as denial of service. CPU time, memory, storage and network bandwidth are examples of resources that are managed by an OS; each domain must be accountable for all resources that it consumes. When resources are shared between domains, the security policy must specify which domain is accountable for that resource.

Sharing is also the source of problems with regards to program isolation and resource management. For instance, isolation provided by Java class-loaders can be broken because Java shares certain system classes among domains. These classes can be used to gain references to objects in other domains [11, 8]. Sharing between domains complicates accountability since it is not always clear who should be charged for the memory consumed by a shared object [4].

### 2.2.2 Lana protection domains

The Lana kernel supports protection domains for programs and shared sets of objects. These two kinds of domain are represented by a `Program` and an object `Space` respectively, and are organized into a hierarchy with the root being the kernel itself.

An example of the hierarchy is shown in Figure 2. The kernel has created two children spaces, S1 and S2, and a child program `Prog1`. A space houses a set of objects; a program is an executable unit. In Figure 2, an object in space S2 creates a child program `Prog2`, and `Prog1` creates two child spaces S3 and S4.

The goal of object spaces is control access between objects. On creation, each object is assigned a space. An object can always invoke objects in the same space. On the other hand, an object can invoke an object in another space only if its enclosing space has explicitly been granted an access right for that space. Access rights for spaces are set by the owner space or program. Objects in a space may always invoke methods of objects in owned spaces; otherwise
the calling object's space must have been explicitly granted the access right. In Figure 2, program \texttt{Progl} has granted (objects of) space \texttt{S3} and program \texttt{Progl} the right to invoke methods of objects in space \texttt{S4}. In Lana, by method invocation we mean the creation and transmission of an event to an object for which that object possesses an \texttt{EventHandler} method.

A \texttt{Program} object is self-contained; it carries its own code archive and can be instantiated by any object. In Figure 2, the kernel has created \texttt{Progl} while an object in space \texttt{S2} has created \texttt{Progl}. Objects of space \texttt{S2} have been granted the right to invoke \texttt{Progl}.

The object space model controls access to objects, but does not control aliasing which is the source of real problems in object-oriented languages as testified by the security problems encountered with Java. For this reason, objects in Lana can be declared as \texttt{private}. An object that is declared as \texttt{private} in a program or in an object space can never be passed outside of the scope of its enclosing program or space.

```java
public class FileMgr extends Program {

    Crypto C;
    File {private} f;

    main(argc, argv){
        Space sp = (Space)argv[0];
        C = sp.newInstance("Crypto");
        f = new File();
        C.encrypt(f); // error signaled !!!
    }
}
```

In this program – written in Java-like pseudo-code – a space object is passed to the new program which instantiates an object of class \texttt{Crypto} in that space. The program then creates a file object and tries to encrypt the contents of the file. An error is signaled in this command since the program is attempting to pass an object declared as "\texttt{[private]}" to another space. Recall that Java contains a "\texttt{private}" modifier though its meaning is completely different. In Java, a

\textbf{Figure 2} A hierarchy of Programs and object Spaces.
private variable cannot be named from outside the class in which it is used; this does not mean that any object bound to that variable cannot become referenced from outside of the declaring context.

A Space or Program object is also a unit of accountability. Both possess a size method which enables the owner space or program to ascertain the amount of memory used by the space. A program is charged for its own objects and for objects in owned spaces, and all spaces and programs owned by these spaces. An owner may terminate a space or program that exceeds some limit.

We have implemented the object space model over Java 2. Access control is implemented by having access control objects interposed between objects of different spaces. These access control objects contain a pointer to a centralized security policy object that verifies each inter-space access. This implementation is described in [8].

2.2.3 Code mobility as a mechanism for security

To assign access rights to a protection domain, one must be able to identify the level of trust of the program running in the domain. This is typically a problem of authentication and, in Java, is treated by comparing a digital signature attached to the program with a list of trusted users. The digital signature is a credential of the program being run. A host must not just be able to authenticate programs, but also the information they furnish. This is the believability security property. Believability is particularly important in an InternetOS context since basic OS components such as compilers, verifiers, etc. might be loaded from remote sites.

Different aspects of this problem have been addressed. Proof-carrying code [24] is a technique where code providers must send their code along with a proof that asserts their safe execution. The receiving host then simply verifies whether the proof attached to the code is valid. Java byte-code verification [23] is a mechanism that ensures that the byte-code of the program received does not violate certain safety properties. For instance, it checks that a variable is not accessed before being defined.

It is clear that a flexible mechanism must exist to aid a host platform enforce believability. In Lana, signed credentials can be attached to a program or object. A credential is any object that can help to authenticate the program or the information within. For instance, it could be a string of the form “The public key of this program’s sender is k”, in which case the credential is signed with the public key of the certificate authority. However, the credential can also be a program in byte-code form; this is known as an active credential. As an example, the active credential of a program could be its proof code, signed by the program owner. The active credential for a mobile agent could be a program that pings a specific host site for kill signals (This verifies the the agent is not a daemon). The advantage of the active credential mechanism is that the verification procedures can be specific to each application.

As with any credential, a key is needed to attach and verify a credential from an object. Our base object class contains the methods:

```java
public void encodeCredential(Object credential, Key k);
public Object extractCredential(Key k);
```
Key is a basic class of Lana with two subclasses: SharedKey and AsymKey to represent symmetric and asymmetric keys respectively. The active credential mechanism has been implemented in a mobile agent platform [7].

Survivability is another security notion that benefits from code mobility. Survivability is the security property that an application tolerate attacks made on individual hosts. This is achieved by replicating a program and sending it on different network itineraries. For this reason, replicate is a basic method of Program. A voting mechanism can then be programmed into the program to sift through replica results.

2.3 Coordination

A user’s machine needs to be able to discover resources on its network, to become aware of new resources and the removal of existing resources. Resource types of interest to an InternetOS include code libraries, servers, mobile devices e-mail files, etc.

Resource discovery is particularly important in the context of spontaneous networks which are formed by PDAs – carried by people – endowed with short-range communication facilities. Here, the resources available to a user are defined by the PDAs and fixed stations in his immediate vicinity. One application of this is a market place in a town square, where sellers “publish” their offers onto the network. These offers are then picked up by buyers on their PDAs who sift through them for the articles they want. Efficient resource discovery is also crucial for the Internet where the number of resources is huge and constantly evolving.

For these reasons, resource discovery constitutes a basic protocol of an InternetOS running on any modern computer. Our approach is based on an extension to the Linda shared space model [12], where processes communicate by placing and retrieving messages from a globally visible data space. Placing a message in the board is an asynchronous operation. Message retrieval is associative: the receiving processes specifies attributes of the message it requires rather than names or addresses. Consider the example of Figure 3. Here several processes exchange messages by placing and retrieving messages from MessageBoard. In Linda, entries on the board are tuples of values of basic types. The Figure 3 example is a market place where two items for sale have been publicized – a TV and a PC. A potential buyer looking for a TV can retrieve offers by specifying to MessageBoard that he wishes only to retrieve a tuple that contains a field “TV” – this is the associative property of the model.

This model is well suited to our needs for two main reasons. First, the associative search for information means that no fixed naming conventions need be followed by processes: the communication protocol is wholly defined by MessageBoard and other Lana classes. Thus, a PDA can become alive in a spontaneous network and immediately participate. A second advantage is that communication between programs is “disconnected”. This means that no expensive socket-like network connections need exist between communicating programs, and that these programs need not be on the same network for a communication to take place. For example, a device can connect to the network to send a message, disconnect itself for a time, and then log in later to retrieve its response.

In Lana, we wish to use an extension to the Linda model that integrates access control
primitives. In order to protect the system from faulty and malicious processes, messages placed on the board can be locked with Key objects – either SharedKey or AsymKey. A message retrieved from the board can only be decoded with a matching Key object. Thus, a message tuple locked with shared key \( k \) can only be unlocked with \( k \); a tuple locked with key \( k_a \) from the asymmetric pair \( (k_a, k_b) \) can only be unlocked with \( k_b \). This secure message board system has been implemented in Java over the JVM; the model and the implementation are described in [29]. The tuples in Figure 3 are all guarded by keys.

The Lana kernel therefore contains a MessageBoard class that represents the secure board. Since an instance of MessageBoard can contain a large repository of information, we do not expect every node to possess one. Nevertheless, a node should be able to find one on its current network whenever needed.

2.4 Disconnected Operation

Throughout this text, we have spoken of wireless and mobile devices. These two terms are sometimes used interchangeably, though it is incorrect to do so [28]. A device can be mobile, without being wireless, and vice versa. In this paper, apart from the standard Internet, we are interested in networks with the following characteristics:

- Network bandwidth is inferior, perhaps by an order of magnitude. Transmission error rates can be much higher.
- Devices can become disconnected for periods of time, or intermittently so. Disconnection can be by choice of the user, who wishes to avoid a costly network connection.
- The network configuration can be highly dynamic - especially in the case of ad hoc and spontaneous networks. This means that the netscape is constantly changing: devices and resources may join and leave the network at any time.
A system or application developer must deal with two issues when designing for this kind of environment. First, an application running on a device that becomes disconnected must have enough code and data cached in order to continue running during a disconnection. Second, after reconnection, the network must be notified of changes made to data since cached copies can have been concurrently modified. Consistency management procedures must merge updates to define a coherent value for the data item.

One of the big questions concerning disconnected operation is whether it should be application and user transparent, or whether the user should be aware of disconnection, in order to program around it. The Coda file system for instance chose transparency [18]. The system caches files locally before disconnection, and a log containing a list of modifications is replayed to the server after reconnection in order to manage consistency. Modern thinking however argues for an application aware approach to disconnection [15, 17]. One reason for this is that it is more efficient and simpler, because the user can specify the data items that need to be cached and then updated following reconnection. This avoids pre-caching of data items that are not used during disconnection.

For these reasons, Lana also takes an application aware approach to disconnection, and offers support for caching and for consistency management.

The first element of caching support is the mobility of programs and objects. This means that programs and data can be moved to sites before they become disconnected from the network. In particular, objects that are moved in Lana are transferred with their class, so the code an object needs is always available. This is in contrast to Java, where an object moved between sites is serialized for transmission. The serialized form of an object is simply a byte array copy of its data and the data of referenced objects.

A second element of caching support comes from object groups. Whenever an object is moved, then all objects in the same group are automatically moved along with it. This approach allows users to specify caching policies. In Lana, the group concept is implemented using spaces: all objects of a space belong to the same group. When an object of one space is moved, all other objects of the space are moved with it.

Regarding consistency management support, classes in Lana can be defined to implement the Replicable interface. This interface defines two methods for this: createCopy and checkConsistency. The createCopy object creates a copy (replica) of an object – which can be moved to another site. Object replicas can be manipulated independently, though the original copy remains the master copy. When a checkConsistency method is executed on a replica, the object attempts to contact the master copy via the location field - again using the best effort strategy. The checkConsistency method is a method to allow a programmer to specify his own consistency management policy – a prerequisite for optimistic concurrency control. Note that the message board can also be used by objects for locating and communicating with object replicas.
2.5 Performance

Since our platform is not only targeted at desktop or server machines but also at handheld computers, performance expectations may vary. For instance, for a handheld computer such as the palm pilot, it may not be adequate to use a Just-In-Time compiler (JIT). The reason is that the implementation of the JIT in addition with the memory footprint generated by the runtime compilation process consumes too much space. Therefore, it is more sensible to use a finely-tuned interpreter [25]. In addition, since an interpreter is simpler and faster to implement than a JIT, we have chosen this approach for the first prototype of the Lana platform.

An option to further improve performance in terms of space consumption on machines with small memory is to compartmentalize the virtual machine. The idea is to delegate certain tasks dedicated to the VM to a remote server with greater capacity. For example, security-related tasks such as verifying the integrity of the code or authenticating its origin could be performed on a trusted node on the network rather than on the handheld.

On a desktop or server computer, it is certainly reasonable to use a JIT and is even desirable for performance. Obviously, it is possible to run the totality of the VM on the same machine. However, this does not mean that the idea of delegating some tasks of the VM to another node or to mobile programs should not be exploited [27]. Indeed, some computers could share the same trusted remote computer to perform the security-related tasks or even to perform the code just-in-time compilation. Delegating a specific task to another node or program has the advantage of simplifying management of the platform since each module is clearly separated from the rest of the system and duplication of tasks on different nodes does not exist. From a software engineering point of view, system components can be added at will. This is the same argumentation used for the development of micro-kernels from operating systems [26, 22].

2.6 The Language

As mentioned, we are building an Internet Operating System kernel that takes a programming language approach. This means that support for key properties such as security, disconnected operation and coordination is included in the semantics of the language, and/or the base classes from which all other classes inherit. We have outlined the main classes and interfaces in this section; we summarize here.

![The hierarchy of classes in Lana.](image-url)
Object is the root class, which implements basic functionality such as locking and credentials encoding, and which all applications inherit from. The Event class is the basis for message exchange between objects. An event can be a method call, a return message from a method, an OutOfMemory error, etc. Events are serviced sequentially by a thread.

A Program object contains a method main. When a program is instantiated, a thread is created that executes main, and which then awaits events. A method is defined for each event that the program expects. This thread can issue method calls on objects in owned spaces, or issue events to remote objects and spaces, so long as it possesses the required access rights. All methods that modify state are synchronized (executed in mutual exclusion). An object being modified by a method is copied onto the stack of the calling thread, and copied back to the heap when the method terminates. In this way, the failure or termination of a thread cannot leave an object in an inconsistent state.

The Space class represents an object set. Objects in a space possess the same mobility and security constraints: Access rights are granted to spaces rather than to individual objects, and moving an object to another site implies moving all other objects of the same space.

The MessageBoard class implements the associative shared data space repository. Objects placed in this space are locked with Keys – either SharedKey or AsymKey – and locking can be implemented using typing or encryption depending on the level of trust that the user places in his environment.

The Location class represents a site position. Finally, the interfaces Replicable and Mobile denote mobile and replicable programs and objects.

Our original goal for the project was to stay as close to Java as possible for compatibility reasons. However, Java contains too many features that are incompatible with what we want to do: the thread model, the lack of resource control, the absence of program mobility, the lack of support for disconnected operation. On the other hand, Java does have a rich class library with a number of useful services. Our approach is therefore to have a version of the Lana kernel that runs within a JVM. Lana programs can communicate with the JDK via bridge objects that we provide. The bridge possesses converters that transform objects of Lana basic classes, like Object and String, to and from their Java counterparts.

We have started to develop the Lana kernel, based on the object model of Figure 5 and a minimal set of byte-code instructions for which we are programming an interpreter in Java. We chose Java because of the convenience of prototyping with that language. For efficiency reasons, the interpreter will later be rewritten in C once the core design is stable. We will then add components to the library that are typically found in the JVM, e.g., byte-code verifier, compiler. Our aim is to demonstrate that the language contains enough features for Internet programming and can be used to program Internet applications. The design is preliminary, and choices may change with experience gained in the implementation. Our goal here was to present a reasonable set of abstractions that can be used for a first prototype.
3 Past, Present and Future Work

The Lana InternetOS kernel will be used in several projects currently underway in the Object Systems Group. One of these is the Active Business Objects project [19] whose aim is to exploit mobile agent technology to build and use business information systems. In this way, new business procedures can be dynamically added to a system, and some practices, like partner searches can be handled “off-line” by agents.

The goal of the Dilemma European Union funded project is to maintain a distributed database that contains technology offers. This system uses mobile agent technology for basic tasks such as searching for offers, maintenance of administrative information at the various sites. The importance of agents in this case is that many tasks can be executed by the user while he is off-line. This is a significant advantage for a database distributed over several European countries, where communication latency is costly.

The benefit of Lana to Active Business Objects and to Dilemma is that it directly supports the mobile agent concept. In particular, it offers security guarantees needed to protect host platforms and executing code from malicious agents - the protection domains implemented by the space mechanism, and the active credentials authentication mechanism.

The kernel developed will reuse experience and code from existing systems developed in our group. The JavaSeal system is a mobile agent system that runs over Java 2 and which offers the ability to displace Java programs (agents) between network hosts. One of the key goals of the system is security: agents run on the platform are isolated from each other and can only communicate via system provided channels [9]. Implementing JavaSeal taught us much

![Diagram of object and class structures in Lana](image_url)

**Figure 5** The object model in Lana.
about the problems in implementing secure systems over Java. There is no support for resource control, aliasing is still a big security risk, agents need a simple communication infrastructure and there is too much happening inside of the JVM over which we have no control.

Access control is a key requirement of our new kernel. The problem of aliasing led us to investigate an access control mechanism for objects that separates the naming of objects from the ability to invoke methods on the objects [8]. Each object is assigned to one of several spaces, and all method calls between objects of different spaces are controlled by a security policy, though no attempt is made to control aliasing between object spaces. The issue of agent communication led us to extend the Linda programming model with access control primitives. Each object placed in the space is locked using keys, and can only be retrieved by an agent yielding the correct key [29]. This model was implemented in Java.

Current work on virtual machine design is oriented towards efficiency and minimal size. Several Java VMs have appeared over the last few years that were designed with efficiency as their main goal, e.g., HotSpot [2], Jalapeno [1]. JavaSoft have produced a micro edition of the Java development environment centered around the K virtual machine. Designed to run on hand-held devices and smart-cards, the micro edition differs from the standard JDK by the absence of many class libraries and methods from the base hierarchy. Bershad et al. describe the design of a distributed JVM in [27]; this has similar goals to Lana in that JVM is decomposed into orthogonal interacting modules, where modules need not execute on the same site. It is clear from all of these developments that Java is still the major force for Internet programming. It is for this reason that we insist in the Lana VM being able to co-exist and to interact with a standard JDK.

References


