The following paper was originally published in the
Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems
Portland, Oregon, June 1997

Metis: A Thin-Client Application Framework

Deborra J. Zukowski, Apratim Purakayastha,
Ajay Mohindra, Murthy Devarakonda
IBM Thomas J. Watson Research Center
Yorktown Hts, NY

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Metis: A Thin-Client Application Framework

Deborra J. Zukowski
Apratim Purakayastha
Ajay Mohindra
Murthy Devarakonda

IBM Thomas J. Watson Research Center,
P. O. Box 704, Yorktown Hts, NY 10598.

Abstract This paper introduces a thin-client programming model and then presents an object-oriented framework for developing applications using the model. The programming model and the framework have evolved from interactions with developers and users of commercial applications. The key aspects of the thin-client programming model are that the client downloads application front ends from the network; that these applications rely only on services found on network servers; that the services are bound as late as possible; and that the applications interact with each other within the confines of a workspace. We implemented the framework using Java Beans and JDK 1.1, and developed several sample applications using the framework.

1 Introduction

Fueled by $Java^{TM}$ and other Internet technologies, new re-engineering efforts are underway to develop commercial applications using a thin-client programming model. In a thin-client programming model, the software client would be substantially *thinner* in that it contains only the graphical user interface (GUI) and a small amount of essential application logic. Most of the application logic runs as *services* on various servers throughout the network. The client software is written using Java so that it can run on any client hardware. The thin-client model is distinct from its hardware counterpart, known in the industry as the Network Computer. However, the thin-client programming model can be the force that makes Network Computers widely deployed.

An application development paradigm becomes popular if appropriate tools are available that enable developers to leverage its benefits easily. While the Java programming language[1], Java Development Kit (JDK) [2], Java component technology [3], and remote access mechanisms [4, 5] enable platform-independent programming, they are only a set of building blocks. Previous work in object-oriented systems suggests that frameworks [6] can be a promising way of achieving widespread use and reuse of software architecture. Therefore, there is a need for a thin-client application framework that is capable of bringing together all parts of an application (the front-ends running on the client and the services available on network servers) and supporting the whole with system services. Lacking such a framework, developers may find it difficult to bootstrap themselves into the new paradigm, and they might resort to an older and less portable methodology such as the Microsoft Windows environment.

Metis, the thin-client application framework presented in this paper, is a related, inter-operable set of objects that enable robust application development in the thin-client paradigm. The goal of Metis is to create a fully server-managed environment for an application, as opposed to the traditional client-server approach. Towards this end, the framework advocates and supports a thin-client programming model where an application consists of application front ends (AFEs) and a collection of backend application-specific services. AFEs rely solely on application-specific services and system services provided by one or more network servers. Thus, AFEs do not depend on local operating system functions. AFEs request services in an abstract manner without specifying the physical location of a service provider. That is, a requested service can be any one of the appropriate service instances available in the network. AFEs

bind, on demand, to these network services. The late binding of services allows server manageability, flexibility, and fault-tolerance.

Metis provides Java classes on the client side for locating and binding to a service instance and for switching to an alternate service instance in case of a failure. In addition, Metis provides a workspace-based client environment suggested by a common commercial application characteristic: interacting sub-applications. The Metis workspace hosts and manages a set of sub-applications; each sub-application is in the form of an AFE. The workspace manager provides visual tools to customize the workspace by adding or deleting AFEs. Workspace configuration information is stored on a server.

In the current implementation, the Metis workspace provides the following object instances for use by the AFEs, and the list may grow as additional objects of common applicability are identified:

- Service location and binding object;
- User authentication object;
- Controller objects for accessing and managing system services such as printing and data storage.

On the server side, the Metis framework depends on support services including an authorization service that ensures controlled access to the system, a code service that maintains a secure repository of trusted AFEs, and a directory service that presents a searchable access to services. These support services must be fault-tolerant and scalable besides using industry standard protocols. Therefore, Metis uses a directory service supporting the Light-weight Directory Access Protocol (LDAP) [7]. Such directory services are likely to become common place and even more robust in the future.

In addition to the above mentioned services, Metis requires printing and data-storage services, and a mechanism for launching and managing application-specific services on various servers. The latter can be accomplished, for example, using the servlets mechanism [2].

The rest of the paper is organized as follows. Section 2 presents the Metis thin-client programming model, sections 3 and 4 describe the Metis framework and implementation respectively. Section 5 discusses the related work, and Section 6 concludes the paper.

2 Thin-Client Programming Model

The key aspects of the Metis thin-client programming model are that the client downloads AFEs from the network; that these AFEs rely only on services found on network servers; that the services are bound as late as possible; and that AFEs interact with each other within the confines of a workspace. AFEs are securely installed and downloaded using a code service. They are also made 'thin' by implementing most of the application logic as one or more services. Late binding to these services provides:

- manageability, because services can be moved across server machines without impacting AFEs;
- flexibility, because services can be selected based on server load; and
- fault-tolerance, because a service can be obtained from an alternate server.

The workspace is a container for AFEs, allowing for interaction, as well as providing a shared environment. One important part of that environment is the authorization information that can be read from a smart card or provided as part of a logon process from an authentication service. The authorization information is used first to determine if a user is allowed to use the system, and then to identify the user's access rights to available AFEs. Afterwards, this information can be used directly by the AFEs to authenticate themselves to the service providers.

Figure 1 outlines the various building blocks of the thin-client programming model. It shows three important parts – the client workspace, application-specific services, and support services needed to provide full thin-client functionality.

2.1 Client Workspace

The client workspace provides a combination of functions in Metis. It provides the AFE container function, some of the conventional desktop functions, and a *virtual environment* of network services. These will be discussed in detail in this section.

Visually, the client workspace has a customizable layout that can be configured on a per-user basis using configuration information stored on a server. When a user logs on, all framework objects are instantiated.

UserProfile: The user profile includes an authorization object that contains user information including name and time of logon. The authorization object is passed with directory and code service re-

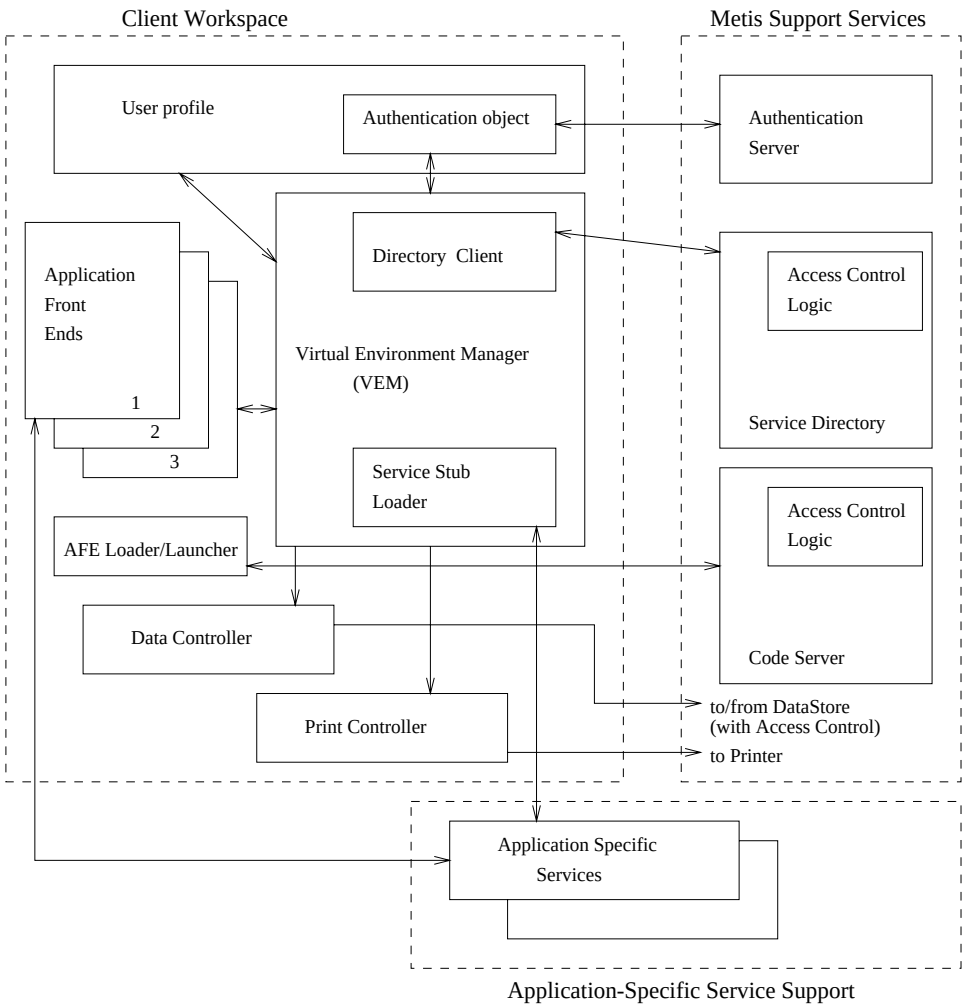

Figure 1: The schematic shows the three important parts of the thin-client application model: the client workspace, support services, and application-specific services. The client workspace contains user profile and authentication objects, objects to find and bind to services, and controllers for data and print. AFEs execute in the context of the client workspace.

quests. These support services recognize the object and restrict the user to only those AFEs and services that allow access by the user. Note that as long as services and AFEs are written to use the authorization object, a single logon procedure is possible.

Application Front Ends: The AFEs are downloaded to the client from a code server either when the workspace is initialized (if they were previously active) or when the user activates one on the workspace. They can be removed from the workspace as needed. Interactions among AFEs, such as data exchange and event notifications, are important especially when the AFEs are implementing sub-applications. Interactions are supported using $JavaBeans^{TM}$ technology.

Virtual Environment Manager: A virtual environment manager (VEM) is a fundamental client object provided by Metis. It is the only entity with which an AFE can request Metis services. Prior to accepting an AFE's request, the VEM checks the AFE's signature to ensure that it is allowed access to the Metis system. As long as the AFE is recognized, the request is forwarded to one of the following VEM clients that act as delegates to Metis support services.

1. **Directory Client:** The VEM has an internal directory client that communicates with the Metis Directory Service upon request of a service from an AFE or the workspace. The directory client and Metis Directory Service together

provide late binding of services. At this time, services can be requested by name and service attributes though clearly, a higher level protocol can be supported. The directory client retrieves service stubs. If the service stub is an object, the directory client instantiates the object and passes a reference back to the caller AFE. If the service stub is a location, the directory client passes the location back to the caller AFE. To reduce the possibility of creating a flurry of messages, called a network storm, that can be caused by a failure of a widely-held service, the directory client retrieves and caches more than one service stub, when multiple providers of the same service are available. Should an active service fail, an alternate stub is fetched, instantiated if necessary, and returned to the AFE.

2. **Service Stub Loader:** The VEM has an internal service stub loader that communicates with the service providers to download any stub code that the service might need for providing the service. The service stub loader is only used if the service stub returned by the directory needs to be instantiated.

AFE Loader/Launcher: The Workspace has an internal object that communicates with the Metis Code Server to download code and launch AFEs. The AFE Loader also checks for digital signatures, uncompresses, and decrypts AFEs as needed. Enabling a client to download code from a centrally administered source makes its use attractive for intranet environments where the software distribution and maintenance on traditional PC clients is expensive. The AFE Launcher runs the AFE when it is selected by the user.

System Services: AFEs need a common set of system services such as printing, storage, and error logging. While application services are private to each AFE, Metis allows sharing of system services. When an AFE requests a system service, an associated controller object is returned. If a stub to the requested service does not exist, the controller creates one by accessing the Metis Directory Service to indicate where the stub class is. The class is downloaded, and instantiated. The controller, in turn, manages the stub instances. For example, in Metis a print controller is a single point of access used by all to 1) create access to specific printers and 2) send information to them. Most of the controllers are provided to support the AFEs. However, the data controller is also used by the Metis workspace to access user configuration.

2.2 Application-Specific Services

The Metis design imposes minimal requirements on service developers. They are free to implement services in any language. Communication between the service and the client-resident service stub can use any protocol, e.g., IIOP [5], RMI [4], or a private protocol. The service providers may provide a client-side stub with a well-known interface for access to its services, or, may only provide a location for AFEs to access services using a mutually understood protocol. The AFEs and service providers may use the authorization object provided by Metis for authentication purposes.

To better integrate services into the network, Metis provides a tool that can be used to register the services with the Metis Directory Service. For AFEs to dynamically access services, they must be registered with the directory. Registering a service makes it immediately available. Removing a service from the Metis Directory Service does not impact AFEs currently using the service. However, when an AFE detects that the service is no longer available, i.e., the service was removed from the server, it can fail-over to another service registered in the Metis Directory Service.

2.3 Metis Support Services

Metis has a number of server components performing distinct functions. While these services are not fundamentally part of the Metis thin-client programming model, they are needed to support that model. For example, the model states that AFEs can bind to any service available on the network that meets its requirements. To have the capability to find all such services, a directory service is used.

Metis Directory Service: The Metis Directory Service accepts queries from the directory client and sends results back to the client. It does not manage the physical service directory. Instead, the Metis Directory Service acts as a client to an LDAP [7] directory server. LDAP is an emerging standard in distributed directory services offering reliability and scalability. Each service in the directory has a unique service location and a number of attribute/value pairs. Each service must at least have a *name* attribute with a non-empty value. Services can be looked up by a name and a search filter composed of a boolean expression of attribute/value pairs. The Metis Directory Service can also perform access control via LDAP with the authorization ob-

ject that the directory client supplies. The Metis Directory Service allows service providers and code-server administrators to add, modify, or delete services in the directory.

The design of the Metis Directory Service simplifies ports to other directory technologies supporting both current and emerging network directory standards. It also can be easily enhanced to provide intelligence to the service selection process. As mentioned earlier, AFEs must currently know the given name of a service and important attributes as well as their correct values. For example, if an AFE wanted to access a color printer service, in the present design it would have to ask for one by name, e.g., ColPrt2. In the future, it might want to ask for a printing service that is physically close, e.g., nearby & color.

Metis Code Server: The Metis Code Service is an AFE repository. It provides central administration of client code and lends itself to be a *tuner* for a third-party code service that uses Marimba [8]. The Metis Code Server can perform access control using the authorization object. It can also digitally sign AFEs to identify them as part of the Metis system. Like the Metis Directory Service, the Metis Code Service is independent of the actual code distribution mechanism.

Metis Authorization Service: The Metis authorization process can either be completely self-contained, e.g., all authorization information is present on the user's smart card, or it can use a standard authorization service such as Kerberos [9]. A reference implementation for the Metis Authorization Service is being developed.

System Servers: Controllers provided by Metis act as the contact points for actual system services available on the network. In particular, a data-store service and printing service must be available on one or (preferably) more servers, and must be registered with the Metis Directory Service. These services receive requests from their associated controllers and return responses.

3 Metis Framework

The thin-client programming model described in the previous section recommends a general software architecture that allows applications to be written flexibly and to run within a manageable and fault-tolerant environment. However, to build these applications from scratch would be very difficult. The Metis framework, shown in Figure 2, presents Java classes and interfaces needed to make programming these applications easier.

The Metis framework provides instances of the key objects discussed in the programming model including the workspace, user profile, AFE loader/launcher, VEM, directory client, service-stub loader, and controller objects. It also manages AFE objects. In addition, interfaces are provided to assist the application developer with the task of writing a thin-client application. These interfaces can be grouped as follows:

- AFE integration into the workspace
- AFE access to services
- Metis System Services interfaces
- Metis Support Services interfaces

The following subsections discuss the interfaces shown in Figure 2.

3.1 AFE Integration into the Workspace

One purpose of the workspace is to contain and launch AFEs. A Metis user indicates which AFEs the workspace contains by using a "ToolBox" that provides the user with a list of all AFEs that he is entitled to use. When the user selects an AFE to add to the workspace, the Toolbox returns the location of the AFE class. The workspace passes the location to the code service client to download the class from the Metis Code Server. The icon associated with the class is accessed and added to the workspace as a button. To launch the AFE, the user double clicks the button. An AFE can be removed from the workspace container by removing the button. If it is active, then the AFE is destroyed. The workspace configuration is automatically saved when the user logs out.

Another purpose of the workspace is to provide resources to all AFEs. These resources include the user profile object and the controllers. To access these resources, the AFE is required to get a reference to the workspace object by implementing AFEInterface. When activating an AFE, the workspace calls the setWorkspace() method on the AFE, passing a reference to the workspace object.

3.2 AFE Access to Services

To access services, AFEs use the interface called the VEMInterface implemented by the workspace object. The workspace object delegates the VEMInterface calls to its member VEM object, that truly implements the VEMInterface.

To bind to a service, the AFE calls the request-

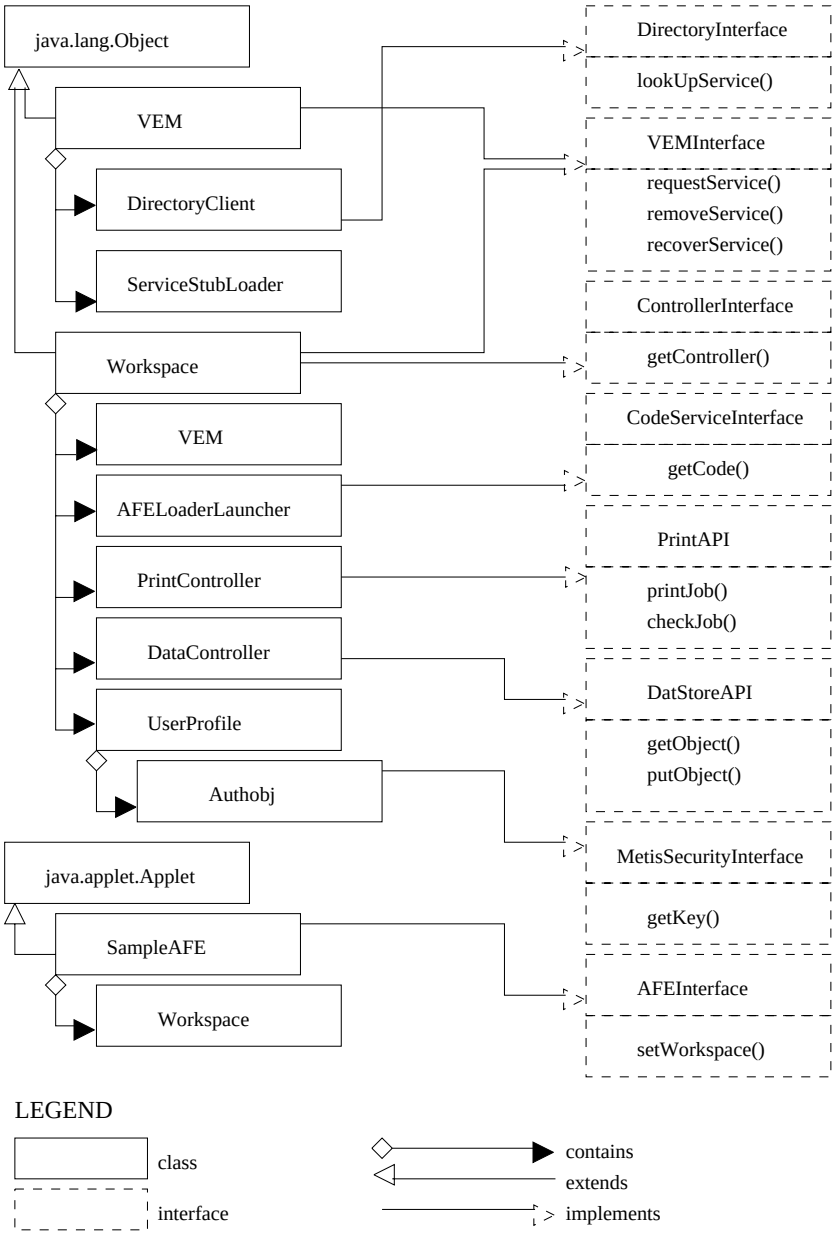

Figure 2: The figure shows important classes in Metis and their relationships. AFEs acquire a handle to the Workspace by implementing the AFEInterface. Workspace implements the VEMInterface and the ControllerInterface, and thus provides the necessary support for AFEs. Workspace uses delegation (to the VEM object) in implementing the VEMInterface. The workspace object contains VEM, controller, and UserProfile objects, and manages AFE instances.

Service() method on its workspace reference. The requestService() method takes the AFE instance, the name of the service, and a preferred list of attribute/value pairs as arguments. A Metis object, called a ServiceInfo, is returned that contains an instance of the service stub or, if desired, the service location. All information needed to access the service is now available to the AFE. Should a service fail while being used, the AFE can re-request a new service by calling the recoverService() method on the workspace, passing a reference to itself and the active ServiceInfo object.

3.3 Metis System Services Interfaces

Metis defines a generic Controller class for managing instances of objects that provide shared system services to all AFEs on the workspace. The Controller class provides a graphical user interface for adding and removing system-service instances from the workspace and for allowing the user to select a specific instance to provide the service. Service-specific controllers extend the Controller class and implement service interfaces to provide the desired service. The services are explicitly configured using a GUI provided by the Controller. A user can also setup a specific instance of the service provider as a default for that service.

To access the system services, an AFE calls the getController() method. For example, should the AFE need access to the data store, it calls the getController() method, passing it the type of system service required, e.g., 'DATASTORE'. The VEM returns the instance of the appropriate controller for that service. The AFE then uses the instance when it needs system services. The Controller in turn, delegates the request to the default service provider, or allows the user to select a service provider from the list of configured providers.

Metis also provides reference APIs for system services such as printing and data storage. The printer API allow an AFE to print, check the status or remove a print job. The data storage API allow an AFE to store and retrieve files from a data store.

3.4 Metis Support Services Interfaces

The Metis Support Services interfaces are not visible to the application programmer. They are hidden within the VEM, controllers, and the user profile. At this time, three key interfaces have been identified. The DirectoryInterface has been designed and implemented. The CodeServiceInterface and the MetisSecurityInterface are being developed.

The directory client uses the DirectoryInterface to avoid being tied to a single directory protocol. Any physical directory service can transparently plug into Metis simply by enhancing the Metis Directory Server with updated implementations of the DirectoryInterface. The interface has been kept small, containing only one method, to minimize the complexity of directory access. When the VEM client is requested to access the Metis Directory Server (e.g., an AFE called the requestService() method), the VEM calls the lookUpService() method on the directory client. The authorization object, name, and attributes passed to the VEM are passed as arguments to the lookUpService() call.

4 Metis Implementation

This section gives the highlights of the implementation for both the Metis framework and the AFE suite used to demonstrate it. The discussion of the Metis framework implementation will be restricted to the workspace implementation and the service access interfaces, i.e., the VEMInterface and the DirectoryInterface implementations, along with important utility objects that are used. The discussion of the AFE implementations will center on what specifically was done to integrate the applications into the thin-client programming model. Overall, the Metis framework is about 7000 lines of Java code.

4.1 Metis Workspace

The reference Metis implementation supports Java bean behavior, to allow a natural mechanism for AFEs to interact within the workspace. The Sun Microsystems reference implementation of the BeanBox is the basis of the Metis workspace. The BeanBox provides bean containment, and support for property sheets, visual manipulation of beans, and state storage.[1]

At workspace startup, a logon process verifies the user and creates the user profile object if the verification is successful. This process includes showing a dialog box for user name and password information and checking it against an authorized user list. The user is also allowed to use a "smart card"[2] instead of providing information directly to the logon dialog. If the user passes verification, the VEM and controller

[1] Unfortunately, some of these functions (e.g., event interactions, property sheets, and pickling) had to be disabled to get the base functions working since the version of JDK1.1 (Beta 3) was unstable. These functions will be reintegrated as JDK1.1 gets more mature.

[2] A floppy drive was used as a place holder to smart card hardware.

objects are instantiated.

In Metis, the workspace is populated with buttons, each showing the icon of the AFE. Each button contains the URL of an applet and an appletviewer instance with a reference to the applet. The appletviewer has been modified from the JDK implementation to separate the applet instantiation from its execution. When a button is selected, the appletviewer initiates the applet execution. The initial layout on the workspace is read from the user's configuration file, accessed from a data store.

The ToolBox has been completely rewritten from the BeanBox implementation. In our implementation, the ToolBox queries the directory server to get a list of AFEs that the user can access. It allows a user to add an AFE as a button to the workspace, and when the button is added the AFE loader downloads the AFE code from the Metis Code Server.

4.2 VEMInterface Implementation

An AFE requests a service by calling the requestService() method on its workspace reference. The AFE passes a reference to itself, the well-known name of the service, and a special Filter object. Passing the requester of the service enables the VEM to keep state on each AFE. It also allows application designers to, in effect, build a service providing object that gets services on behalf of all AFEs in the suite. The Filter object was designed to simplify the attribute/value logic specification. At this time, only the "AND" function is provided, e.g., a service must have "attr1=val1" & "attr2=val2". The Filter class will be enhanced in the future to allow arbitrary logic specifications. An example code segment showing service request is:

```
Filter filter = new Filter();
filter.addElement("height","low");
filter.addElement("speed","slow");
workspace.requestService(this,
    "my_service", filter);
```

The requestService() implementation in the VEM class does the following list of actions. Note that not all exception paths are included in the list.

1. If a null filter was passed in, the VEM creates one. A special attribute/value pair ('type'/'Service') is added. The type attribute was added because the Metis Directory Service contains both services and AFEs.
2. The VEM checks its internal state to see if the AFE is a known service user.
3. If the AFE is not a known service user, its authorization is checked to ensure that the AFE has the credentials needed to use the Metis system. If it passes the check, it is added to the VEM internal state. Otherwise, the requestService() throws an exception.
4. The VEM checks its internal state to see if the AFE has previously asked for the same service. If so, it returns the associated ServiceInfo object.
5. The lookUpService() method is called on the directory client and the returned list is wrapped in a ServiceInfo object.
6. The VEM checks the validity of the ServiceInfo object. The object will be invalid if the directory search could not find any services, e.g., if the service was unknown or if the user did not have access to the those registered. An exception is thrown if the ServiceInfo is invalid. A timestamp is also added to the new ServiceInfo object. The ServiceInfo object is then added to the VEM internal state and returned to the AFE.

The object class, ServiceInfo, returned by the requestService() method maintains the information returned by the directory lookup. For the reference implementation, the locations of the services are stored within the ServiceInfo object as URLs. The class provides accessor methods for AFEs to use.

The AFE uses the ServiceInfo object to get to the service. During its use, the service could fail. The AFE, when it detects such an occurrence, can request a replacement service by calling the recoverService() method on its workspace, passing it a reference to itself, and the ServiceInfo it used to get the previous service. The workspace delegates the request to its VEM object. The VEM class's recoverService() implementation does the following list of actions. Again, not all exception paths are included.

1. Add the currently accessed service to the blacklisted services maintained in the ServiceInfo object. This blacklist is simply a means of tracking which of the services returned by the directory lookup have been used, and failed.
2. See if there is another service known in the ServiceInfo that has not yet been used. If so, setup

that service, make it the current service in the ServiceInfo object, and return back the same ServiceInfo object.

3. Otherwise, save the timestamp and the current blacklist. The timestamp is important in later steps. The blacklist is needed because more services may be available than those returned during the previous directory lookup(s). That is, the blacklist transcends the partitioning done by the lookup; It is used to capture the AFEs access to ALL relevant services.

4. Request the directory client to do a new lookup, passing it the blacklist, and wrapping the return list in a new ServiceInfo object.

5. If the ServiceInfo object is valid, then there were other services as yet unused by the AFE that met its requirements. Copy back the timestamp and the blacklist. Return the ServiceInfo object to the AFE.

6. If the ServiceInfo is invalid one of two scenarios could have happened. First, the current sweep through all possible services may have finished. If so, a new sweep is started by clearing the blacklist and creating a new timestamp. Second, there may be no more services available, even though they may still be registered with the directory service. One way for this scenario to happen is if there is a network partition that does not affect access to the Metis Directory Service but that interferes with access to the servers that the services are running on. It is detected by noting that the time required to finish a sweep is less than a pre-configured time determined by the system administrator.

7. If no new services are available, VEM repeats steps 4 through 6 one more time. If, even after the retry, it cannot satisfy the request then it displays an error message.

4.3 ControllerInterface Implementation

Controllers that implement the ControllerInterface provide AFEs with access to system services. In the current version, two types of controller classes are available: PrintController and DataController. The PrintController provides access to printers while the DataController provides access to data stores. Note that there is only one instance of the Controller for each system service. The interface consists of the method below.

```
public Object getController(int type) {
  switch (type) {
    case ControllerInterface.PRINTER:
      return printController;
    case ControllerInterface.DATASTORE:
      return DataController;
    default:
      return null;
  }
}
```

4.4 DirectoryInterface Implementation

The DirectoryInterface need only define the following method:

```
public SearchResult
  lookUpService(Authobj authobj,
       String name, String filter,
       URL[] blacklist, int howmany,
       String attrlist)
  throws java.rmi.RemoteException;
```

The requestService() method of the VEMInterface calls this method to access the LDAP directory. The parameters passed to this method are:

authobj: The authentication object from the workspace used for access control in the LDAP directory.

name: The name of the service being looked for.

filter: A filter composed of attribute/value pairs.

blacklist: A list of service locations that the client specifically does not want returned even if matched.

howmany: A count of matches to be returned.

attrlist: A list of attributes that are to be returned with every match.

The method returns an object containing service locations and attributes. The implementation of the lookUpService() method on the Metis Directory Server is straightforward. The Metis Directory Server acts an LDAP client. It constructs an LDAP query, and submits the query to an LDAP server. If desired, a random subset is chosen from the services returned by the LDAP server and are returned to the client. At the present time, the Metis Directory Server uses an LDAP client API. The use of the Java

Naming and Directory Interface (JNDI) [2] is being investigated.

The interface is simple but flexible enough to be used for various purposes. In addition to looking for service names and services with specific attributes, it can also return a list of all services (subject to access control) when used as:

```
SearchResult s=lookUpService(authobj,
    null, null, null,
    ALL_POSSIBLE, null);
```

where the null values indicate no filtering and subsetting is to be performed; or, can return the count of all services with the name "LotusNotes" when used as:

```
int i=lookUpService(authobj,
    "LotusNotes", null, null,
    ALL_POSSIBLE, null).getCount();
```

4.5 AFE Implementations

A number of applications were implemented to demonstrate the usefulness of the framework, including an application (called ViewGlass) that can access Lotus Notes servers and a financial application suite. In this section, the AFE/service partitions are described as well as AFE use of the VEMInterface to access its service for the ViewGlass application.

4.5.1 ViewGlass Implementation Notes

ViewGlass provides a user with the ability to access her Notes mailbox (to send, read, and delete messages), and to read discussion databases. The functional split for this application is that the AFE, written in Java, would contain the GUI, rich text browsing support, and a private communication layer that directly accesses a proxy service. The service, written in 'C', would accept messages from the GUI, call the appropriate NotesAPI functions, and return information back to the GUI. The service runs as a daemon and contains client-specific state.

During the ViewGlass initialization process, the requestService() method is called to get the location of a known Lotus Notes service. The location of the service is then accessed and a socket connection is made. This is shown in the code below.

```
  try {
    sinfo = workspace.requestService(
    (Object)this,"LotusNotes",null);
  } catch (Exception e) {
    System.out.println(
    "LotusNotes service not found");
    destroy();
    return;
  }
  URL url = sinfo.getURL();
  server_name = url.getHost();
  server_port = url.getPort();
```

Connection failures are notified to the AFE via the exception mechanism. The AFE calls the reset() method to recover from failure.

```
public void reset() {
  try {
    sinfo = workspace.recoverService(
        (Object)this,sinfo);
  } catch (Exception e) {
    System.out.println(
    "LotusNotes service not recovered");
    proxy_recv = null;
    destroy();
    return;
  }
  URL url = sinfo.getURL();
  String server_name = url.getHost();
  int server_port = url.getPort();
  // ...  make the connection
  wk_sp_frame.recover();
}
```

Note that the reset() method ends with a call to the recover() method. Since some client state is maintained in the service, full recovery is not possible from just the client. However, the state completely contained in the client is recovered.

5 Related Work

During the last year, many players in the computer industry have focused attention on alternatives to the traditional client. One of the design points for most of the efforts is to ensure that the users continue to have access to all resources that they are accustomed to. One well-investigated system is to provide access to applications using a browser. In this system, the application runs mostly on well-known servers. HTML pages, some enhanced with small Java applets, are sent back to the client. The browser relies on an underlying operating system to get access to files and printers. The browser metaphor works well if there are only a few applications that do not interact much and if each application has a restricted amount of user interaction.

The browser system is one possible choice for enabling the use of Network Computers as the hardware client for thin-client applications. However, for environments where an application is composed of many sub-applications that may interchange data frequently, the browser metaphor is not sufficient. The browser metaphor also lacks the integration of network-based access to system resources. Other alternatives like defining Network Computer desktops or webtops (i.e., the Lotus DeskTop and HotJava Views) have been investigated as extensions to the browser metaphor. Also, many vendors are vying to provide environments for Network Computers to access system resources commonly found on traditional desktops, like printing and file access [10].

During the Metis effort, we emphasized enabling commercial applications for Network Computers. Many of these business applications are currently single-system based. Moving to a network (1) introduces unreliability not often found with standalone systems, (2) raises security concerns, and (3) distributes resources like printers and files. While the browsers and Network Computer desktops could handle the latter two issues in future implementations, they are not meant to address the first. Both the Lotus DeskTop and HotJava Views could be integrated with Metis by replacing the workspace used in the reference implementation, to address all three issues today. Metis would then also provide a framework for developers to implement robust applications for browsers and Network Computer desktops.

The previous paragraphs focused on browsers and desktops that provide access to complex applications for Network Computers. However, Metis provides a distributed application technology as well as a user system. There are several distributed application technologies for traditional clients and servers that some developers could use for Network Computers. These technologies include CORBA and design patterns.

CORBA [11, 12, 13], is a distributed application technology specified by OMG that emphasizes reusable services and facilities. These specifications allow applications to interact with other application modules independent of the machine architecture and language the modules have been written in. OMG's Trading service specification allows an application to query and identify service names that match a particular criteria. These service names are then bound to a particular object as per the Naming service specification. The combination of the two services can be used by applications under CORBA to achieve late binding of a name to an object. Metis provides similar late binding of service providers to an AFE using an LDAP directory server. AFEs can specify service properties through the Filter class. Results of the match are available to the AFEs for binding and use.

Design patterns [14, 15] have been proposed as a technique for application development that also emphasize reuse of software architectures, including those for distributed systems. Design patterns allow software developers to write their applications using high-level models that are independent of language and machine architecture. The patterns focus on key components and their interaction to facilitate reuse of software. Design patterns have been used for writing large scale commercial applications. The Metis workspace has been written as a design pattern for desktops on thin clients. The workspace provides components for locating and binding to services, access to system services, and security components. The intent is to allow application developers to use the workspace pattern in developing application suites for thin clients.

6 Conclusions

In this paper we presented a thin-client programming model where clients download application front ends that have a presentation layer and some application logic, but the bulk of an application is executed as services on remote servers. We described the design and implementation of a framework, called Metis, that enables the thin-client programming model, and showed how it can be used in sample applications.

The design of the Metis framework has an open architecture composed of abstract interfaces for various services so that any implementation can be plugged in. We implemented both client- and server-side infrastructure and realized a full end-to-end framework that provides support for:

- finding and binding services
- access control and authentication
- system services
- code services

Metis support for late binding makes it possible to write reliable, flexible, and manageable thin-client applications. Moreover, the Metis framework provides true platform independence beyond the language level by virtualizing all system resources as services. A demonstration of the reference Metis implementation that includes the Metis classes and

API documentation is available at the IBM alphaWorks web site (http://www.alphaworks.ibm.com) as TCAF.

During the course of this work we identified several areas of further research which may be beneficial to the thin-client programming model. These include support for application-specific recovery, remote event mechanisms,and improved security and communications. We plan to explore the above areas as we enhance Metis to fully realize the benefits of the thin-client programming model.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[2] Javasoft. *Javasoft Home Page*. URL = http://www. javasoft.com/, 1997.

[3] Javasoft. *JavaBeans Component APIs for Java*. URL = http://splash.javasoft.com/beans, 1997.

[4] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. In *Proceedings of the USENIX Conference on Object-oriented technologies*, pages 219–231, 1996.

[5] OMG. *CORBA/IIOP*. URL = http://www.omg.org/corba/corbaiiop.htm, 1997.

[6] R. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(5):22–35, June/July 1988.

[7] Timothy Howes and Mark Smith. A Scalable Deployable Directory Service for the Internet. In *Proceedings of INET 95*, 1995.

[8] Marimba, Inc. *Marimba*. URL = http:// www.marimba.com/, 1997.

[9] B. Clifford Neuman and Theodore Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[10] Novera EPIC Release 1.0. Novera Epic Developer's Guide. Technical Report EPICDEV-100-1, Novera Software, Inc, November 1996.

[11] OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, 1995.

[12] OMG. *CORBAservices: Common Object Services Specification*. OMG, 1995.

[13] OMG. *CORBAfacilities: Common Facilities Architecture*. OMG, 1995.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[15] Douglas C. Schmidt. Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *Communications of the ACM*, 38(10):65–74, October 1995.