

The Runtime Architecture for A New Component Programming Language

Thesis Proposal

Xiaoqi Lu

Advisor: Dr. Scott Smith
Department of Computer Science
The Johns Hopkins University

December 2003
(Revised Feb 2004)

Abstract

As the networked computation becomes prevalent, software development becomes more complex. Moreover, the demands for component composition, reliable networking, and security become essential. Unlike common implementation techniques, we propose to address these issues using a programming language-based approach, specifically by designing a language with built-in common network abstractions and security mechanisms. This proposal focuses on the runtime architecture design of our proposed language, in which the basic building block is called *cell*. We first discuss our research motivations, the basic design of cells, and the corresponding prototype implementation we have completed. Then, we present our proposed research for completing the full picture of the cell runtime architecture: the *system service model* that encapsulates the local system resource management via cell *connectors* (typed interfaces of cells), and the *strong service persistence model* in which the connections between cells can have strong persistence with build-in failure detection and recovery mechanisms.

1 Introduction

In the rapidly changing computing world, several major trends become more prevalent. First, as code complexity and program size grow larger

and larger, new programming paradigms for constructing large-scale systems via code reuse and component composition become increasingly essential. Secondly, the computational infrastructure is constructed in a widely distributed yet loosely connected way, with resources scattered in decentralized machines. Thirdly, security-related concerns are becoming a critical priority. Every practical system has to address security concerns in one way or the other.

From the programming language point of view, current available programming languages are not yet adequate for solving all the issues, in the sense that they are designed fundamentally for standalone computers and traditional programming paradigms. Hence, programmers usually have to go outside of the language to solve common problems. Our approach is to look for a solution to these concerns via built-in language abstractions, by designing and implementing a new general-purpose programming language tailored for today's computing environment. We believe a language solution is a better way to solve many problems because programming languages sit as the base of every computation system and handling problems from the root will lead to more elegant and reliable answers. With this proposed language, application developers can concentrate more on software systems' functionalities instead of on low-level details of tackling each of these individual problems. This research proposal, as a part of our programming language design project, mainly focuses on the runtime aspect of the language, specifically, the design and realization of the runtime architecture for our proposed programming language.

2 Major Concerns

Our language design aims to seamlessly integrate static code reuse and dynamic runtime composition. *Software composition can take place at different times during the lifecycle of an application. According to the time of occurrence, there are two basic types of integration mechanisms: static composition and dynamic composition. The static composition focuses on code reuse performed before the execution of an application. This approach allows developers to specify the building blocks of a system at design time and assemble them statically. In contrast, the dynamic approach is to integrate executing components so that the parts to be composed can be decided at runtime. Traditionally, dynamic composition refers to runtime library loading from local machines, such as Java classloader [Lia98, GJS+] for dynamic class loading. Our research studies a different form of dynamic composi-*

tion: dynamic linking between runtime components. This linking is built on abstraction of persistent connections, locally as well as across networks. In general, the static reasoning and checking of an application's nontrivial properties helps users understand and hence reuse the software. The static approach achieves crucial efficiency by saving the work that otherwise has to be done at runtime. On the other hand, the dynamic composition capability of a software application is essential for system flexibility so that systems can be updated and extended at runtime. Our research is closely related to both module systems like Units/Jiazzi [FF98, MFH01] and component technologies such as CORBA/COM/JavaBeans [Szy98, Omg, Mica, Sunf]. Module systems emphasize the capabilities of expressive code assembling in a rather static way, while component protocols focus on the runtime service discovery and the component interoperability.

Network computation is also one of our major design concerns. The decentralized characteristic of distributed systems demands ever-higher application scalability, communication consistency, system fault tolerance, and security requirements. Many disparate domains of research have studied different aspects of network-related issues. For example, Java's RMI [Suna, Sund, Hor01] mechanism implements an integrated remote/local invocation semantics; CORBA and DCOM [Omg, Micb, Szy98] are designed to achieve the unanimous invocation and the service discovery; The E language [Mil] is a programming language designed for developing secure distributed systems. We believe a proper language abstraction is a good solution to network problems since it can automate common network programming tasks such as parameter marshalling, socket and request demultiplexing, fault detection/recovery, and hence saves programmers a lot of work of low-level error-prone programming. One of our goals is to provide language-level abstractions for network communications via explicitly declared pluggable interfaces (These interfaces are declared statically using our proposed language, and they are pluggable in the sense that persistent connections can be established and destroyed on them dynamically at runtime). This differentiates our study from other related works.

In distributed programming models, security is usually an extra layer of protocol added on top of the core specification. For instance, CORBA SEC [Omg] is proposed as the security specification built on the core CORBA. Although a layered architecture is a good design pattern, we believe that security is a special aspect of software systems that crosscuts every component of the whole system. As a result, security can be better addressed if it is a built-in language feature. For example, the Java guarantees the memory safety of Java applications, which otherwise has to be taken care of

by programmers using languages such as C and C++. Furthermore, since most layered security protocols are framework guidelines, how well a security system actually works depends on programmers' understanding and implementation, which usually leads to many practical problems. Hence, we want to extend the Java memory protection model to other system resource management issues, and at the same time to improve security policy expressiveness so that applications will be able to customize fine-grained, flexible security policies. More specifically, we want to model both system resource level and application level security on the pluggable style interaction channels between software components.

An explicit component runtime architecture is an integral part of our language proposal. The basic idea is to build an administration layer between low-level operating systems and high-level user applications. This layer provides the execution environment for applications, takes control of local system resources and is responsible for the safe application isolation. The functionality of the administration component is similar to the Java Virtual Machine (JVM) [Sunc]. Unlike the JVM technique, we model the interactions between the system administration and the applications via statically and explicitly declared interfaces, which entails a clear-cut boundary between the system management domain and the user application space. Such a runtime management model has the advantage of disallowing escapes from the administration control. From a middleware perspective [SS02], the runtime administration layer is a middleware that encapsulates operating system communication and resource management.

3 Initial design and the prototype project

We initiated our language development by first focusing on the runtime model. In this section, we review the initial language runtime framework and the prototype implementation of the architecture that has been completed.

3.1 Design Framework

The basic building blocks of our component-based programming language runtimes are cells [RS02]. They are essentially autonomous runtime entities and deployable containers of objects and codes. Cells expose type interfaces (*connectors*) that import (*plugin*) and export (*plugout*) operations or classes. Cells can be dynamically connected and disconnected, locally or across the network via these *connector* interfaces. In Figure 1, "MusicStore" and "MusicClient" are two example cells. "MusicStore" cell has a

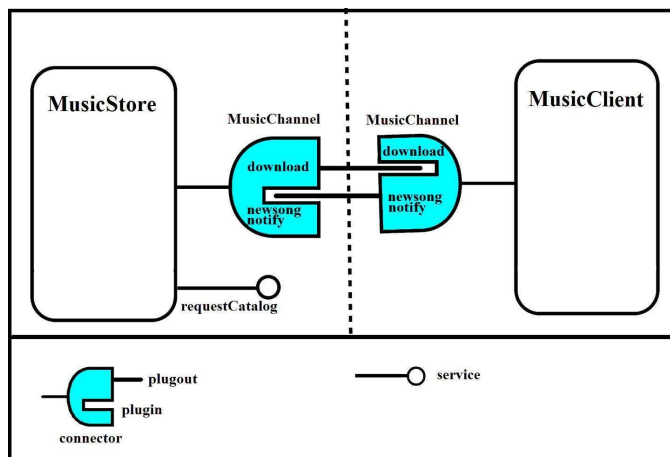


Figure 1: Two cells communicating via a connection on connectors

connector named “MusicChannel” with a plugout (“download”) and a plugin (“newsongnotify”). “MusicClient” also has a “MusicChannel” connector but with reversed type of plugin/plugout. The dashed line between the cells indicates that they may be local or remote to each other. The connections established on connectors (we refer as *plugged connections*) are peer-to-peer persistent runtime links that are formed based on the mutual agreement of the two cells involved. In Figure 1, the two cells have a plugged connection on their matching “MusicChannel” connectors. Cells also provide standard client/server style interfaces (*services*) for local or remote invocations, such as the “requestCatalog” service of “MusicStore” in the figure. Cell service interfaces are designed for normal one-time lightweight invocations, while connectors are for establishing long term and persistent communications. Cells run on *Cell Virtual Machines* (CVMs) that are represented by a special type of cells — president cells. CVMs play the role of forming the execution environment for cells. They are responsible for loading cells into memory and unloading them back to storage upon request. A CVM may have one or more cells running within it. Due to their strong independence, cells are fundamentally mobile entities that can be unloaded, transferred over the network, and reloaded in a new location.

The design of the cell architecture captures the essential concerns we discussed in the previous section. First, cells are strongly isolated and cannot interfere with each other’s computation. And, with carefully designed connector and service interfaces, cells can only interact with one another in

delineated ways. With a clean-cut component boundary, cells are capable of being deployed as software products, being reused and composed by legal third parties. Secondly, connectors and services of cells wrap the details of the low-level network communication. Plugged connections are long lasting as compared to service invocations. A single language statement triggers a connecting process on connectors. Once the process succeeds, the connection stays for continued use. This connecting procedure also implicitly performs the possible authentication checks and interface mappings since the premise of a successful return of the procedure is the mutual agreement between the two cells involved. The persistence of plugged connections also enables installing per-connection based security policies. Thirdly, CVMs sit between the underlying operating system and local application cells and enforce the isolation of the local cells and the protection of system resources. Moreover, every cell is a unique entity in the sense that it has a persistent identity. They can serve as principals in a security model.

In addition to the related work we discussed in the previous section, the mobile aspect of cells is also related to the research on mobile computation such as Seal [VC99, VT97]. Moreover, ArchJava [ACN02, ASCN03] has a similar notion of connectors but its focus is the communication integrity mainly on the object level.

3.2 Prototype Implementation

We launched a project to implement a prototype [Lu02] of the initial cell runtime architecture so that we can test the feasibility and practicability of our ideas. The cell prototype was built on top of the unmodified Java Virtual Machine. As discussed in [Smi02], cells in the initial project were prototype-based components similar to objects in Self [US87]. We gained valuable experience from this project and discovered some problems that in turn led to significant changes to the initial design. In the following section, we briefly introduce the prototype implementation.

3.2.1 Cell Identity

Every cell has a unique unforgettable cell identifier (CID) that addresses a cell universally. Part of a CID indicates the CVM that issued the CID. CIDs were designed to be cells' public keys but we simplified them to be readable string names with assumed uniqueness in the prototype.

CellReference is a class whose instances are runtime handles to cells. A cell reference object stores information for locating a cell locally or over

the network and it hides the difference of possible remote/local access context when referring to the cell. A cell reference has two important properties: first, it contains the location information about where the referent cell might be residing and its unique CID; secondly, it may contain cached information for fast direct access to a cell. The direct access can be either a local hard reference to a local cell or a stub reference to a remote cell. If a cell reference becomes out-of-date, the update can be done based on the location information it stores. The Cell Virtual Machines are responsible for resolving cell references upon request.

3.2.2 Cell Syntax

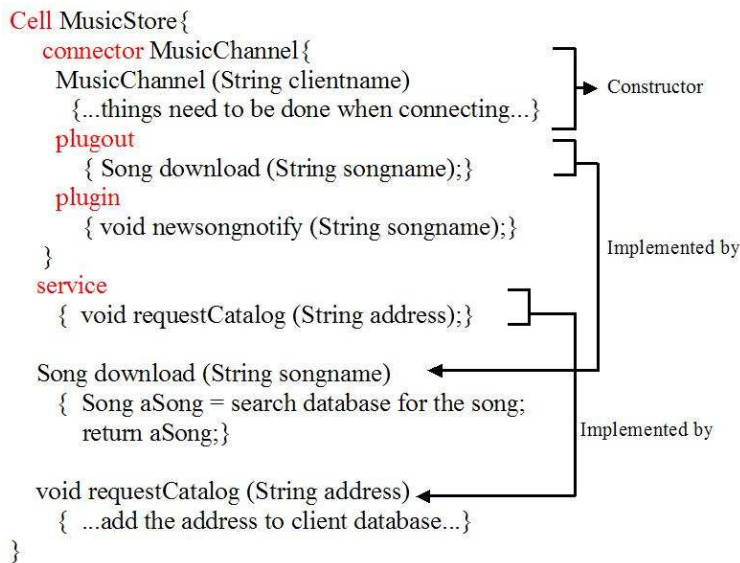


Figure 2: Source code for MusicStore

We have the primitive Java-like language syntax for programming cells. Figure 2 shows the source code of the “MusicStore” cell. As in the code segment of Figure 3, the “MusicClient” cell uses **Connect** to link with “MusicStore” and download a song. The returned connection is stored in a local variable **myStore**. We implement a language preprocessor that parses a cell source file, translates the extra syntax into Java code, and creates other necessary Java files such as stub files. Then the Java compiler compiles all the generated Java files.

```

Connect MusicStore @ MusicChannel ("MusicClient") as myStore;
                                     └──────────┘
                                     provide the parameter required by MusicChannel constructor

if (myStore != null){
  Song aSong = myStore.download("Sky"); ← invoke plugin
  Disconnect myStore;
}

```

Figure 3: MusicClient code segment

3.2.3 The Cell Virtual Machine and the President Cell

In the prototype, every CVM is represented by a *president* cell. President cells have special privileges to perform functions such as monitoring network activities. The implementation for president cells is part of the cell core package and users cannot alter the code arbitrarily. In the future discussion, we use the two notions, CVM and president cell, in an interchangeable way if there is no confusion.

A president cell is always the first cell loaded into a CVM by a bootstrap program. Then it takes over the control of managing the local CVM. In the prototype, the CVM and its president cell do not have much resource control capability yet. They mainly fulfill basic functionalities:

- Only the president can handle requests of loading other cells. To load a cell, the president first finds its signature file, checks its structures, then constructs and initializes the cell into the runtime.
- A CVM keeps a name registry that records the cell references of all cells loaded locally. It also provides a well known **lookup** service for queries. The references to president cells are always published on the local well-known registries, implemented via the Java `rmiregistry`.
- A cell can request its local CVM to resolve a stale cell reference. This process may involve contacting CVMs on other remote machines.

3.2.4 Cell Interactions

The simplest cell interactions are those on client/server style service interfaces. The semantics of such invocation is identical to a normal local method call in an object-oriented language. First, a cell reference to the cell

should be obtained and then this cell's public services can be invoked with proper arguments.

The more interesting type of interactions is the ones on the peer-to-peer, plugging-style connections. Connecting on connectors is an operation explicitly invoked at runtime. To make a connection with another cell on connectors, a cell first locates a cell reference to the other cell, and then issues a **Connect** statement. A valid connection is returned only if the types of all the plugin/plugout pairs on the two connectors match and the two cells involved agree on this connection. After the connection is established, any plugin operation invocation is handled by its connected plugout. The connect protocol is asymmetric: one cell initiates the process and the other cell responds accordingly. A plugged connection stays alive until one of the involved cells shuts it down with a **Disconnect** statement.

We developed the prototype runtime architecture in both local and distributed environment, on which we had the opportunity to analyze details of our initial language design. Some problems were discovered and some refactoring was made accordingly. For instance, originally we had class plugin/plugouts on connectors, which means cells can import codes at runtime via connectors. Then we found this introduced more complexity than benefits. Therefore, we refactored our design by removing class plugins/plugouts and adding an assembly layer. Eventually, assemblies will be the code aspect of cells, which is studied in one of our other projects.

4 Proposal for future research

With the experience accumulated in our previous research, we are confident about the theoretic and practical importance of our proposed language model. However, this research is still in its early stage; more in-depth work remains to be done. In the following section, we discuss the major directions we plan to pursue in future research.

4.1 Service model of the Cell Virtual Machine

4.1.1 The Service Model

The CVM, as the application execution environment, is an essential part of the runtime architecture for cells. It bridges the cells with the underlying operating system and provides runtime support. It also provides a machine independent platform in a heterogeneous networked setting. However, in

the initial implementation, the CVM only had a few basic services such as **lookup**, and otherwise was an unmodified Java Virtual Machine. Obviously, the architecture of the CVM has a substantial influence on our language design because it is the foundation of the cell runtime model. Our major concern in designing the CVM architecture is that it should enable CVMs to interact with operating systems and cells in a secure and efficient way. Before introducing the service model of the CVM, we make some discussions about the Java runtime model first.

Java is a safe language in the sense that it enforces memory management and protection for Java programs. It adopts the sandbox security model that makes it possible to run some untrusted codes safely. Nevertheless, this sandbox model [Gong] is not always desirable. For instance, Java applets usually have very restricted access rights, which greatly limits their functionality. To support legacy codes and applications with high-performance requirements, Java offers the Java Native Interface(JNI) [Sunb]. Using JNI, Java applications can incorporate native codes written in programming languages such as C and C++. JNI is very powerful, but also problematic. First, with native codes inside, a Java application is not machine-independent anymore, losing one of the main advantages of Java. JNI introduces a back door through which a Java application can bypass the JVM. Malicious applications can circumvent the security control enforced by the JVM using JNI easily. Even though the bytecode instruction set enforces a safe and structured memory access model, an application can detour the restrictions via native methods.

We believe that a major cause of the native code problem in Java is that the JVM itself does not provide any explicit access interfaces to low-level operating system APIs: the only means to use the native APIs is to go around the JVM using JNI. Furthermore, Java itself, including its APIs and the JVM, is implemented as a mysterious mixture of native codes and Java codes, which is a black box for programmers. To avoid the black box, researchers have to build their own JVM implementations or find a open-source one such as [DA01, Kaf, Ibm].

In the CVM service model we propose, all system native codes are considered part of the CVMs, and the CVM exposes the native APIs via explicitly declared connector or service interfaces. This means all native code uses are through either service or connector interfaces of local CVMs. Moreover, CVMs guard the system resources by placing them behind the connector/service interfaces. Figure 4 gives us a simple demonstration of the service model. The CVM offers a connector “FileIO” with low-level read and write operations on a file. The constructor of the connector requires a file name. The result of a **Connect** statement on the “FileIO” connector is

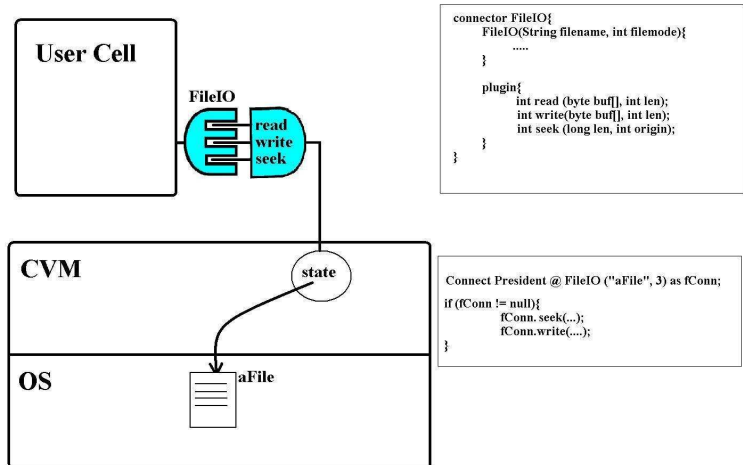


Figure 4: The Cell Virtual Machine Service Model

similar to a call to the file **open** method in Unix. If the user cell is allowed to access the specific local file, the CVM then permits the establishment of the plugged connection through which the cell can operate on the file. Unlike the Unix file descriptor, the plugged connection does not give the user cell a direct reference to the file. Instead, only the CVM has the direct access to the file and it just reserves an access channel for the user cell. If anytime later it wants to revoke the user cell's access right on that file, the CVM can **Disconnect** the connection. This indicates that our connection-based access mechanism is in a sense similar to the capability model but with built-in revocation mechanism. In addition, unlike the traditional capabilities, a plugged connection cannot be transferred beyond a cell boundary in any way. Within a cell on the other hand, the plugged connection can be passed around at will. In this aspect, our language has a built-in capability model that we refer as the connection-as-capability. The discussion in [MYS03, Mil, HCC97] argues that a capability model can be as powerful as Access Control Lists (ACLs) and it avoids problems like the confused deputy problem. The E language [Mil] is a capability-based language. Compared to its ad hoc object-reference-as-capability approach, ours is more expressive and declarative.

In the service model architecture, application cells can directly have access to the low-level codes without escaping from the CVM runtime environ-

ment and its security control. The explicitly declared interfaces of CVMs serve as guarded channels and they are the only doors through which legitimate users can access resources. To implement the service model, local libraries need rewritten to exclude native codes and to add the declarations of corresponding connectors for interacting with local CVMs. As a result, applications can still take advantage of the high-level language features by using the libraries. Another advantage of this service model is that the interactions with the native APIs are encapsulated as plugged connections, where we are able to place a stricter and more fine-grained security control. Moreover, we believe that the security architecture built on this model would be more efficient and safer in the sense that privileged security checks can be performed on a per-connection basis, instead of per-instruction basis. Furthermore, CVMs are now the only entities that need privileged operations.

The big challenge in developing the service model is that all low-level interactions with operating systems need taken care of, such as the graphic device management, the threading and signal mechanisms. Language libraries need to be carefully re-designed and re-implemented. In addition, we need to prove formally that the service model can indeed achieve the desired security compared to other mechanisms. A complete implementation is needed to show the practicability and efficiency of the system.

4.1.2 Comparison of stack inspection and the service model

In the JVM access control system every class is owned by some principle with certain set of privileges. The JVM libraries belong to the system domain and are shared by all applications. This library architecture and its sharing mechanism introduces a serious security problem: an untrusted application might be able to perform unauthorized operations by making indirect invocations via some other system code it has permissions to use. Figure 5 demonstrates the problem. The “UserApp” is not allowed to access local files, but can use the “setEnvironment” system method. If the “setEnvironment” actually writes to the local “hostProperty” file, then the “UserApp” can gain access to the local file by invoking the “setEnvironment”, without violating any security constraints. In this example, “setEnvironment” is a confused deputy. It has authority stemming from two sources, the untrusted code (“UserApp”) and itself. To solve this, Java system libraries need a way to distinguish between the calls originated from untrusted codes and those from trusted ones [WF98]. Therefore, the Java runtime system relies on its stack inspection mechanism to examine the runtime stacks, and makes the security decision at runtime. The stack inspection will deny the invoca-

tion chain in Figure 5 unless the “setEnvironment” explicitly scales up the permissions of the “UserApp” by performing a doPrivileged operation.

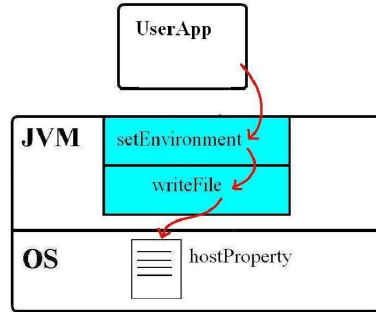


Figure 5: Confused deputy problem in JVM

In the cell model, every cell is an isolated entity. Even two cells initialized from the same code don’t share any states as the objects created from one class do in Java. In essence, the cell model enforces not only runtime isolation but also strict code separation. From the standpoint of a cell, it exclusively owns its code including system libraries that it uses. As described in the CVM service model, all native sensitive operations are provided via connector interfaces by local CVMs. Figure 6 shows the same scenario in Figure 5, which however has been refactored into the CVM service model. The CVM has plugged out the **write** operation in the connector interface. When a connect command is issued in a library code on behalf of the owner cell, it is the identity and privileges of the cell that are checked by the local CVM to grant or deny the connection. Hence, if the “UserApp” dose not have any file permissions, it then cannot access local files either directly or indirectly. In the CVM service model, we no longer need the stack inspection, but still achieve the same level of security as provided in the JVM. While the stack inspection is a remedy for the confused deputy problem in the JVM, the service model with library separation and the connection-as-capability architecture is the natural solution to such problem.

Compared to the ad-hoc stack inspection, the CVM service model has an explicit and declarative interface architecture. In addition, the security checks enforced by the CVM is more aggressive in the sense that they are performed at the spot of a connect request, instead of tracing back the stacks passively. The CVM service model is more efficient as well since once a

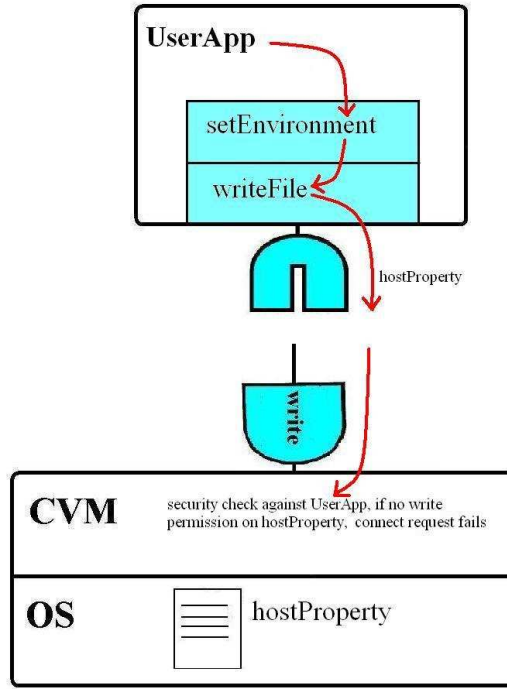


Figure 6: Library architecture in the CVM Service Model

connection has been established, there is no need to do repeated checks on invocations coming from this connection. In contrast, the stack inspection has to do security checks on a per invocation basis. Moreover, cell connections are revocable capabilities, which enables the system to adjust its security policy dynamically and correct mistaken security decisions to avoid the damages to future computations. On the other hand, the Java stack inspection primitives such as the `doPrivileged` are hard coded and cannot be revoked unless the code is rewritten, recompiled and reloaded. Therefore, any careless use of `doPrivileged` may result in uncontrollable leverage of permissions.

4.1.3 Web Services and the CVM service model

Web services are in reality simply XML-based interfaces to businesses, applications, and system services. The web service model became popular mainly because it is a simple and scalable high-level protocol [Kre01, W3C].

It hides the heterogeneity of operating systems, programming languages and communication protocols. A web service can be created and deployed by a web service provider using the programming language and platform of the provider's own choice. The provider defines the web service, usually using WSDL [CCM01], and registers the service along with its definition somewhere, possibly in UDDI registries [UDD03]. The service definition serves as a specification others need to follow in order to utilize this web service. UDDI registries play the role of traditional Yellow Pages where providers can publish their services and potential users can look for the services they need. Once a user find the needed web service, he can code his client application according to the published service specification to interact with that web service. In short, the web service model provides a set of high-level specifications for service description, registration and discovery.

The nature of web services makes them suitable for loosely coupled applications. The interaction between a web service and its clients can be ad hoc and very diverse. In addition, web services are essentially designed and deployed in the client/server structure. On the other hand, cells are more appropriate for tightly coupled applications because they are capable of communicating via persistent connections. This also enables cells to support a more peer-to-peer interaction model other than the traditional client/server pair. Moreover, the CVM service model is more than an architecture design. It has an integrated language foundation, which gives it the power of language expressiveness not found in web services.

The scope of CVM service model sits lower in the computing system hierarchy as that of the web service model. While web services are a high-level network solution at application level, the CVM is the middleware built on top of operating systems for efficient and secure system resource management. Essentially, the two models address the issues in distinct computation domains.

4.1.4 CVM, JVM and CLR

The CVM provides a system-independent executing environment for cells by constructing a middleware layer on top of the underlying operating systems similar to Java JVM and Microsoft CLR. The JVM and the CLR are essentially the same [Gil02, Sin03, Gou01]. However, they have different objectives and implementation techniques. The CLR works on Windows only, but theoretically supports development in any language. The JVM, on the other hand, enables Java programs to run on different platforms. Our CVM also aims to support heterogeneous platforms. However, the CVM adopts

the novel connector-based architecture to model interactions between applications and the local system. This connection-as-capability service model achieves securer and more efficient system resource management. Here, we briefly review some related terminologies in the JVM/CLR and the CVM for clarity and future reference.

	JVM/CLR	CVM
Language	Object oriented programming	Component-based programming
Object	Everything is an object	Cells encapsulate objects
Reference	Object references can be passed around but not always valid	Cell references are universally valid
Identity	Objects have no persistent unique identities	Every cell has a persistent unique identifier, CID
Interface	Common interfaces for service invocations	Pluggable (connector) interfaces and service interfaces
Isolation	ClassLoader and soon available Isolate APIs in Java; application domains in CLR	The CVM enforces the isolation of every cell uniformly
Isolation Communication	Java RMI and Isolate APIs; CLR remote invocation APIs	Inter-cell interaction via declared interfaces can be remote or local at runtime, depending on invocation semantics

Table 1: Comparison of JVM, CLR and CVM

Cells are analogous to the objects in the JVM or the CLR. They are containers and owners of objects. Object references cannot go beyond the boundaries of their owner cells. Every cell has a unique identity: the CID, which is universally valid. CVMs keep cell references up-to-date at their best effort so that in most cases a cell reference is valid even if it is passed crossing address spaces, for example, over the network.

*As indicated in Table 1, library APIs in the CLR or the JVM enable programmers to code computations intended to run in an isolated way. Programmers need to explicitly create and destroy isolated domains. The communication between isolated parties is rather primitive. For example, in the CLR, programmers declare *RemoteType* for remote invoke-able operations, and create *Channels* for carrying remote invocations. In Java, two isolated programs*

may create a *Link* object to send and receive messages/events. These inter-isolation interactions are hard coded at developing time so that interfaces designed without remoteness cannot be invoked remotely at runtime. On the other hand, in the cell model inter-cell interactions are uniformly conducted on declared service or connector interfaces. No special interface for the remoteness such as the *Java.rmi.Remote* is needed. An invocation on a cell interface can be either remote or local at runtime because CVMs take care of details of the dynamic call semantics. Therefore, programmers don't need to handle remote computation explicitly in their programs. In this sense, the CVM is a network-aware middleware. CVMs also enforce a full-featured code separation discussed in section 4.1.2. In neither the JVM nor the CLR, there is a similar concern of such code separation.

4.2 Service persistence

Real world networks consist of heterogeneous components and are subject to dynamic changes, especially with the involvement of wireless devices such as laptop computers and PDAs. The dynamic characteristics of networks introduce flexibility but also bring in challenges for software development because network changes always have an unpredictable impact on distributed applications. For example, a network linkage breakdown may cause a route change in the communication between components. However, programmers do not usually care about the low-level package routing. Their concern is mainly whether the needed communication can be carried on instead of how. Preferably, programmers should not have to handle network changes that have no effect on the application functionality, which our language model tries to achieve. In our language, we aim at the persistence abstraction. Persistence is an essential property for improving fault tolerance and the quality of service of distributed systems. The common solution to service persistence is implementation-based in which developers need to program some protocols explicitly. Our study is to provide a language abstraction for persistence. The E language [Mil] has some extent of persistence in the sense that objects can retain unforgettable identities over several runs. Our language design is to provide not only the life-long cell identities but also *persistent states of cell connections*.

Compared to invocations of service interface, cell's plugging style connections inherently have some form of persistence for providing service continuity. However, to improve the fault tolerance, the quality of service and the system flexibility, this type of persistence is not adequate because the plugged connections are lost if the execution or the communication of any

application involved is interrupted. For example, consider a server cell that serves several local and remote client cells via plugged connections. The connections between the server and the clients may be cut by broken network links. Even if the broken links come back shortly and the exact network topology restores, the previous plugged connections cannot recover. Furthermore, for cells running on mobile devices, the migration of those hardware devices also causes service interruptions. In most of these scenarios, the interruptions are not fatal and permanent, but they are usually visible to applications and are left to programmers for explicit handling. *We divide the persistence problem in the networking environment into two levels: the persistence of plugged connections, and the persistence of other computation tasks over the connections. Our research tries to solve the first problem — the resumption of interrupted plugged connections.* What we want to achieve is a proper language abstraction of the persistence so that our programming language can have a built-in comprehensive solution to repair the previous existing connections automatically when possible. *Reconstructing computation associated with these persistent connections, such as problems addressed in mobile computing [Cze99, VC99], is out of the scope of our research. Our model is a low-level platform on top of which more sophisticated high-level consistency can be developed.*

Our initial idea is to enable programmers to declare a connector using the keyword “persistent”, then the language runtime makes efforts to keep the validity of the connections on such declared persistent connectors, even if the cells involved crash or migrate. To realize our designed language-based persistence, the protocol for failure detection and the mechanism for recovery from failures need carefully addressed.

4.2.1 Failure detection

Failure detection in distributed systems involves extensive underlying communication because constant information exchange is usually needed to detect a failure on the network in a timely manner. In addition, the detected failure needs to be aggregated over the network efficiently. Most of existing failure detection schemes such as those in SDS [Cze99] and JINI [Sune] rely on the announce/listen model originally proposed in IGMP [De89]. Briefly speaking, they use dedicated naming/lookup servers to constantly monitor the responsiveness of services in their domains, and thereby decide whether the services are still alive or not. They focus mainly on the detection of one kind of failures—the shutdown/crash of services. DCOM [Micb] adopts the same announce/listen model and uses a modified ping protocol to help

servers detect the failure of their clients. Our goal is to detect not only the failures of cells (service/client shutdowns) but also hardware failures related to the provided services (i.e. any error of a printer may be considered as a failure to the printing service that needs the printer). CVMs play the central role in detecting failures in our proposed design.

CVMs will cooperate with the operating systems on their local hosts to monitor hardware failures. As discussed in the Service Model of CVMs in section 4.1, CVMs encapsulate the low-level operating system services into explicitly declared connector or service interfaces through which cells can use local hardware and software resources. As a result, CVMs can be aware of some possible failures on local resources, for example, via failed invocations of low-level APIs. In addition, CVMs can take advantage of network operations supported by local operating systems such as **ping** to detect a broken network connection. Upon detection of failures, CVMs propagate the failures to other remote CVMs and notify the local cells interested in the failures.

CVMs are responsible for detecting application level failures (the failures of cells) on their local hosts. In most of existing failure detection schemes such as those in SDS and Jini, service discovery servers and the services (applications) in their domains are not running on the same machines. Consequently, it is impossible to separate the failure of the services from that of network connections. If they detect that a service is not responding (i.e. they do not receive some expected messages from the service), service discovery servers cannot tell whether the non-responsiveness results from the failure of the service or from the broken network connection or both. Separating the two kinds of failures is important for choosing appropriate failure recovery steps. CVMs are in a better position to detect application level failures. A polling and notification model may be a possible solution, in which a CVM constantly polls cells running within and aggregates the failure to others.

4.2.2 Recovery from failures

The resumption of previous active plugged connections after failures or changes is a non-trivial problem. First, the states of failed connection before failure (i.e. the CIDs of other cells that the cell was connected with), the information related to the local resources exported on plugged connections, and the information required for checking data consistency violations due to the sudden interruption all need to be restored. In order to achieve this goal, not only do we need the object serialization-alike scheme as that used in Java but also some transaction semantics in the failure recovery

mechanism as that utilized in MTS model [Micc]. Secondly, if a failure has been fixed, a notification mechanism is needed for triggering the recovery of the corrupted services. For example, if a network link failure leads to the interruption of a plugged connection, the recovery of the linkage should be able to trigger the automatic resumption of the connection. Moreover, an efficient service discovery model is indispensable for initiating the proper service recovery. Many service discovery models have been proposed, such as [Cze99, Sune, MD88, VHH98, RLS98, GPV+98, RSS97]. All those models incorporate dedicated service discovery servers and organize them in either hierarchical manner or loose clusters. In our design, CVMs provide the service discovery functionality. Many problems need solved to make CVMs fully functional service discovery servers such as:

- Design a scheme to enable CVMs cooperate with each other efficiently and make the service discovery mechanism scalable to the internet environment.
- Devise a proper service description scheme for service discovery via CVMs.

Our service discovery model should also enable cells to register their interested services so that they can be properly notified when those services become available, a scenario especially useful when several cells need to cooperate in a chain to provide certain services. Backup service is another possible enhancement to our service discovery model. In particular, if a service becomes unavailable or over-loaded, CVMs may load backup cells to assist fast failure recovery and improve the availability of the service. Issues such as when to initiate backup services and how to transparently transfer to backup services need resolved to make the whole process transparent to clients of the service.

Interestingly, from the discussion above, we find that in order to provide the persistent model, the CVM has to solve service registry and discover problems similar to those in web services. However, these problems are tackled in quite different ways. Instead of having neutral service registries, CVMs are the built-in components fulfilling service registry and discovery. Moreover, the service discovery of CVMs is more dynamic in the sense that once a service becomes available, cells waiting for the service should be notified in a timely manner at runtime. In contrast, there is no default support for automatic service notification in the web service model.

5 Conclusion

Compared to the works in the literature, we take a programming language approach to solve problems that are usually addressed by language-independent protocols such as CORBA and web services. One of the most important advantages of the language approach is simplicity. Programmers only need to understand the language syntax to code complicated distributed applications since most low-level network logics are taken care of by the language runtime. CORBA, in contrast, is a tedious protocol. Programmers need in-depth understanding of the whole CORBA architecture in order to develop working applications. Moreover, although theoretically CORBA is a language and vendor independent protocol, the reality is that interoperability problem still exists among different vendor ORB implementations. Web services succeed for their simplicity. The loose structure of the web service model makes web services a good mechanism for building loosely coupled distributed systems. But, its interoperability is usually limited to a certain service and the clients complying the specification of that service. The pluggable interfaces of cells, on the other hand, naturally fit for tightly coupled applications. The programming language approach also has more expressiveness power than those language independent protocols like CORBA or web services since the later has to use tedious rules/data structures to express computation logics.

Due to the crosscutting feature of security, a programming language with build-in security features is superior to stacked security protocols such as the CORBA SEC built on top of the core CORBA and WS-Security [ADH02] for web services. The cell model introduces flexibility that is not possible in web services and CORBA. For example, in both CORBA and web services, the potential remoteness of an object or a service must be decided at design time. On the other hand, the cell language and its runtime construct an abstraction in which the localness or remoteness of any interactions among cells can be dynamically decided at runtime.

The programming approach also has its own limitations. First of all, programmers are forced to use one programming language unanimously. Moreover, from programmers' point of view, using a language with high-level abstractions would cost them the flexibility of low-level fine-grained controls and the ability of doing potential programmatic optimization.

In this proposal, we present the CVM service model and the strong persistence service model as proposed future studies. Our proposed models could improve network application development with a programming language approach and build software systems that are more robust. We are also interested in exploring other topics such as adding more network com-

munication paradigms like multicast into the connector abstraction. Our initial design is rather on theoretical level. Many important issues need to be carefully investigated. For instance, exception handling is one of the tough problems in the distributed environment, especially when mobile computation is involved. We also need to formalize the language specification. Furthermore, implementation, testing, and comparison are also crucial for proving the practicability of our language model.

References

- [ACN02] J. Aldrich, C. Chambers, D. Notkin, Architectural Reasoning in ArchJava. In *Proceedings of the ECOOP 2002-Object-Oriented Programming: 16th European conference*, pages 334-367, Malaga, Spain, June 2002.
- [ADH02] B. Atkinson, G. Della-Libera, S. Hada and etc. Specification: Web Services Security (WS-Security). Version 1.0 05, April 2002. <http://www-106.ibm.com/developerworks/library/ws-secure/>.
- [AJD+96] M.P. Atkinson, M.J. Jordan, L. Daynes, S. Spence. Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system. In *Proceedings of the 7th Workshop on Persistent Object Systems*, Cape May, NJ, 1996.
- [ASCN03] J. Aldrich, V. Sazawal, C. Chambers, D. Notkin. Language Support for Connector Abstraction. In *ECOOP 2003*, 2003.
- [BF03] T. Blanc, C. Fournet. From Stack Inspection to Access Control: A security Analysis for Libraries (Draft), November 2003. <http://research.microsoft.com/~fournet/tmp/from-stack-inspection-to-access-control-long-draft.pdf>
- [BHL00] G. Back, W. Hsieh, J. Lepreau. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. In *4th Operating Systems Design and Implementation symposium*, 2000.
- [Cze99] S. Czerwinski et al. An Architecture for a Secure Service Discovery Service. In *Mobile Computing and Networking*, pages 24-35, 1999.
- [CCM01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Service Description Language (WSDL) 1.1. <http://www.w3.org/TR/2001/NOTE-wsd1-20010315>, March 2001.
- [DA01] P. Doyle, T. Abdelrahman. Jupiter: A Modular and Extensible JVM. In *Proceedings of the Third Annual Workshop on Java for High-Performance Computing, ACM International Conference on Supercomputing*, 2001.
- [De89] S. Deering. *Host Extensions for IP Multicasting*. IETF, SRI International, Menlo Park, CA, Aug 1989. RFC-1112.

- [FF98] M. Flatt, M. Felleisen. Units: Cool Modules for Hot Languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 259-270, 1998.
- [FG02] C. Fournet, A. Gordon. Stack Inspection: Theory and Variants. In *Proceeding of the 29th ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.
- [FKF98] M. Flatt, S. Krishnamurthi, M. Felleisen. Classes and Mixins. In *Conference Record of POPL 98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, CA, 1998.
- [Gil02] S. Gilmore. Comparing the JVM and .NET. <http://www.dcs.ed.ac.uk/home/stg/MRG/kickoff/slides.pdf>
- [GJS+] J. Gosling, B. Joy, G. Steele, G. Bracha. *The Java Language Specification*. <http://java.sun.com/docs/books/jls/>
- [Gong] L. Gong. Java 2 Platform Security Architecture Version 1.2. <http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html>.
- [Gou01] K J. Gough. Stacking them up: a Comparison of Virtual Machines. In *Proceedings of the 6th Australasian Computer Systems Architecture Conference (AustCSAC'01)*, page 55, Gold Coast, Queensland, Australia, January 29-30, 2001.
- [GPV+98] E. Guttman, C. Perkins, J. Veizades, M. Day. Service Location Protocol, Version 2. IETF, November 1998. RFC 2165.
- [Har] N. Hardly. The Confused Deputy. <http://www.cis.upenn.edu/~KeyKOS/ConfusedDeputy.html>
- [HCC97] C. Hawblitzel, C. Chang, G. Czajkowski. SLK: A Capability System Based on Safe Language Technology. 1997.
- [Hor01] C. Horstmann. *Core Java Volume II: Advanced Features*. Sun Microsystems Press, Upper Saddle River, NJ, 2001.
- [Ibm] IBM. Jikes Research Virtual Machine. <http://www-124.ibm.com/developerworks/oss/jikesrvm/>
- [Lia98] S. Liang. Dynamic Class Loading in the Java Virtual Machine. In *ACM SIGPLAN Conference on OOPSLA '98*, pages 36-44, 1998.

- [Lu02] X. Lu. Report on the cell prototype project (Internal Phd Qualifier Project Report), March 2002.
- [Mica] Microsoft Corporation. The Component Object Model Specification. <http://www.microsoft.com/com/resources/comdocs.asp>
- [Micb] Microsoft Corporation. The Distributed Component Object Model. <http://www.microsoft.com/com/tech/dcom.asp>
- [Micc] Microsoft Corporation. Microsoft Transaction Server. <http://www.microsoft.com/com/tech/MTS.asp>
- [MD88] P.V. Mockapetris, K. Dunlap. Development of the Domain Name System. In *Proceedings of SIGCOMM'88*, August 1988.
- [Mil] M. Miller, E Language <http://www.erights.org/>
- [MFH01] S. McDirmid, M. Flatt, W. Hsieh. Jiazz: New age component for old fashioned java. In *OOPSLA 2001*, 2001.
- [MYS03] M. Miller, K. Yee, J. Shapiro. Capability Myths Demolished. *A submission to USENIX 2003*, <http://zesty.ca/capmyths/>, 2003.
- [Kaf] The Kaffe virtual machine. <http://www.kaffe.org/>.
- [Kre01] H. Kreger. Web Service Conceptual Architecture (WSCA 1.0). <http://www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf>, May 2001.
- [Omga] Object Management Group. The Common Object Request Broker: Architecture and Specification, revision 2.5 edition, September 2001. <http://www.omg.org/technology/documents/vault.htm>
- [Omgb] Object Management Group. CORBA Security Service, version 1.8. http://www.omg.org/technology/documents/formal/omg_security.htm#Security_Service
- [Pj] The PJama Project. <http://www.dcs.gla.ac.uk/pjava>
- [RS02] R. Rinat, S. Smith. Modular Internet Programming with Cells. In *Proceedings of the ECOOP 2002-Object-Oriented Programming: 16th European conference*, pages 256-280, Malaga, Spain, June 2002.

- [RLS98] R. Raman, M. Livny, M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*. July, 1998.
- [RSS97] J. Rosenberg, H. Schulzrinne, B. Suter. Wide area network service location. IETF Internet Draft, November 1997.
- [Son03] D. Sonoski. Java programming dynamics, Part 1: Classes and class loading. <http://www-106.ibm.com/developerworks/java/library/j-dyn0429/>, April 2003
- [SS00] C. Skalka, S. Smith. Static Enforcement of Security with Types. In ICFP'00, Mortreal, Canada, 2000.
- [Suna] Sun Microsystems. *The Java Tutorial: A practical guide for programmers*. <http://java.sun.com/docs/books/tutorial/>
- [Sunb] Sun Microsystems. *The Java Native Interface-Programmer's Guide and Specification*. <http://java.sun.com/docs/books/jni/html/jniTOC.html>
- [Sunc] Sun Microsystems. *The Java Virtual Machine Specification*. <http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>
- [Sund] Sun Microsystems. *The Java Remote Method Invocation*. <http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html>
- [Sune] Sun Microsystems. Jini technology architectural overview. white paper. <http://www.sun.com/jini/whitepapers/architecture.html>.
- [Sunf] Sun Microsystems. JavaBeans Component Architecture Documentation. <http://java.sun.com/products/javabeans/docs>.
- [Smi02] S. Smith. Component Programming Languages (Grant Proposal), 2002.
- [SS02] R. Schantz, D. Schmidt. Middleware for Distributed Systems Evolving the Common Structure for Network-centric Applications. In *John Marciniak and George Telecki, editors, Encyclopedia of Software Engineering*. Wiley&Sons, New York, NY, 2002.
- [Szy98] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

- [Tay] B. Taylor. Java Class Loading: The Basics. <http://www.developer.com/java/other/article.php/2248831>
- [UDD03] Universal Description, Discovery and Integration of Web Services. <http://www.uddi.org/>.
- [US87] D. Ungar, R. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pages 227-241, Orlando, FL, 1987.
- [vAB+96] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure network objects. In *IEEE Symposium on Security and Privacy*, May 1996.
- [VC99] J. Vitek and G. Castagna. Seal: A framework for secure mobile computation. In *Internet Programming Languages*, 1999.
- [VHH98] M. Van Steen, F. Hauck, P. Homburg, A. Tanenbaum. Locating objects in wide-area systems. *IEEE Communications Magazine January 1988*, pages 104-109, 1998.
- [VT97] J. Vitek and C. Tschudin. *Mobile Objects Systems: Towards the Programmable Internet*, volume 1222. Springer-Verlag, Berlin, Germany, 1997.
- [Sin03] J. Singer. JVM versus CLR: A Comparative Study. In PPPJ2003), Kilkenny City, Ireland, June 2003.
- [W3C] W3C. Web Services Activity. <http://www.w3.org/2002/ws/>.
- [WF98] D. Wallach, E. Felten. Understanding Java Stack Inspection. In *Proceedings of Security and Privacy '98*, Oakland, California, May 1998.