The following paper was originally published in the
Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)
Santa Fe, New Mexico, April 27-30, 1998

Programming Network Components Using NetPebbles: An Early Report

Ajay Mohindra, Apratim Purakayastha,
Deborra Zukowski, and Murthy Devarakonda
IBM T.J. Watson Research Center

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/

Programming Network Components Using NetPebbles: An Early Report

Ajay Mohindra, Apratim Purakayastha, Deborra Zukowski, Murthy Devarakonda
IBM T. J. Watson Research Center
Hawthorne, NY
{*ajay, apu, deborra, mdev*}*@watson.ibm.com*

Abstract

A network-centric application developer faces a number of challenges, including distributed program design, efficient remote object access, software reuse, and program deployment issues. This level of complexity hinders the developer's ability to focus on the application logic. NetPebbles removes this complexity from the developer through a network-component based scripting environment where remote object access and program deployment are transparent to the developer. In NetPebbles, a developer selects needed network components from a distributed catalog, and then writes a script invoking its methods as if the components are local. When the script is launched, the runtime determines the component sites in the network and transparently moves the script as needed. Using three simple examples with different data flow patterns, we show that the NetPebbles approach is superior to the traditional client/server systems and mobile agent technologies because a scripting language is easy to use, it requires less code, and the distributed systems complexity is hidden from the programmer. This paper is an early report on the NetPebbles project, describing the motivation, design, prototype implementation, and the experiments using the NetPebbles approach..

1 Introduction

NetPebbles is a scripting environment for writing and executing distributed component-based applications. The prominent among its features is the use of transparent program mobility both for accessing remote components and for mobile agent-like behavior. The resulting benefit to the programmer is a great deal of simplicity.

A NetPebbles programmer writes a script by first selecting needed *interfaces* from a distributed catalog and then invoking interface methods as if the *components* implementing the interfaces are local. An end user launches such a script into the network, where the NetPebbles runtime dynamically determines the component sites and transparently moves the state of the script to component sites as necessary. Implied in this model is the possibility that a component might be downloaded to the script execution site, and even that both the script and the component might execute at a third site. In a typical scenario, after visiting several component sites, the script terminates by returning to the launching site. However, a NetPebbles script can be programmed to remain forever in a waiting mode listening to asynchronous notifications.

NetPebbles programs are written in a Basic-like typeless scripting language. Interfaces are the same as the interface definitions of the Java language. However, each interface must be registered in a distributed catalog with a globally unique identifier. One or more components may implement each interface. Components also have globally unique identifiers and are also registered in a distributed catalog. Components are medium-to-large grain Java Beans with certain restrictions that make them highly self-contained. For example, in the current implementation, the NetPebbles runtime handles all exceptions thrown by a component and components may not send or receive events. The current NetPebbles implementation consists of an interpreter, a runtime environment, and a library to access a distributed catalog, all written in Java. This allows us to run the NetPebbles environment on all machines supporting the Java environment.

We selected three different applications, represent-

ing three different data flow patterns, and implemented them using NetPebbles, Java/RMI, and Aglets. The data flows have the broadcast, two-party interactive, and multi-party synchronization patterns. We used the same components in all three implementations. NetPebbles implementations are significantly smaller than the implementations using Java/RMI and Aglets, and the programming complexity is correspondingly less. For two out of the three applications, the number of network messages are also the fewest with NetPebbles. Within the scope of these experiments, we find it encouraging that NetPebbles is able to deliver on the promise of simplicity without additional network messages.

The NetPebbles project is relatively young at this point. Work is in progress to address many challenges of this approach to distributed component programming. Issues related to using typed components in a typeless language, recovery and transactional issues, and transitive trust requirements are some of the design challenges that are being addressed. We are using system administration for a large network of PCs, workstations and laptops as a real life scenario to validate our approach to simplicity, reuse of existing software, and mobile agent-like behavior. A visual builder is being developed to provide an interactive interface browsing and script development tool.

This paper describes the motivation and the programming model of NetPebbles, practical experiments in understanding the advantages of the NetPebbles model, the current implementation, an ongoing case-study using NetPebbles, and future work.

2 Motivation and An Example

The NetPebbles design is motivated by three goals: (1) Enable large-scale software reuse in developing distributed applications; (2) Simplify distributed programming for rapid development and deployment; (3) Provide mobile agent-like behavior without the extra cost of programming complexity.

The enormous interest and early successes of the Java Beans and ActiveX component technologies indicate that future software development will increasingly center around creating and reusing software components. We envision that this paradigm will extend well beyond today's user interface development. There will be components that provide access to enterprise data, platform-specific legacy software, system information, expert analysis tools, centralized information collection points, and even administrative APIs of printers and so on. Not only will there be a multitude of components in a network, there will also be many components providing the same function. We may assume that the functionality is captured through Java-style interface definitions, and that an interface may be implemented by one or more components. It is also likely that there will be a distributed catalog providing descriptions of such interfaces and components. The NetPebbles goal is to leverage such a scenario.

Ousterhout [1] suggests that application development using scripting languages (such as Tcl and Perl) is inherently simpler than using systems programming languages (such as C and Java). Part of the simplicity is derived from the fact that the scripting languages make it easy to invoke powerful pre-existing programs and compose several such components to provide more complex logic. In the NetPebbles design, we take this idea one step further by treating network components as if they were local. In most cases, programmers have to worry little about where components are instantiated. Method invocation on components is uniformly simple. By hiding the complexity of accessing remote components, the NetPebbles goal is to simplify distributed programming and thus allow for rapid application development and deployment.

Mobile agent-like behavior can be valuable in scenarios where a user-specific task needs to execute asynchronously in a network of servers. For example, a mobile agent can be launched from a palmtop device using wireless communication and results can be retrieved later perhaps at a different physical location. In another example, a mobile agent can be installed at a server in the network to wait for an unusual event and then respond to it in a certain programmed manner. The NetPebbles design allows for such programs to be written with little or no additional complexity over single-system programs. The key function provided in NetPebbles to support mobile agents is the transparent program mobility, which allows a programmer to specify where a component should be instantiated (and hence where its invocation should occur) either abstractly (through a set of attributes) or concretely.

An Example

To provide a concrete introduction to the features and benefits of NetPebbles, we discuss here a simple example and describe how it executes. Ms. Sys Admin is a system administrator for a group of about a hundred PC users. The group uses Lotus Notes as the primary tool for communication. She is interested in tracking what versions of the Notes client is running on every PC to make sure they don't diverge too much. She attempted to do this by periodic e-mail. She quickly realized that some people did not reply to e-mail, some people did but were annoyed, some people provided wrong information, and some did not even know how to find the information. Ms. Sys Admin could use a small NetPebbles script that visits every workstation, downloads her own NotesSniffer component that simply records the Notes Client version, and returns to her workstation with the collected information. Figure 1 shows the corresponding script for the application.

The application uses three components that implement the *IDomainAdmin*, *IMachineAdmin* and *IReportGeneration* interfaces, respectively. Methods for each interface are shown in Figure 1. The IDomainAdmin interface manages a group of machines in a domain. It allows one to create new domains, add and remove machines from a domain, and get a list of all machines in a domain. The IMachineAdmin interface is used to administer software on a machine. The interface allows one to add and remove software, get a list of all installed software, get the version for a software, and notify the users by writing to the console. The IReportGeneration interface is used to generate a report in a tabular form.

Variables are initialized in lines 1–3. In line 4, the **createComponent** function is called. The function's arguments are an interface that a component must implement, and a filter for component attributes. It searches the catalog of advertised components and locates a component that supports the IDomainName interface and manages the domain named "dept231". If the component properties allow the component to be downloaded then the runtime may download the component. Otherwise, the runtime moves the script to the site where the component is located. In either case, the createComponent() call returns an instance of the component. After the instance has been created, the script invokes the getAllMachines() method of the component to get a list containing all machines (IP addresses) in the domain for dept231. Note that the syntax of method invocation requires the complete signature of the method. This is because the language is typeless. For object-oriented languages like Java that allow method overloading, without a complete signature, it is impossible to determine which method to invoke. Line 8 declares another array to store the version numbers. The **length** function returns the number of elements in the data referred to by the variable.

Lines 10–18 contain a for loop that iterates through the set of machines. In line 11, the **createComponent** function creates an instance of a component that supports the IMachineAdmin interface. Note that the **at** keyword in the statement indicates to the runtime that the component should be created at a specific site. If the runtime is unable to create a component at that site then it is noted and script continues to execute. Line 16 defines a group and adds the instance to that group. A group is a language object that supports an unordered collection of identical objects. Operations such as **add**, and **remove** are supported on a group. A group allows the same method to be invoked on all the objects of the group using a single statement. In line 17, the getVersion() method returns the version number for "Lotus Notes". In line 19, the consoleWrite() method is called on all members of the group to notify client users about the activity. In lines 22–24, a report generation component is created at the system administrator's home machine to display the results.

3 The Component Model

This section describes what we mean by components and how we use components in NetPebbles. The NetPebbles component model is based on well-known concepts of interfaces, components, attributes, and globally unique identifiers. To encourage reuse of existing component software, we borrowed ideas freely from the Java Beans and ActiveX technologies. At a high level, NetPebbles components are self-contained Java Beans. However, like the ActiveX components, certain aspects of components are immutable and all aspects of components have globally unique identifiers.

In NetPebbles, an **interface** is a group of semantically related methods or functions, while a **component** is an implementation of the interface. A component can implement one or more interfaces. Each

Component Interfaces

```
IDomainAdmin                              IMachineAdmin                     IReportGeneration
  - void addDomain(String)                  - void addProgram(String)       - void generateReport(String[])
  - void addMachine(String, String)         - void removeProgram(String)
  - void removeMachine(String, String)      - String[] getAllProgram()
  - String[] getAllMachines()               - String[] getVersion(String)
                                            - void consoleWrite(String)
```

NetPebbles Script

```
1)  filter="domainName=dept231"                            //get admin component for dept231
2)
3)  interface = "IDomainAdmin"                             //component supporting this interface
4)  comp = createComponent(interface, filter)              //create instance
5)  list = comp.getAllMachines()[]                         // get list of machines
6)
7)  interface = "IMachineAdmin"
8)  dim version[length(list)]                              // declares an array to store version numbers
9)
10) for (i=0; i < length(list); i++)
11)   mcomp = createComponent(interface, null) at list[i]  //create a instance at a specific machine
12)   if (mcomp == null)
13)     version[i] = "Unable to retrieve version from" + list[i] // Note failure and continue
14)     continue
15)   endif
16)   add mcomp to group G                                 // define a collection and add mcomp to it
17)   version[i] = mcomp.getVersion(String)["Lotus Notes"] // get version number for Lotus Notes
18) endfor
19) G.consoleWrite(String)["System Admin NetPebble requested Lotus Notes version number"]
20)                                                        // write to console of all machines indicating that
21)                                                        // the netpebble visited
22) interface = "IReportGeneration"                        //component to display results
23) comp = createComponent(interface, filter) at home
24) comp.generateReport(versions)          // generate report at Sys Admin m/c
25) exit
```

Figure 1: A NetPebbles script and associated interfaces used for the system administration example.

interface and component is identified by a **globally unique identifier** (GUID)[1]. An interface's GUID is called an *InterfaceID*, while a component's GUID is called a *ComponentID*. Interfaces and components may have **attributes** associated with them. Attributes provide either informative description (such as its semantics and suggested use) or parameters to assist the NetPebbles runtime (such as a preferred execution site and usage cost). Attributes are also identified by GUIDs, called *AttributeIDs*.

NetPebbles InterfaceIDs are immutable, i.e. once an interface has been defined, both the syntax and semantics associated with the interface cannot change. The InterfaceIDs are required to be registered in a component catalog. The NetPebbles interface browser allows an application developer to browse through these interfaces in the catalog. The NetPebbles runtime uses the catalog to locate and bind to components that implement the interfaces. NetPebbles ComponentIDs and AttributeIDs are also immutable. Immutability of ComponentIDs ensures that newer versions of a component do not cause runtime exceptions because they no longer support published InterfaceIDs.[2] Immutability of AttributeIDs guarantees that semantics associated with an attribute will not change from the point a NetPebbles program is built to when it executes. Components provide necessary read-only accessor methods to examine the ComponentIDs and AttributeIDs as-

[1] A globally unique identifier is generated using a combination of machine hardware address, current time and a 32-bit random number.

[2] With mutable ComponentIDs, such a scenario is possible if there is a delay in updating the catalog whenever a component is changed.

sociated with a component.

In the Java Beans model, package names are used as a means to identify interfaces and components, however, this mechanism does not assure immutability of interfaces. The Beans model does not have a registry or a way to look up method signatures or properties without using introspection. NetPebbles addresses these shortcomings by using mechanisms from the ActiveX model.

Even though, ActiveX interfaces are immutable and a local component registry exists, the ActiveX programming model is component-centered. Typically an ActiveX programmer is expected to find a component (in the registry), instantiate it, and then "discover" interfaces supported by the component. NetPebbles view is that interfaces are the starting point for a developer. A developer specifies the needed interface and lets the NetPebbles runtime find a "suitable" component that implements the interface.

The NetPebbles interface browsing requirements have two related technologies: The CORBA trader service and the Light-weight Directory Access Protocol (LDAP). In the CORBA trader service [2], interfaces can be hierarchically organized and can have attributes that capture non-computational aspects. The hierarchical relationship can be used to capture both interface inheritance as well as attribute aggregation. However, to locate a component, a programmer still has to know the interfaces and attributes that the component supports. An integrated (perhaps, a visual) browser/search-engine that understands the relationships among interfaces, components, and attributes is needed for NetPebbles. We are investigating suitable enhancements to the CORBA trader service, and prototyping it using the LDAP [3] distributed directory technology.

3.1 Interfaces and Components in The NetPebbles Catalog

The NetPebbles catalog is a collection of all published interfaces and components, and contains the information described below.

Interfaces – Each entry contains the following information:

- **Interface identifier (InterfaceID)** – Identifier assigned to the interface.
- **Method signatures** – List of method signatures supported by the interface. A method signature is of form *<resultType> methodName(<argType1>, <argType2>, ..., <argTypeN>)*.
  For methods that do not return any results, the resultType field is set to **void**.
- **Interface attributes** – Set of AttributeID/value pairs associated with the interface. This information is used by a programmer at build time to find suitable interfaces.
- **Component identifier (ComponentID)** – List of component identifiers published in the catalog that implement the interface.

Components – Each entry contains the following information:

- **Component identifier (ComponentID)** – Identifier assigned to the component.
- **Location** – Location of the component. This field is of type URL.
- **Class name** – Class that should be instantiated to use the component.
- **Supported interfaces** – List of interface identifiers implemented by this component.
- **Access control list** – List of principals that are authorized to use this component. More details on the security and access control model can be found in section 5.2.2.
- **Recognized guarantors** – Set of guarantors that the component vendors would honor. NetPebbles uses a certificate based scheme for enforcing security and access control (see section 5.2.2).
- **Server affinity** – This field indicates the preference of the execution site where the component should run. By default, the NetPebbles script moves to the site where the component is located. However, if the component needs to use graphics for interacting with the user then the runtime needs to download the component to the client machine where the user is located. The NetPebbles runtime currently

supports only two values for this field – affinity value set to 1 indicates that the script should migrate to the component server, and affinity set to 0 indicates that the component should be downloaded to the client machine.

- **Component attributes** – Set of AttributeID/value pairs associated with the component. For example, attributes could contain the name of the vendor, cost for using the component, and access control requirements. We are currently working on a more detailed attribute specification model for NetPebbles.

Tables 1 and 2 show two interfaces, IDomainAdmin and IMachineAdmin, along with two components that implement these interfaces. Note that for sake of readability and convenience, we use symbolic names instead of very long numbers as globally unique identifiers in the Tables and in the rest of the paper.

At build-time, the developer browses and selects from the set of InterfaceIDs that are available in the component catalog. The argument to the createComponent statement contains the InterfaceID that the developer selects. The developer can also specify a set of AttributeID/value pairs that the runtime should use when resolving the InterfaceID to a particular ComponentID. For example, a developer might want to use components that were developed by a particular vendor. At runtime, NetPebbles contacts the component catalog and resolves the InterfaceID into a specific ComponentID.

4 The Experiment

This section describes three simple examples we wrote using NetPebbles, Java RMI, and Aglets (the Java-based agent technology from the IBM Tokyo Research Laboratory), and what we have learned from this exercise. The examples have been chosen to represent three common data flow patterns, i.e. broadcast, two-party interactive, and broadcast interactive. We used the same set of Java Beans in programming with the three technologies.

4.1 Lunch time notification – broadcast

In this example, a user informs a set of people that it is time to go for lunch. The data flow is broadcast (see Figure 2a). The example uses two interfaces (ILunchGroup and ILunchGUI) and thus two components implementing the interfaces. The implementation strategies are described below:

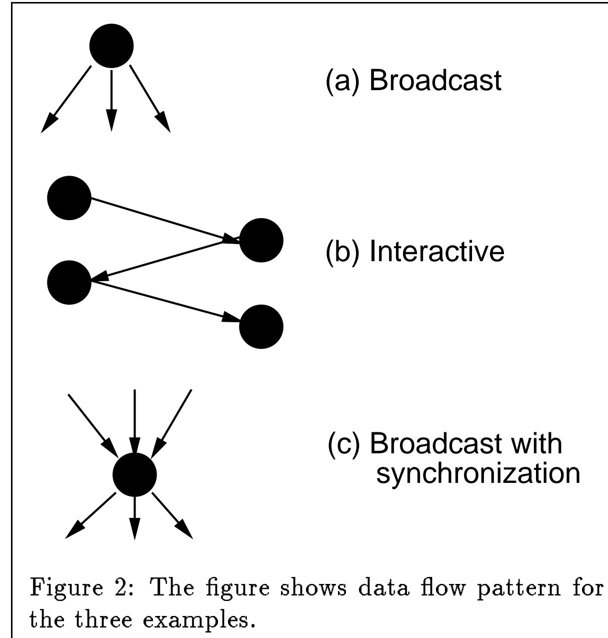

Figure 2: The figure shows data flow pattern for the three examples.

NetPebbles: In this approach, the script first instantiates a component that implements the ILunchGroup interface, and then by invoking a method on the component it retrieves a list of user and workstation names for a lunch group. The component is probably invoked on a database server machine where information about administrative or informal group lists are maintained. Next, for all users in the group, the script migrates to the user's workstation, creates a component implementing the ILunchGUI interface and displays the "Lunch Time" message.

Java/RMI: For the RMI version, it is required that all machines in the lunch group run "Impl" servers corresponding to the component implementing the ILunchGUI interface. These servers export interfaces that can be called from a client. For this implementation, the client (the machine of the user who set out to inform the group about lunch) iterates through the group, invoking the display method remotely on the "Impl" servers running at the user's workstations.

Aglets: The application logic is the same as for Java/RMI, except that the programmer writes

Table 1: This table shows two examples interface entries.

Interface		
Interface identifier	IDomainAdmin	IMachineAdmin
Method signatures	void addDomain(String) void addMachine(String,String) void removeMachine(String,String) String[] getAllMachines()	void addProgram(String) void removeProgram(String) String[] getAllPrograms() String [] getVersion(String) void consoleWrite(String)
Interface attributes — description	This interface allows one to create new domains, add and remove machines from a domain and get a list of all machines in a domain	This interface allows one to add and remove software, get a list of all installed software, get the version for a software, and notify a user by writing to the console
Component identifier	Comp1	Comp1, Comp2

Table 2: This table shows two examples component entries.

Components		
Component identifier	Comp1	Comp2
Location	http://machine1.xyz.com	http://machine2.abc.com
Class name	COM.xyz.Admin.class	COM.abc.Machine.class
Supported interfaces	IDomainAdmin IMachineAdmin	IMachineAdmin
Access control	world execute	world execute
Recognized guarantors	Verisign	Citibank
Server affinity	0	0
Component attributes — vendor — cost	 XYZ Inc 0	 ABC Inc. $100 per use

code to handle migration explicitly. At each visited workstation, the agent creates an instance of the component implementing the ILunchGUI interface and displays the message.

4.2 TicTacToe – Two party interactive

The data flow pattern is two-party interactive (see Figure 2b). The TicTacToe application exhibits this pattern. After the first player makes his move, it is the second player's turn. The two players alternate until some player wins or the game ends in a draw. The application uses a TicTacToe GUI component to display the state of the gameboard. Note that the application requires that the gameboard be continuously displayed while the game is being played.

NetPebbles: The NetPebbles script creates component instances at the players machines during the first exchange of moves and these components persist as a part of normal NetPebbles execution. They display the game board. The NetPebbles runtime transfers the game state between the machines while the game is being played as a part of moving the script program state.

RMI: TicTacToe "Impl" servers are started on the players machines and the client controls the game by invoking appropriate methods on the servers.

Aglets: Since Aglets do not support persistence of component instances, persistence is simulated using slave agents. The application uses a master agent to control the game and uses messages between the two slave agents to transfer the game state.

4.3 Multi-party meeting scheduler – Broadcast with synchronization

In this application, a user tries to schedule a meeting time with a group of participants. The application is written such that it first gathers previously scheduled appointments from the calendars of all participants, finds a conflict-free time, and lastly updates individual calendars to add the new meeting. The data flow pattern for this application is shown in Figure 2c. The individual calendars are stored on a set of servers (perhaps, the individuals belong to different administrative domains). The application uses a GUI component to interact with the user for initiating a meeting request, a Planner component to access individual calendars, and a Scheduler component to schedule a meeting time.

NetPebbles: The NetPebbles script instantiates a GUI component at the requester's machine for reading details about the meeting request (meeting date, duration, subject, and list of participants). The script then migrates to the location where individual participant's calendars are stored. The Planner component is used to access scheduled appointments for the requested meeting date. After collecting information about all the appointments, the script uses the Scheduler component to determine the meeting time. After the meeting time has been finalized, the script visits each of the Planner components to update the participants calendars. The script completes execution by displaying a confirmation message at the requester's machine.

RMI: In the RMI version, "Impl" servers are started for the Planner, Scheduler and GUI components. The client implements the application logic by sending messages to these "Impl" servers.

Aglets: Although an Aglet version is not implemented at this time, the application logic is similar to NetPebbles. An agent performs similar tasks as the NetPebbles script.

4.4 Discussion

Table 3 shows the lines of code comparison and Table 4 shows the message count for the three technologies. Consistently, the number of lines of code for NetPebbles is small compared to the others. The NetPebbles program sizes varies from 19% to 42% of Java RMI and Aglets implementations. Even though the lines of code is not an ideal measure of complexity by any stretch of imagination, we use it in the absence of any other reasonable metric we could find. Our experience in coding these examples – a thoroughly subjective measure but certainly a better measure than the number of lines – also suggests a very similar trend.

The small code size and simplicity of NetPebbles implementations is a result of the scripting approach and the fact that the runtime hides the details of remote component access and script mobility. First, the scripting language does not require the developer to worry about importing packages, declaring types of the variables, or handling exceptions. We believe that this allows the developer to focus primarily on thinking about the application logic. Second, as much as possible, the complexities of the distributed system should be handled by the system. This is the approach we take in NetPebbles.

Using an agent technology, such as Aglets, requires the developer to be aware of the mobility aspects of the application. The program has to be aware of the location where it is executing so that it can do the right thing. In NetPebbles, mobility is transparent to the program – the script transparently moves to the location where components are available. From the programmers perspective, all component invocations appear as local calls. Additionally, Aglets do not support persistence of component instances naturally. It has to be simulated by use of slave agents. This further complicates programming for now the developer has to write code to handle messages exchanged between agents.

The Java/RMI approach requires that the programmer be aware of importing correct packages, types, and handling exceptions. The programmer has to ensure that the appropriate application binds to the correct stubs for accessing remote services, and application-specific "Impl" servers are running on all the machines that the application needs to get service from. This adds to the deployment complexity of the RMI based applications. Under NetPebbles, the runtime creates component instances on demand, i.e. when the script needs to use them.

With respect to the message count, the NetPebbles approach typically exhibits fewer number of messages when compared to either Java/RMI and Aglets. This is because the script moves to the

Table 3: This table compares the number of lines of code to implement the three examples.

Applications	Approaches		
	NetPebbles	Aglets	RMI
Lunch Notification	10	48	54
TicTacToe	34	105	82
Meeting Scheduler	45	–	191

Table 4: This table shows the message count for the three examples.

Applications	Approaches		
	NetPebbles	Aglets	RMI
Lunch Notification $\mathcal{N}$ is the number of participants	$3\mathcal{N}+1$	$\mathcal{N}$	$4\mathcal{N}$
TicTacToe $\mathcal{M}$ is the number of rounds. $5 \leq \mathcal{M} \leq 8$	$\mathcal{M}+8$	$6\mathcal{M}+8$	$4\mathcal{M}+8$
Meeting Scheduler $\mathcal{O}$ is the number of participants	$6\mathcal{O}+5$	–	$7\mathcal{O}+11$

site where the component is located – subsequent invocations on the component do not require sending new messages. Applications that rely on agent mobility generate fewer messages under Aglets (as in the Lunch application). However, when persistence of components is needed Aglet-based solutions generate messages that are comparable to the Java/RMI approach. The Java/RMI based approach would typically generate the maximum number of messages as each method invocation on a component requires a new message on the network. Note that in the current implementation of NetPebbles, during component creation a message is always sent to the component catalog to resolve component names and locations. These messages can be easily avoided by using intelligent caching strategies.

Clearly, the trend seen here may not extend to large applications. Furthermore, we do not imply that RMI (and all the classic client/server techniques) and Aglets are irrelevant or inherently inferior. These approaches are valid when an application developer wants to use the full power of a systems programming language, and therefore he or she is willing to invest in the necessary development costs. But when the developer can satisfy the application requirements largely by reusing components, the NetPebbles approach simplifies program development and deployment.

5 Implementation

This section describes two aspects of the current NetPebbles implementation. First, it describes the support that exists for creating a NetPebbles program, and next it describes the support that exists for executing a NetPebbles program.

5.1 Support for creating applications

5.1.1 Scripting Language

The NetPebbles research focus is on system design, but in order to freely experiment with the system-level issues such as efficient program mobility and so on, we needed our own scripting language. Our long term goal is to integrate NetPebbles programming language features into an existing scripting language. At present, NetPebbles provides a typeless, object-based scripting language called NPScript. The language borrows its syntax from the BASIC programming language, and it supports basic control flow constructs found in other scripting languages. In addition, it allows one to instantiate components and perform method invocations on those components. The component instances are persistent through the life of a NetPebbles script. The language does not provide RMI/RPC-like semantics or provide constructs to define new objects.

For creating components, NPScript provides the **createComponent** keyword. The syntax is as follows:

```
<varName>=createComponent(<interfaceName>,
     <filter>) [at <locationName>]
```

where *varName* is the handle to the resulting instance, *interfaceName* is the InterfaceID that the component should support, and *filter* is a stringified boolean expression of component attribute/value pairs. Optionally, using the **at** keyword, one can also specify the *locationName* indicating where the component should be instantiated. Under the covers, the runtime contacts the component catalog and resolves the *interfaceName* and *filter* to a ComponentID. If a *locationName* is specified then the runtime migrates the script to that location and also downloads the component to that location. Otherwise, the runtime either migrates the script to the component host, or downloads the component to the current location (further details are discussed in section 5.2.1). The runtime then creates an instance of the component and stores a handle to it in *varName*. The syntax for performing method invocations is as follows:

```
[<resultVar>=]
   <varName>.methodName(type1,...,typeN)
   <arg1,..., argN>
```

where *resultVar* is the variable to store the results of the invocation, *varName* is the handle to the component instance, *methodName(argType1,..., argTypeN)* is the complete signature of the method to invoke, and *<arg1,..., argN>* is the list of arguments.

The language is inherently typeless – a programmer does not have to indicate the type of the variable before using it. The typeless nature of the language simplifies programmability. However, NetPebbles uses typed components written in Java, which require that data be appropriately converted to the correct type before the method is called. This requirement makes the runtime more complex because it has to convert the data values to the correct types before invoking methods on the components. Another artifact of using typed components is that two methods in the same interface can have the same name but different argument types. This makes the method resolution ambiguous at runtime if only the method name is specified in the scripting language. Thus, NetPebbles requires the programmer to completely specify a method with a name and types of arguments. The combination of a method name and its argument types uniquely identifies a method in an interface. It also provides information to the NetPebbles runtime to appropriately convert data to the types that the components expect.

As an effort to simplify programming, we are considering support for collections at the language level. The basic idea is to allow the programmer to create a collection and populate it with homogeneous components. The programmer can then perform any operation on the collection in a single statement without having to iterate through individual elements of the collection. Support for language constructs that allow the NetPebbles script to fork itself, or perform asynchronous method invocations are also being considered.

5.1.2 Visual Builder

The NetPebbles environment includes a visual builder that allows users to build entire scripts. Current component-based visual builders fall into two categories, those that provide assistance with GUI building and those that allow arbitrary connection of component events and properties. Examples of GUI builders include Visual Basic [4] and Visual Oblique [5]. Examples of generalized component interconnectors include IBM Visual Age for Java^{TM} [6] and the many Java Bean tools now available.

The NetPebbles Visual Builder design addresses the following two questions: (1) Why should one use a visual building environment when scripts are so simple to write? and (2) Given that a visual builder is useful, what should a program look like?

Thus far, some insight to both questions has been gained. First, a visual building environment is useful for NetPebbles as it can provide a programmer with drag-and-drop construction, syntax hiding, NetPebbles-specific error/notification handling, and migration hints. Second, visual programming languages, while at first attractive, provide a representation that is quickly intractable. However, the emphasis on dataflow in these languages does help with data navigation. Therefore, the program representation should be textual, but should exploit an underlying dataflow structure. For simplicity, the NetPebbles visual builder allows each variable to be assigned in only one place. (Note that this is a broadening of a strict "write-once" concept introduced in elementary dataflow to allow for assignments in loops.) Also, to better assist savvy programmers, the organization of the text should be

capable of illustrating the program's implicit migration and some notion of the size of the state must be visually available. The investigation also has resulted in a sequence of proof-of-concept demos, that help to assimilate gained insight into the builder architecture.

5.2 Support for executing applications

The basic runtime support in NetPebbles includes resolving InterfaceIDs to ComponentIDs, providing support for script migration, and component instantiation and execution. A few design goals for the NetPebbles runtime are: heavy client-side infrastructure to initially launch NetPebbles should not be mandated, server-side infrastructure (NetPebbles runtime) should not need constant administration, deployment of special NetPebbles specific protocols should not be required, and the whole infrastructure should be portable. The NetPebbles component catalog is implemented using an enhanced CORBA trader service. All of the client and server side infrastructure is written in Java. The NetPebbles runtime also supports integrated features described below.

5.2.1 Launch and Execution

In the current implementation, the runtime is deployed as a Java servlet under a web server at all machines that host NetPebbles. The NetPebbles runtime consists of a script execution engine and necessary data structures to keep keep track of active NetPebbles. The runtime uses HTTP for communication. A user initiates execution of a NetPebbles script using a Java program called *launchpad.* Lauchpad takes in as arguments the name of the file containing the NetPebbles script and the name of the *home* server machine where the script should start executing. Launchpad first checks the script for any syntax errors. If there are no syntax errors then it constructs a NetPebbles message containing the script and sends it to the web server using the HTTP POST command.

After receiving the NetPebbles message, the NetPebbles runtime initializes some data structures and starts executing the script. The script and its associated program state (program counter, stack and data heap) is transferred to other NetPebbles runtimes as needed. When the script completes execution, it returns to the home server, which then sends out cleanup messages to all runtimes visited by the script. These messages are needed to cleanup any persistent component instances and data structures created by the script.

A NetPebbles script need not migrate from one host to another. Instead, it might download required components. Optionally, if a location is specified by the programmer using the **at** keyword, the runtime migrates the script to the specified location and also downloads the component to that location. For example in the Lunch application, to find members of the group the script can either migrate to the database server where the component is located, or the component can be downloaded to the current script location. However because of the **at** keyword, to display the lunch message the script migrates to and the component is downloaded to the specified user's machine. In the absence of the **at** keyword, the runtime makes the decision based on the server affinity attribute of a component published in the catalog. The actual classname to instantiate the component is also obtained from the catalog. All component instances accessed by a script are managed by a private instance manager. When asked to create an instance of a component, the instance manager searches through its classpath to locate the appropriate class to instantiate. Instead of returning the actual instance to the script, the instance manager generates and returns a unique handle (consisting of the server IP address and a GUID). The instance manager uses this handle to invoke any methods on the component instance. The handle is also used by the runtime to determine if it needs to migrate to another site before making a method invocation. We consider script mobility as a valuable feature to support in the NetPebbles environment. With mobility a NetPebbles script becomes a mobile agent. It preserves network bandwidth by data forwarding, naturally supports disconnected operations, allows for very thin clients, and prevents component piracy. The NetPebbles environment supports script mobility and accordingly deals with its implications on issues such as security, monitoring, and state migration.

Due to the use of servlets, all NetPebbles scripts run on the same underlying Java virtual machine. However, each script runs in a separate context with its own execution thread, state and instance manager. This separation prevents scripts from inadvertently accessing/manipulating component instances created by other scripts. We are currently working on defining a more detailed security model for

NetPebbles.

5.2.2 Security and Access Control

In light of mobility, security is an important part of the NetPebbles environment. NetPebbles uses a certificate based scheme for performing security and access control. Trust among NetPebbles scripts, hosts, and components is established by authentication. Each NetPebbles script is associated with a principal that identifies a person or role. Each host is also associated with a principal. Each component is associated with a global package name and a manufacturer. Authentication is performed using public key certificates, and an authenticated NetPebbles script is transitively trusted by a chain of NetPebbles servers that have authenticated one another (delegation). A NetPebbles script and components are transported using the SSL protocol that ensures privacy and integrity. Components allow access by individual principals or by members of a group of principals. Access control information is advertised in the catalog for NetPebbles to determine access rights prior to migrating to a host. We have implemented a prototype of our security infrastructure using Java Security APIs.

The NetPebbles runtime uses the certificate associated with the script for performing access control checks at runtime. An application is allowed to bind to components based on the access control fields specified in the component entry. The NetPebbles runtime also makes available the principal of the script to any component instance the script creates to facilitate component specific access control checks done by the component.

6 System Management - A Case Study

To gain experience in applying NetPebbles to large applications, we are prototyping solutions for systems management. In this section, we describe this ongoing case study.

The Problem Statement

Systems management in an enterprise environment is a challenging problem. System administrators have to routinely perform tasks to keep the application software and system files up-to-date, and take back up of client machines. The task is further complicated with the presence of portable computers as the machines could be disconnected from the network when a maintenance task is scheduled.

Current Approaches

The most popular approach to systems management is the one based on the SNMP protocol. In this approach, each client machine runs an SNMP agent process that executes management procedures as directed by a central control station. The client machine is assumed to be connected to the network at all times. Studies have shown that SNMP based solutions are inflexible, less scalable, and difficult to manage for large environments [7, 8].

To address the scalability limitation of SNMP, researchers have proposed an alternate solution called Management by Delegation (MbD) [7]. In MbD, the central control station dispatches delegation agents to the client machines using a remote delegation protocol. The agents, which are a collection of procedures, are executed at the client machines by a delegation process. The results of the execution can either be pulled by or pushed to the control station. Although, the MbD based approach is more scalable than SNMP, it is still inflexible and difficult to manage as the procedures themselves are hard to modify and maintain.

At IBM Research's Watson site, system administrators have the responsibility of managing over 4000 machines consisting of IBM RiscSystem 6000s running the AIX operating system, and IBM PCs and laptops (running either Windows 95 or OS/2 operating system). The administrators currently use a central server solution for systems management (similar to SNMP). A maintenance daemon that implements necessary system administration specific commands executes on client machines. System administrators add commands to a well known file that is stored on a shared network file system. The daemon periodically monitors this file, reads commands and performs the necessary tasks on the client machine. The main challenges faced by the administrators are to handle heterogeneity of underlying systems, disconnected machines, and propagating changes to the maintenance daemon.

The NetPebbles approach

The NetPebbles programming model is a natural fit to the systems management area. At present, system administrators routinely manage single systems using scripts written in Tcl, Perl, or Ksh. The NetPebbles paradigm extends the notion of scripting to the network. In a simple scenario, a system administrator would write a script to perform some management tasks on a set of machines. The tasks themselves are performed by components that are available on the network. The script specifies the interface supported by a component and optionally, the location of the component instance. It also specifies what methods to call on the component instances. The script visits each machine in the list, downloads (by runtime) relevant components from a central component repository and completes the task of management. The use of a scripting language simplifies programming. The ability to download components from a central code repository simplifies manageability of the components (equivalent to the management procedures under SNMP). The ability to move from one machine to another improves the scalability of the system.

In a more complex scenario, one can envision that the NetPebbles script could be programmed to perform application and machine specific tasks. To handle disconnection, one can envision a scenario where the script and related components are downloaded on the machine before the machine is disconnected. The script continues to perform system management tasks offline. When the machine reconnects later either via a dial-in or a LAN connection, the script can report back to the system administrator with the results. We are currently prototyping some of these ideas at IBM Research.

7 Related Work

Mobile agent projects such as Telescript [9], Aglets [10], Itinerant agents [11], Tacoma [12], Agent Tcl [13], the Mole project [14], The Frankfurt project [15], etc. have investigated issues related to migrating code from machine to machine. These projects cover issues related to transport, state migration, language, security, and navigation. It is beyond the scope of this paper to discuss each of them in detail. Instead, we mention the noteworthy aspects of some projects that have influenced the design of NetPebbles.

The Frankfurt project uses HTTP as a transport protocol and observes that using a generic infrastructure for transport is critical for successful deployment of an agent infrastructure. We take a similar approach in NetPebbles. The Agent Tcl project has perhaps the most comprehensive treatment of security and language issues. They use a public key security infrastructure based on PGP, and have access control mechanisms. Like Telescript they also stress the usefulness of a limited vocabulary scripting language for security. They have influenced the NetPebbles security model. The Aglets and the Mole project use Java for infrastructure to ensure that their infrastructure is generally portable.

Unlike most of the these systems, NetPebbles uses a scripting language leveraging its inherent simplicity. Unlike Agent Tcl (which is a scripting language), NetPebbles supports transparent mobility thus the programmer is not burdened with the task of programming mobility explicitly. We should, however, point out that NetPebbles does not support component-instance mobility and hence the concept of mobility is somewhat simpler in NetPebbles than in some of the other mobile-agent systems.

NetPebbles has similarities to workflow systems in that a NetPebbles script can be thought of as executing workflow "items" on different machines. However, the classical workflow systems such as IBM's Flowmark [16] and HP's OpenPM [17] are closed systems controlled by a single workflow server. Thus, they are fundamentally different from the NetPebbles model. New "WebFlow" systems [18] are no different than classical workflow systems, except that they have a browser front-end. The COSM project [19] suggests mechanisms that can make programs written in a workflow specification language behave like a mobile agent to carry out workflow functions. Unlike NetPebbles, the focus of the COSM project was to enable workflow using CORBA services.

Work in distributed programming environments (such as Java/RMI, COM/DCOM, DCE, and CORBA) is also pertinent to NetPebbles. Throughout the paper we discussed how NetPebbles relates to these technologies. Unlike these systems, NetPebbles hides the complexities of remote component access from the programmer. It uses transparent program mobility as a single mechanism to address the requirements of both remote component access as well as mobile agent-like behavior.

8 Summary and Future Work

In this paper, we described NetPebbles, a script-based programming environment for composing and executing distributed component-based applications. NetPebbles uses program mobility to support both remote component access as well as mobile agent-like behavior. Furthermore, since NetPebbles programmers use components as if they were local and write programs in a scripting language, they find that there is a great deal of simplicity in using NetPebbles. Typically, a NetPebbles programmer selects needed interfaces from a component catalog and writes a script invoking methods on components implementing the interfaces. The components are highly self-contained Java Beans with globally unique identifiers.

We implemented three examples with different information flow patterns using three different technologies: NetPebbles, Java/RMI, and Aglets, and analyzed the implementation complexity and the number of messages used. Results show that NetPebbles indeed offers a significant amount of simplicity, as seen in the number of lines of code as well as in the programming effort needed. In addition, NetPebbles also requires the smallest number of network messages in most cases. However, to understand how NetPebbles extends to more realistic scenarios, we are developing NetPebbles scripts to manage a large network of PCs, workstations, and laptops.

Since NetPebbles is a relatively young project, we recognize many technical challenges that need to be addressed. The following is an incomplete list of issues we are working on.

1. Recovery: As a NetPebbles script travels through the network, there are many opportunities for failures, such as hardware, software, and communication failures. There must be mechanisms in NetPebbles runtime or in the infrastructure it uses to handle such events. Recoverable messaging mechanisms such as the IBM MQ Series are being investigated as a way to address this need.

2. Transactional Support: Should there be transactional mechanisms in the NetPebbles language? Transactional mechanisms can help multi-server updates such as a travel reservation involving hotel, airline, and car, but it can also increase the programming complexity.

3. Exceptions, Events, and Data Pipes: What are the ways in which components should be allowed to interact with each other and with the NetPebbles script? Should we allow a component to throw an exception that can be handled by the script writer? What about Java-style events? Is there a need to connect components to form a pipeline for data to flow?

4. Business Model: How will script users pay for component use and resource consumption of other machines? The Tacoma project [12] considers an *ecash* approach where a mobile agent starts out with a certain amount of ecash and spends it along its way. For intranets and extranets, such a scheme may be an overkill.

5. Transitive Trust Issues: As NetPebbles travels through the network, it becomes necessary for each server to verify that the preceding servers in the path have not introduced security violations (either maliciously or otherwise) into the program state.

6. The Scripting Language Issues: Using typed components in a typeless language raises several composability issues. In addition, introducing a new language is always a difficult marketing challenge. We would like to be able to provide the necessary NetPebbles mechanism in an existing scripting language while being able to implement program mobility in an efficient manner.

In spite of many remaining challenges, the NetPebbles programming environment has been successful in showing that the complexities of remote component access and mobile agent-like behavior can be hidden from the programmer. It also shows that scripting languages can be used for distributed component programming while retaining the simplicity inherent in such languages. This work suggests how the Java Beans style component model can be used to encourage large-scale reuse of network resources rather than just GUI programs.

Acknowledgement: The authors gratefully acknowledge the feedback provided by the anonymous reviewers of the COOTS '98 program committee and the Adventurous Systems and Software Research projects review committee in IBM Research. We especially thank our colleague Sandra J. Baylor for her input on the presentation of this final version, she nearly re-wrote the abstract for us.

References

[1] John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *Computer*, pages 23–30, March 1998.

[2] OMG. CORBA services: Common Object Services Specification. URL = http://www.omg.org.

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

[4] Steven Holzner. *Advanced Visual Basic 4.0 Programming*. M & T Books, 1996.

[5] Krishna Bharat and Luca Cardelli. Migratory applications. In *Proceedings of the ACM Symposium on User Interfaces Software and Technology*, November 1995.

[6] IBM. *IBM's Visual Age for Java*. URL = http://www.ptsdirect.co.uk/static/ibmvaj.html, 1997.

[7] German Goldszmidt and Yechiam Yemini. Distributed management by delegation. In *Proceedings of 15th International Conference on Distributed Computing Systems*, 1995.

[8] James Herman. *The Sorry State of Enterprise Mgmt.* URL = http: //www.psgroup.com/ features/1996/f396d.htm, 1996.

[9] General Magic. *Telescript Technology: The Foundation for the Electronic Marketplace.* URL = http://www.genmagic.com/Telescript/ Whitepapers/wp1/whitepaper 1.html, 1996.

[10] Danny B. Lange and Daniel T. Chang. IBM Aglets Workbench: Programming Mobile Agents in Java. URL = http://www.trl.ibm.co.jp/aglets/whitepaper.htm, 1996.

[11] David Chess, Benjamin Grosof, Colin Harrison, David Levine, and Colin Paris. Itinerant agents for mobile computing. Technical Report RC20010, IBM T.J. Watson Research Center, October 1995.

[12] Fred B. Schneider Dag Johansen, Robbert van Renesse. Operating System Support for Mobile Agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, May 1995.

[13] Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management*, December 1995.

[14] Fritz Hohl Markus Straber, Joachim Baumann. Mole - A Java Based Mobile Agent System. In *Proceedings of the ECOOP 96 Workshop on Mobile Object Systems*, 1996.

[15] Anselm Lingnau and Oswald Drobnik. An Infrastructure for Mobile Agents: Requirements and Architecture. In *Proceedings of the 13th DIS Workshop*, September 1995.

[16] Frank Leymann and Dieter Roller. *Business Process Management with Flowmark.* URL = http://www.software.ibm.com/ad/flowmark, 1995.

[17] Ming-Chien Shan Jim Davis, Weimin Du. OpenPM: an Enterprise Process Management System. *Bulletin of the Techincal Commitee on Data Engineering*, 18(1):27–32, March 1995.

[18] Viewstar Corporation. *Internet Workflow.* URL = http://www.viewstar.com, 1997.

[19] M.Merz, B.Liberman, K.Muller-Jones, and W.Lamersdorf. Interorganizational Workflow Management with Mobile Agents in COSM. In *Proceedings of the First International Conference on Practical Applications of Intelligent Agents and Multi-Agents*, 1996.