Multithreaded Java/CORBA Agents for Remote Server Access

by Louis J. Sacco and Bruce R. Maxim

Until Notes and Domino R5, a server-based scheduled agent could not access databases that reside on other Domino servers. Now, with R5 support for CORBA, Notes developers can create Java agents that do just that. CORBA stands for Common Object Request Broker Architecture, a standard that allows a Java applet or application to create a persistent session with a remote Domino R5 server. In previous articles in THE VIEW, you learned the basics of creating standalone CORBA-enabled Java applications and applets that connect to a Domino server from a remote machine,¹ and you saw how to leverage CORBA support in a Domino Web application.² In this article, you’ll learn how to create CORBA-enabled Java agents.

Why might you want to use CORBA-enabled Java agents? As LotusScript/Java programmers, we have found that such agents make it possible to create more efficient applications in networked Domino environments. For example, we have often found it necessary to run essentially the same code on many servers when developing administrative tools that deal with infrastructure changes across an enterprise. What CORBA offers in situations like these is the ability to have centralized agent code, running on a single Domino server, that can access other Domino servers in the enterprise. In fact, one of us recently designed an application to create mail file databases on many different Domino servers. For each new mail database required in the enterprise, a CORBA-enabled Java agent running on a single server


creates a mail database from the mail template on the appropriate remote Domino server, and then updates a centralized log file located on another server. This architecture certainly beats going to each server to create a scheduled “database creation” agent, or replicating a database application with the “database creation” agent across all servers to achieve the same effect.

Another benefit of using a Java/CORBA agent for tasks like these is that you can use Java for concurrent processing. Because the Java language provides several facilities that promote multithreading, you can optimize your agent by processing multiple tasks (such as the creation of databases from templates) concurrently. This is especially useful on machines that have multiple processors.

In this article, we’ll examine the benefits of multithreaded CORBA-enabled Java agents in more detail and demonstrate how to harness the power of multithreaded CORBA-enabled Java agents for your own applications. To this end, we will explain what it takes to get your R5 Domino server set up to host Java/CORBA applications, how to create a simple Java/CORBA agent to test your implementation, and how to build an application that creates mail file databases on remote Domino servers by means of a scheduled Java agent running on a single server.

If your programming strengths are in LotusScript, not Java, be assured that once you get past the Java programming conventions, such as some of the Java keywords, curly braces, and semicolons, it’s fairly easy to put together an application that can access any Domino server in your enterprise (because the Domino object model is essentially the same for Java as for LotusScript). This is not meant to oversimplify the Java programming language but rather to stress that much of the code you will be writing will have a direct correlation to the LotusScript classes we’ve learned to love so much. We recommend keeping the Java/CORBA R5 chart from Lotus close at hand to use as a crutch until the Java language becomes more ingrained.

---

### How CORBA Enables Remote Access

Before Domino and Notes R5, a developer who wanted to use server-based agents to access remote servers was constrained by the Domino agent security model. In order for a pre-R5 agent to run, the signer of the agent must be authenticated with the server and have rights to run scheduled agents to get access to resources, and so sessions must be opened under the privileges of that ID. But a server makes its connection to a remote server over TCP/IP via Notes RPC (remote procedure calls), and so the session is opened under the privileges of the server’s rights, not of the person who signed the agent. Thus, access to the remote server under the auspices of the user is not possible because there is no basis for authentication of the user’s credentials.

Authentication via CORBA is quite a bit different. Instead of Notes RPC, CORBA uses IIOP (Internet Inter-ORB Protocol) to transport data over TCP/IP. Don’t let these terms intimidate you. Just think of CORBA as another method of communication. It allows you to pass your HTTP password over a TCP/IP network in order to gain access to another server, which is just how you would do it if you were logging onto the server from the Web. Similarly, you can create an agent that accesses multiple servers and that your application users can run in the foreground, because they have authenticated with their ID and are able to instantiate sessions across servers.³

³ Of course, this solution comes with a significant performance cost: the application user’s client is tied up while running the agent, and the user’s PC workstation is usually far less powerful than a server machine.
The sample database and agents featured in this article are available for free download from THE VIEW’s Web site at http://www.eView.com. You can fully customize the sample application by means of variables in a profile document.

Benefits of Using CORBA/Java Agents

We’ve just said that CORBA-enabled agents can help developers create more efficient applications in networked Domino environments, and we gave as our example the application that can create databases on other servers. To understand exactly what we mean by “more efficient,” let’s look at a before-and-after picture.

The Traditional (Pre-CORBA) Approach: Distributed, Replicated Notes Databases

To implement an application that can create databases on other servers — mail file databases, for example — without using CORBA, a developer typically would use mail-in databases on several servers, with each server containing a local agent to create new mail databases.

A centralized application mails documents to each remote server. Each remote server runs its scheduled agent to query a view of the new documents sent to the mail-in database for processing in a FIFO (first in/first out) fashion. The agent then processes the documents and moves them to a completed successful/failure view.

The mail-in databases are replica copies of each other, so that the agent code can be shared. To identify where the agent should run, the developer can use the “*” notation (introduced in R4.6) or the “-AnyServer” notation (introduced in R5). See Figure 1.

To keep things straight across all the servers, the developer would typically use Reader/Author names fields and/or selective replication to prevent documents meant for one remote server from replicating to another remote server (each server must receive only those

![Figure 1 - Settings for Distributed, Replicated Notes Databases](image-url)
documents that are specific to it). Figure 2 is a diagram of what such an environment might look like.

The Central Processing database takes in some input — from users, external feeds, or some other source — which determines the processing that must take place on remote servers 1 through “n” (i.e., any number of servers). In this example, the processing consists of creating new mail file databases on those servers. The Central Processing database needs to delegate this processing to the mail-in databases on the remote servers because, without CORBA, a scheduled agent cannot access or create databases on other servers.

Notice in Figure 2 that we doubled the Central Processing database to indicate that there can be multiple instances of this style of database, each sending Notes mail messages to remote servers to have a certain task carried out relative to the Central Processing database’s purpose. We depicted the processing of the scheduled agents that takes place on the remote servers as splats.

This pre-CORBA solution comes with a lot of overhead: you have to create, and then manage, replica databases on all these different servers, not to mention the Central Processing database on the central server. From a performance perspective, you have to wait for the Agent Manager to schedule each agent located on remote servers, which can take some time depending on the number of scheduled agents that your agent is competing with. Also, the number of Agent Manager executives, or processes, specified in a remote server’s Server document (on the Server Tasks, Agent Manager tab, under the heading “Max concurrent agents”) determines the number of agents that can run at any given time. The more “executives” you have running on your server, the more agents can run concurrently. Just be aware that while increasing the number of Agent Manager executives can improve the scheduling time for your agents, as
more agents run concurrently, processing power is taken away from other tasks that run on the server. So with this approach, although you have a better distribution of processing, you are at the mercy of Agent Manager.

One alternative, of course, would be to run the agent that creates the mail databases on a Notes client, because a local agent can access databases on any Domino servers in your domain. However, the time it would take to run the agent would be prohibitive, if the client is on a small desktop computer with not much processing power (not to mention the security issues that come about when you run a sensitive task like mail file creation on a client versus a server). And putting a Notes client on a piece of server hardware for the purposes of running the agent is just plain overkill — like driving a Ferrari to get from one end of your driveway to the other. If you’re like us, you could have a tough time justifying that solution to management.

**Figure 3**  
*Distributed Processing via a CORBA-Enabled Java Agent*

Shifting the Paradigm: Moving to Distributed Processing with Java/CORBA Agents

With CORBA in the picture, you can eliminate all the overhead of creating and managing replica databases across several servers.

Still maintaining the Central Processing database (or databases), you can have it manage all your processing with scheduled CORBA-enabled Java agents. These agents check for new documents on a scheduled basis and process them in batches for the amount of time Agent Manager is scheduled to run each agent in its queue. The agents create new mail files on the remote servers as necessary. Using the server with the most processing power in your environment to house your Central Processing database(s) is most advantageous, but not completely necessary. This architecture, which is based on distributed processing (as opposed to distributed databases), is depicted in **Figure 3**.
This diagram is similar to the “pre-CORBA” diagram in Figure 2, except that a Java/CORBA agent in the Central Processing database is communicating via CORBA’s IOP protocol to the remote servers. Notice that the “processing splat” is only on the Central Processing database, indicating that this database is the only place where the Java/CORBA agent is processed.

In this setting, if the agent is doing a task that is mostly I/O bound on the remote server (e.g., creating a database from a template), then that workload is distributed to the remote server, because that is where the database needs to be created (i.e., the agent is writing to the remote server’s disk). If, however, the processing is CPU bound (e.g., highly mathematical/statistical computations that need to be assessed before writing the results to a Notes database), then the central server running the agent will handle the processing. So for I/O bound tasks (a very common type for Lotus Notes), this approach forces much of the processing to the remote box, but with these advantages:

- **Fewer administrative headaches**: With this model you have a single place to maintain code and track errors and a less-cluttered Notes Workspace (since the document processing takes place on one centralized server).

In other words, you can take advantage of distributed processing without the nightmare of trying to manage replica databases across several servers. Personally, we find it a lot easier to go to only one place to gather information or status for an application rather than hunting it down across the entire domain.

- **Flexibility in choosing the servers that are best suited for handling the processing you require**: You have the option of having more than one Central Processing database, each with its own CORBA-enabled Java agent. Thus, you can organize your tasks or business processes into neat little databases that can be moved onto servers that you have designated as “workhorses” (i.e., your most robust servers).

As you can see, this model dramatically enhances our ability to manage applications, compared to our first, pre-CORBA model.

Taking the benefits of distributed processing one step further, let’s now look at the benefits of optimizing your code to run on one server using multithreaded agents with thread pools. (Note that whether you decide to experiment with CORBA or not, you can still take advantage of multithreading as early as Notes 4.6.)

**Performance Benefits of Multithreading**

Java supports multithreading of tasks, which is the ability of a program to run multiple tasks on multiple subjects — modifying data in multiple documents, for example — concurrently.

Using multiple threads to process the guts (i.e., critical section) of your agent allows a significant improvement over serial processing because the program can utilize more of the available CPU and allow interleaving of processes that are CPU and I/O bound.

For example, say one thread is blocked because it is waiting for network I/O (like a response from another server). A second thread could be processing its computational piece of the code while this wait occurs. In serial processing, the second thread would have to wait until the first thread completed. Also, because the program is stored in a contiguous memory space, multithreading allows for efficient swapping of information between threads, if necessary.

While multithreading gives you an obvious gain in overall efficiency, it does come with a cost — context switching. Context switching is the small amount of time it takes to switch processing between threads. In a single-processor system, only one
thread can operate at a time, and the CPU must suspend the processing of one thread in favor of another.

On UNIX platforms (specifically Solaris), context switching is often accomplished by preemption, which is the suspension of one process in favor of another by brute force, usually based on thread priorities.

On Win32 or Mac platforms, context-switching is accomplished by the time-slicing (a.k.a., round-robin) method, which gives each thread a specific amount of time to run before it is suspended in favor of the next thread in the queue.

Because the CPU needs some time to transition between thread states, both methods cause some performance degradation.

With the advent of multiprocessor boxes that can perform parallel processing, the cost of context switching is reduced and the benefits of multithreaded agents can be more readily realized. Parallel processing means that threads can be executed simultaneously based on the number of CPUs available. This mechanism is incredibly powerful when you are dealing with code that is very CPU-intensive, because multiple CPUs can be utilized concurrently to process the code. However, keep in mind that you might generate a condition called “starvation” (meaning other threads are not given the opportunity to run on the CPU) if you do not “yield” the CPU to allow those other threads to run.4

Another thing to consider when writing multithreaded agents is contention for resources, which comes about when you “lock” a resource so that no other process can modify it while your agent is modifying it (this method is also known as creating a “semaphore,” or “mutex” — mutual exclusion). Java provides a mechanism, called synchronization, to create a semaphore that locks the processing of a critical code section of an agent. Once that processing has completed, the semaphore releases the lock and allows another thread to continue processing. If the lock on which another process is dependent is not released, a situation known as “deadlock” occurs.

When designing multithreaded agents, you should try to minimize the contention for resources as much as possible. Typically, you can think of resources in terms of CPU, file I/O, and network I/O (and sometimes user I/O — e.g., waiting for a user’s response). On a single-processor server, if your multithreaded agent utilizes two or more of these resources, you should be in good shape, because one process can do some CPU-intensive piece of the code while another thread may be waiting for network or file I/O to complete. An agent that only uses one of these resources will not as readily reap the benefits of multithreading, unless it is running on a multiprocessor box, because other processes on the machine will likely be in contention for such resources.

In our walk-through of building the sample application, we’ll explain how to implement a multithreaded agent, and we’ll demonstrate how to use Java synchronization. But first, we have to set up and test our Java/CORBA environment.

Creating a Java/CORBA Environment

In an earlier article, Bob Balaban explained how to set up a Domino server to host CORBA applications so that remote machines can access Domino objects on that server (“Programming Remote Java Applications and Applets for Domino R5,” THE VIEW, November/December 1998). We highly recommend reading that article for a good foundation in understanding how to get your CORBA applications to run.

Since Bob’s article was based on the beta release of R5, we will briefly review the set-up here to take account of changes as of Domino R5.0.1a.

4 You’ll find some great information on this topic in the book Java Threads, by Scott Oaks, et al., published by O’Reilly.
Setting Up the Domino R5 Server for CORBA Applications

During the installation of Domino Release 5 on your development server, choose the Advanced option in step 2, and, as shown in Figure 4, mark the checkbox for the Domino IIOP task (DIIOP) in step 3.

Once you have the server installed, you will need to edit the Security tab of the Server document:

1. Under Server Access, set the “Access server” field to include either a wildcard (e.g., */JavaAgnt/SomeCompany) or your explicit username. This entry will be used to sign the CORBA/Java agent that we will create. I recommend using a wildcard and creating a special agent OU for the purpose of certifying IDs to run CORBA-enabled Java agents in your applications.

Note that you can add Anonymous to this field if you do not wish to authenticate CORBA sessions in your agent (we DO NOT recommend this however).

2. Under Agent Restrictions, indicate which users or groups can run restricted (or unrestricted) LotusScript/Java agents on this server.

3. Under the IIOP Restrictions, indicate which users or groups can run a remote restricted (or unrestricted) Java program to access this server.

These items are the most crucial for hosting Java/CORBA applications; most of the others are set by default and do not need to be changed. Figure 5 shows you how we set up our own server.

Other settings to be aware of include the number of threads and “Idle session timeout” (on the Server Internet Protocols/IIOP tab). “Idle session timeout” is a new setting in Domino R5.0.1a. If you expect a lot of activity or network congestion, you can adjust these numbers later, but for now the defaults should be fine (see Figure 6).
Figure 5  
**Server Document Security Tab**

![Server Document Security Tab](image1)

Figure 6  
**DIIOP Thread Pool Setting**

![DIIOP Thread Pool Setting](image2)
Now that you have the Server document set correctly, add the following to the notes.ini file of your server:

```
JavaUserClasses=[absolute path to the ncso.jar]
```

The ncso.jar file is the compilation of Java classes that enable CORBA. You need to add it to your server’s notes.ini to prevent any class definition errors. The JavaUserClasses parameter basically allows you to set the CLASSPATH for the server’s environment. It appends the path(s) you include in this parameter to the server’s existing CLASSPATH, if it already exists. We found this parameter to be invaluable for successful testing with CORBA. The ncso.jar file can usually be found in “c:\lotus\notes\data\domino\java\ncso.jar.”

Testing the Implementation with a Simple Java/CORBA Agent

We need to verify that everything is set up correctly by writing our first Java/CORBA agent. The agent we’ll show you is similar to Bob Balaban’s preliminary agent in his article; we decided to reuse it here so you could have a small CORBA program for testing your implementation.

Create a new agent, and import the code listed in Figure 7 (or, write the agent natively using the new agent IDE in Notes). This agent will get the title, user’s name, platform, and remote server’s name. Schedule the agent to run more than once a day.

For testing purposes, you will need two networked machines, each with a Domino R5 server installed (you can install one of the Domino servers on your Notes client machine, if you wish). Remember to run DIIOP on both servers, because this task is the one that allows the servers to communicate via CORBA’s IIOP protocol over TCP/IP.

Now create a new database on one of the servers you just set up, and create a scheduled agent in this database with the code listed in Figure 7. Wait for the agent to run as scheduled, and then check its output from the server’s remote console, or verify it in the server’s log (log.nsf).

The first line of code contains the import statement; it is what allows this piece of Java code to be compiled. On line 2, you see that this class, aptly named “RemoteAgent,” extends AgentBase, which allows us to run this code in the context of an agent.

On line 6, we instantiate a Domino object called NotesFactory. You can find this object at the top of the class hierarchy in the Domino object model (called “Domino Objects”). By instantiating this object, we ensure that Notes sessions can be derived after this point. On line 8, we create a String variable for our remote server’s hostname (in this case, we used our server’s IP address). Note that the hostname
you use here must be in the DNS (Domain Name System). Otherwise, you will have to use the server’s IP address instead, so that it will properly resolve to the correct server. A good check to verify your connection with the remote server is to ping its hostname or IP address from the source server that will be running the agent. If this is successful, you know your agent should be able to get to this server.

On line 11, we create the session by inputting the remote server’s IP address, our Notes shortname (or common name), and our Internet password (be sure that the Internet password you use is set in your Person document). This line is what allows us to authenticate with another server. Now we can access the methods under the Session class with the exception of the AgentContext class, because our remote session is not running under an agent’s context; rather, it is established via IIOP (see the sidebar on page 92).

Lines 13-18 demonstrate how you can verify the remote server’s name, user’s name, platform, and the database title, but this can really be any task of your choosing.

This example of setting up and testing a CORBA agent implementation demonstrates how to get the most trivial data from a remote database. Now let’s see what remote access can really do for us.
Example Application: Automatically Creating Mail Files for Notes Users on Several Domino Servers

Let’s assume that some fairly dynamic information is coming into a mail-in database on a daily basis — the names of people for whom mail files must be created. Each mail file is to be created on one of several different Domino servers. In this section, we are going to show you how this can be accomplished using a multithreaded, CORBA-enabled Java agent. Begin by downloading the demonstration application, “corbaMFile.nsf,” from THE VIEW’s Web site (www.eVIEW.com). It has two agents. The first agent, called “Create Fake MailDB Requests,” is a LotusScript agent that you can run manually. It simply creates some “fake” documents for demonstration purposes. You can see one of these documents in Figure 8. It contains the name of a person for whom a mail file must be created, and a server location for the mail file and its pathname. In the demonstration application, this agent takes the place of the users who would be mailing in the documents in a real-world application. You will need to modify it a bit to input the names of the servers specific to your environment. The documents are listed in a view, shown in Figure 9, called the “Unprocessed” view.

The second agent is a CORBA-enabled Java agent, called “RemoteAgents.” This is a scheduled agent that checks the Unprocessed view and creates a mail file for each person in the Unprocessed view, on the server specified.

RemoteAgents processes the documents in batch (i.e., all documents created in the Unprocessed view since the last time the agent ran). If you were to use this application in your own Notes and Domino envi-

How to Enable CORBA with SSL

In R5, CORBA can communicate over an encrypted channel via Secured Sockets Layer (SSL). You may be curious as to how to implement an encrypted channel since you will be passing your ID name and password over the network via IIOP.

1. Acquire SSL certificates from your favorite Certificate Authority (e.g., Verisign) and install them on your server (details on how to do this can be found in the Administrative help or in the Server Certificate Admin database located in the root directory of your Notes server).

2. Enable CORBA to communicate on the specified port in the Server document, under the Ports/Internet Ports/IIOP tab.

3. Modify the example (or any) CORBA agent to enable SSL security, as follows:

```java
09 NotesFactory nf = new NotesFactory();
10 String[] args = new String[0];
11 args[0] = "-ORBEnableSSLSecurity";
12 String host = "192.168.0.146";
13 Session ses = nf.createSession(host, args, "lsacco", "xyz@123");
```

4. Be sure your classpath includes TrustedCerts.class, which is usually found in c:\lotus\domino\data\domino\java.
Figure 8  
**A Document Requesting Creation of a New Mail File**

![Mail Database Request Form - Lotus Notes](image)

Figure 9  
**The “Unprocessed” View**

![CORBA - Mail File Creation - Request/Unprocessed - Lotus Notes](image)
In an environment, you might find that sometimes there are many documents to process, while other times there are none. To enable the agent to use processing threads effectively (without overloading the system), we use a thread pool to manage the number of threads that are running at any given time.

The concept of a thread pool is fairly straightforward. Basically, you can think of threads as being like the books in a library. Just as you check out a library book and return it so someone else can use it, the threads in a thread pool are taken out and returned. The CPU (along with your code) acts as the librarian by allocating threads to be used for additional processing. The difference is that when a thread is returned, it is “claimed for dead,” which means you can still access data from its instantiated object (however, we recommend setting it to null to prevent memory leaks), but its position in the pool is open for a new thread to be spawned.

Thread pools are especially effective for managing databases that are dynamic in nature, such as mail-in databases, where neither the amount of data coming into the database, nor the time when it comes in, can be accurately predicted.

The demonstration database contains a profile document for managing several variables. It resides in a separate view, which we called “Profiles\Global.” You can see what this profile document looks like in Figure 10.

This document is primarily used for lowering or raising the number of threads in your pool based on the type of hardware on your server and/or the type of processing that will be taking place. If you increase the number of threads in the Global Profile document of your application, you will also want to at least match that number in the Server document as well (it’s found in “Number of Threads” field on the InternetProtocols tab, under IIOP; the default is 10).

---

Figure 10  
*The Global Profile Document*

---

1 A thread is a path of execution within a computer process.
The Global Profile document also allows you to set the administrator’s shortname and password (used to establish the remote session), as well as the number of minutes the agent should run during daytime hours and nighttime hours (these values should be a few minutes less than the Agent Manager settings in the Server document).

Because this document contains your Administrator’s password, it is encrypted. Be sure to choose public key encryption so that only your ID and the server’s ID can see the document.

In Figure 11 you can see how we set up public key encryption, using a group called “CORBA Servers,” which contains the names of servers that will be running Java/CORBA agents.

The “RemoteAgents” Parent Class

The code for the “RemoteAgents” parent class is listed in Figure 12. The objective of this code is to check the Unprocessed view for new documents containing names and locations for new mail databases and to create the thread pool that the RemoteAgentsChild class will use for processing the documents.

There are only two methods in this class: NotesMain() and getProfileDocument().

```
1 import lotus.domino.*;
2 import java.util.*;
3
4 public class RemoteAgents extends AgentBase{
5     public Session ses;
6     public Database db;
7     private static int lastDocIdx = 0;  // Pointer to index in doc vector
8     private int dayTimeLimit, nightTimeLimit, numThreads;
9     private String adminName, adminPwd;
10
11     public boolean getProfileDocument()
12     {
13         try {
14             View profView = db.getView("GlobalProfile");
15             Document profDoc = profView.getFirstDocument();
16             if ( profDoc != null ) {
17                 dayTimeLimit = profDoc.getItemValueInteger("DaytimeLimit");
18                 nightTimeLimit = profDoc.getItemValueInteger("NighttimeLimit");
```

(continued on next page)
Figure 12 (continued)

```java
    numThreads = profDoc.getItemValueInteger("NumThreads");
    adminName = profDoc.getItemValueString("AdminShortName");
    adminPwd = profDoc.getItemValueString("AdminPassword");
    return true;
  }
  }
  catch (Exception e) { e.printStackTrace();
} return false;
}
public void NotesMain() {
    Object waitSem = new Object(); // Used for wait/notify semaphore
    boolean foundone = true; // Used for wait loop for threads
    // Default time limit (in minutes) if none set in profile document
    int TIME_LIMIT = 10;
    try {
        System.out.println("************ Starting Remote Agents *************");
        NotesThread.sinitThread();
        ses = this.getSession(); // Used for wait/notify semaphore
        AgentContext ctx = ses.getAgentContext();
        db = ctx.getCurrentDatabase();
        Agent currAgent = ctx.getCurrentAgent();
        String agentOwner = currAgent.getOwner();
        // Create a log to return information to the agent's log
        Log alog = ses.createLog(currAgent.getName());
        alog.openAgentLog();
        // Get profile document information
        if ( getProfileDocument() ) {
            // Instantiate AgentTimer
            AgentTimer amgrTimer = new AgentTimer(ses);
            amgrTimer.setStartTime();
            String createdTime = amgrTimer.getStartTimeOnly();
            int hourNum = Integer.parseInt( createdTime.substring(0,2));
            TIME_LIMIT = dayTimeLimit;
            if ( createdTime.endsWith("PM") ) {
                if ( hourNum > 6 && hourNum < 13 )
                    TIME_LIMIT = nightTimeLimit;
            } else if ( createdTime.endsWith("AM") ) {
                if ( hourNum > 0 && hourNum < 5 )
                    TIME_LIMIT = nightTimeLimit;
            }
            amgrTimer.setTimeAvailable(TIME_LIMIT);
        }
        // Get the view with the unprocessed documents and iterate through them
        View view = db.getView("Unprocessed");
        Vector docVct = new Vector();
        Document doc = view.getFirstDocument();
(continued on next page)
```
Figure 12 (continued)

```java
// Do not do any processing if no docs in view
if (doc != null) {
    // Set view update to false so changes don't affect doc pointers
    view.setAutoUpdate(false);
    while (doc != null) {
        // Be sure document is a "request" type and is signed
        // Typically, it is useful to also verify the signer to see if they can process these requests
        // by checking a group in the Domino Directory — something I leave up to you
        if (doc.getItemValueString("Form").endsWith("Request") && doc.isSigned()) {
            docVct.addElement(doc);
        }
        else {
            doc.replaceItemValue("Processed", "Yes-Failed");
            doc.replaceItemValue("Comments", "Not an appropriate request for this database or
            was not signed.");
            doc.save();
        }
        // We don't recycle the doc here because we need it later — it will be recycled when
        // the thread is finished with it
        doc = view.getNextDocument(doc);
    }
    RemoteAgentsChild threadProcs[] = new RemoteAgentsChild[numThreads];
    // if nothing in vector, exit gracefully by setting boolean
    if (docVct.isEmpty()) {
        foundone = false;
    } else {
        // Create a thread pool
        for (int idx = 0; idx < numThreads; idx++) {
            if (idx < docVct.size() && docVct.elementAt(idx) instanceof Document) {
                Document tempDoc = (Document)docVct.elementAt(idx);
                threadProcs[idx] = new RemoteAgentsChild(db, alog, waitSem, tempDoc,
                        agentOwner, adminName, adminPwd);
                threadProcs[idx].start();
                lastDocIdx++;
            }
        }
        alog.logAction("All threads started");
    }
    while (foundone) {
        // Wait to be pinged by a child class object
        try {
            synchronized(waitSem) {
                waitSem.wait();
            }
        }...
    }
}
(continued on next page)
```
catch (InterruptedException iex) {}

foundone = false;

// See who it was that notified us
for (int idx = 0; idx < numThreads; idx++)
{
    if (threadProcs[idx] != null && threadProcs[idx].isDone())
    {
        // Destroy the old thread and reuse its spot to create a new thread
        threadProcs[idx] = null;

        if ( lastDocIdx < docVct.size() && amgrTimer.isTimeAvailable() )
        {
            // Change status to completed
            Document tempDoc = (Document)docVct.elementAt(lastDocIdx++);
            tempDoc.replaceItemValue( "Processed", "Yes" );
            tempDoc.save(true);

            threadProcs[idx] = new RemoteAgentsChild(db, alog, waitSem, tempDoc,
                                                   agentOwner, adminName, adminPwd);
            threadProcs[idx].start();

            // Need this to keep running the while loop to wait
            foundone = true;
        }
        else
        {
            alog.logAction( "All docs processed...no thread reuse necessary." );
        }
    }
    else if (threadProcs[idx] != null)
    {
        foundone = true;
    }
}
else {
    System.out.println("Failed to get profile Document");
    alog.logAction("Failed to get profile Document");
}
// Clean up heavyweight C++ back-end objects created off session object
ses.recycle();
System.out.println("************ Completed Remote Agents ************");

} catch( NotesException e) { System.out.println("Notes Error #" + e.id + ":" + e.text);}
catch (Exception e) { e.printStackTrace(); }
finally { NotesThread.stermThread();}
NotesMain() is the entry point for the Java agent, and the getProfileDocument() method allows us to get information specific to the server on which this application is running (i.e., the number of threads allowed in the thread pool).

**Lines 5-9** instantiate the variables that we will need. Next, we get some critical variables by calling getProfileDocument() (**lines 11-27**). This section of code gathers the number of threads allocated for the thread pool and the name and password of the server’s Administrator. We will open a remote session with CORBA on the remote server using this name and password.

After we confirm the variables from the profile document, we need to get a handle to the documents in the “Unprocessed” view (**lines 68-93**). As we iterate through these documents, we add them to a Java vector (LotusScript programmers can think of a vector as a variant). The vector is a container for the elements that will later instantiate the threads. If there are no documents to process in the view, the agent exits gracefully.

We now create the thread pool (**see lines 95-118**). From the value we collected in the profile document regarding how many threads should be in the pool, we create an array of threads, which are really identified as a class, RemoteAgentsChild. We iterate through the document vector, and spawn or start, the threads. It should be obvious now that we will only spawn the number of threads allowed from the profile document.

That’s basically it. The RemoteAgentsChild and CreateMailFile subclasses will do the rest of the work.

Towards the end of the code, you can see that there is a while loop, beginning at **line 120**. This loop will run infinitely, until there are no more threads to process. What is this all about?

At this point, the RemoteAgents parent class has spawned threads that the RemoteAgents child class (discussed next) is using to process the documents from the Unprocessed view. As each thread completes its task, RemoteAgents destroys the old thread (by setting it to null) and spawns a new thread, but only if there are additional documents to process in the Unprocessed view that the previous spawned threads could not get to.

Look at the while loop (**lines 120-161**). Our semaphore object on **line 124** is waiting to be pinged by the child class so that it can destroy the child’s thread and start a new thread for processing more unprocessed documents. Using the built-in Java functionality for managing threads, we call this waiting to be **notified**.

Once the thread notifies us that it has been completed, we must identify which thread has notified us (see **lines 130-160**). We reuse that spot in the thread pool to instantiate new threads, if necessary. This allows us to continue processing documents that are a part of the vector mentioned above. The agent will continue to run until all the documents are completed from the input queue, (i.e., until the boolean variable returns false on **line 128**).

A caveat is that an agent can run only a finite number of minutes according to the “Max LotusScript/Java execution time” setting in your Server document. Based on how often you schedule this agent to run and how many messages the mail-in database needs to process, you should establish an appropriate time setting for Max LotusScript/Java execution time. You don’t want it too low, because you might not give your agent enough time to complete; on the other hand, you don’t want to set it too high, because your agent could be in contention with other agents that need this resource.

The “AgentTimer” class (included in the downloadable sample database) checks for how much time the agent has to continuing processing. It is implemented on **lines 51-67**.

If you want to monitor how long the agent ran,
use the AgentTimer class to compare against the values found in the database’s profile document (see lines 17-18). You may want to raise the number of threads in the thread pool if the agent is taking too long. If the agent is in heavy contention with server resources, you can lower the number of threads.

**The “RemoteAgentsChild” Class**

The code that actually performs the remote database processing is the “RemoteAgentsChild” class, listed in Figure 13. This class is responsible for the core processing of the documents in the Unprocessed view.

---

**Figure 13**

The “RemoteAgentsChild” Class

```java
1 import lotus.domino.*;
2
3 public class RemoteAgentsChild extends NotesThread
4 {
5     private Database db;
6     private Log alog;
7     private boolean done;
8     private Object waitSem;
9     private Session remoteSes;
10    private NotesFactory nf;
11    private Document procDoc;
12    private String agentOwner, admin, adminPassword;
13
14    public RemoteAgentsChild(Database d, Log l, Object sem, Object procObj, String agentOwner,
15        String admin, String adminPassword)
16    {
17        this.db = d;
18        this.alog = l;
19        this.done = false;
20        this.waitSem = sem;
21        this.procDoc = (Document)procObj;
22        this.agentOwner = agentOwner;
23        this.admin = admin;
24        this.adminPassword = adminPassword;
25    }
26    public synchronized boolean isDone()
27    {
28        return this.done;
29    }
30    public void runNotes()
31    {
32        try {
33            this.sleep(500);
34```
Figure 13 (continued)

```java
35       // Determine type of request and server to run on
36       String requestType = procDoc.getItemValueString("Request");
37       String remoteSvr = procDoc.getItemValueString("NewServer"); // Must be in DNS
38
39       if (requestType.equals("CreateMailFile")) {
40           // Create a remote session on the server
41           nf = new NotesFactory();
42           remoteSes = nf.createSession(remoteSvr, admin, adminPassword);
43
44           // Store docs in a document array to be processed
45           CreateMailFile mailFile = new CreateMailFile(remoteSes, procDoc, agentOwner);
46           if (mailFile.createMailDB()) {
47               alog.logAction(mailFile.getStatusMsg());
48               procDoc.replaceItemValue("Comments", mailFile.getStatusMsg());
49               procDoc.save(true);
50
51               if (!mailFile.setACL()) {
52                   alog.logAction(mailFile.getStatusMsg());
53                   procDoc.replaceItemValue("Processed", "Yes-Failed");
54                   procDoc.appendItemValue("Comments", mailFile.getStatusMsg());
55                   procDoc.save(true);
56               }
57           } else {
58               alog.logAction("This type of request does not exist.");
59           }
60
61           // Recycle doc here because it’s no longer needed as a reference
62           procDoc.recycle();
63           catch (NotesException e) { System.out.println("Notes Error #" + e.id + ": " + e.text);}
64           catch (Exception e) { e.printStackTrace(); }
65           finally {
66               synchronized (this) { this.done = true; }
67
68               // Notify the parent thread
69               synchronized (waitSem) { waitSem.notify(); }
70           }
71       } else {
72           alog.logAction("This type of request does not exist.");
73           procDoc.recycle();
74       }
75       catch (NotesException e) { System.out.println("Notes Error #" + e.id + ": " + e.text);}
76       catch (Exception e) { e.printStackTrace(); }
77       finally {
78           synchronized (this) { this.done = true; }
79
80           // Notify the parent thread
81           synchronized (waitSem) { waitSem.notify(); }
82       }
83       }
```
RemoteAgentsChild does the following:

1. It determines whether the request is of the appropriate type to be processed by this agent (i.e., “CreateMailFile”); otherwise the request is ignored. Keep in mind, in your own testing, that you can build on this format to include various other requests that you may have.

2. If the document is of the proper type, RemoteAgentsChild opens a remote session via CORBA to the appropriate remote server so that we can create the mail database there.

3. This class then instantiates the CreateMailFile class, which actually encapsulates all the functionality required to create the mail file in a nice, neat class. (We’ll look at the CreateMailFile class in more detail later.)

4. Lastly, when all processing has been completed, this class notifies the parent class so that it can delete the child thread to help restore memory and spawn a new thread.

Looking at the code (line 3), you see that this child class extends NotesThread; so think of this class as a thread of execution.

Each time this class is instantiated, it creates its own set of private variables, which are instance variables referenced from the calling or parent class (lines 5-25). This area of the code is known as the constructor. It is called whenever you use the keyword “new” to instantiate the class. This mechanism is similar to what happens in LotusScript when, for example, you use the keyword “New” to create a document and you have to pass the database into the constructor. In this case, however, we are passing in several variables to get the class set up for executing the core code (processing the documents in the Unprocessed view). As a rule, it is good to share as many variables as you can get away with — especially Session, Database, and View objects. By sharing variables, you allow a more efficient use of memory since the class does not have to create the variables over and over again.

After the constructor is called, the thread is ready to run its critical section — the method called runNotes(). A standard for all Notes Java agents that extend the NotesThread class is that, once the thread has started, the runNotes() method is called to do the actual processing. First, however, we need to instantiate a Notes object called NotesFactory (line 41), which is what allows us to create a session on a remote server.

In lines 35-39, the runNotes() method queried the contents of an Unprocessed document, so the appropriate action can now take place. Because the document to be processed contains the information regarding which server the action needs to take place on, in line 42 we can create the session using that hostname and the administrator’s name and password, which were set in the constructor.

Since Java can nicely encapsulate data and methods to allow us to create mail files, we have chosen to put the code for creating mail files into another class, CreateMailFile (which is instantiated on line 46 and to which further method calls are made — e.g., createMailDB() on line 47, and setACL() on line 52). If a “CreateMailFile” request (determined from the “Request” field in the document being processed) is not found, a message will be placed in the agent log, and the thread will die.

For this example, we are just creating mail file databases. However, note that we have made the runNotes() method expandable by checking the request type on line 39, so you could add any number of request types.

Lastly, we need to flip the bit on the boolean variable done, and then notify the parent class to no longer wait (see lines 77-80). Once the parent is notified, the current thread is declared dead and a new thread is spawned, if necessary. If no more threads need to be spawned, the while loop in the parent class will discontinue and wait for the rest of the threads to complete before shutting the agent down.
The “CreateMailFile” Class

Looking at the CreateMailFile class in Figure 14, we see how the mail files are created.

In this class, we are creating a mail file and setting its ACL. As before, we have identified the class variables in lines 8-14. When this class is instantiated, the constructor performs a getDatabase() call on the remote server using the remote session variable that is passed into the method (see line 29).

Perusing the remaining code, you can see that the mail database is created from the mail template identified in a constant (lines 34-63), and the ACL and database title are set appropriately (see lines 67-96).

We will not elaborate too much here, because you can derive most of what is happening from the code itself. The thing to remember is that this class could really be anything that you want your agent to accomplish.

The CreateMailFile class creates a mail file on a remote server. When this class is instantiated, the constructor performs a getDatabase() call on the remote server, then the mail database is created from the mail template identified in a constant, and the ACL and database title are set appropriately.

Figure 14

The “CreateMailFile” Class

```java
1 import lotus.domino.*;
2
3 class CreateMailFile {
4     private final static String MAIL_TEMPL_R5_FNAME = "mail50.ntf";
5     private final static String MAIL_TEMPL_R5_NAME = "Notes 5.0";
6     private final static String ACL_ADMIN = "Administrators";
7
8     private Session ses;
9     private Document doc;
10    private Database mailDB, templR5DB;
11    private ACL mailACL;
12    privateACLEntry userNameEntry, ownerEntry, adminEntry;
13    private String agentOwner, statusMsg, pathVar, templVar, userNameVar;
14    private Name userName;
15
16    public CreateMailFile(Session ses, Document doc, String agentOwner) {
17        try {
18            this.ses = ses;
19            this.doc = doc;
20            this.agentOwner = agentOwner;
21            // Gather document attributes
22            pathVar = this.doc.getItemValueString("Path");
```

(continued on next page)
tempVar = this.doc.getItemValueString("MailTemp");

userNameVar = this.doc.getItemValueString("UserName");

userName = this.ses.createName(userNameVar);

// Get handle to mail template database

// Get handle to mail template database

templR5DB = ses.getDatabase(null, MAIL_TEMPL_R5_FNAME);

} catch( NotesException e) { System.out.println("Notes Error #" + e.id + ":" + e.text);}
catch (Exception e) {e.printStackTrace();}

public boolean createMailDB() {
    try {
        mailDB = ses.getDatabase(null, pathVar);

        if ( mailDB == null || !(mailDB.isOpen()) )
            { 
                if ( tempVar.equalsIgnoreCase(MAIL_TEMPL_R5_NAME) ) {
                    if ( templR5DB != null ) {
                        mailDB = templR5DB.createFromTemplate(null, pathVar, true);
                    }
                    else {
                        statusMsg = "Could not locate mail template " + MAIL_TEMPL_R5_NAME;
                        return false;
                    }
                }
                else {
                    statusMsg = "Mail Template does not exist on this server as specified";
                    return false;
                }
            }
            else {
                statusMsg = "Mail Database already exists on Server." ;
                return false;
            }
    }
    catch( NotesException e) { System.out.println("Notes Error #" + e.id + ":" + e.text);return false;}
catch (Exception e) {e.printStackTrace();return false;}

    statusMsg = "Mail DB created successfully.";
    return true;
}

public String getStatusMsg() {
    return statusMsg;
}

public boolean setACL() {
    try {
        if ( mailDB != null && mailDB.isOpen() ) {

            mailDB.setAccessControlList(((NotesDatabase)mailDB).getACL());

        }
    }
    catch( NotesException e) { System.out.println("Notes Error #" + e.id + ":" + e.text);return false;}
catch (Exception e) {e.printStackTrace();return false;}

    }
Figure 14 (continued)

```java
70    mailACL = mailDB.getACL();
71
72    // Now set the user’s name as the db title
73    mailDB.setTitle(userName.getCommon());
74
75    // Add the user’s name to the ACL and remove Admin ID
76    userNameEntry = mailACL.createACLEntry( userName.getCanonical(),
77                                          ACL.LEVEL_MANAGER);
78    ownerEntry = mailACL.getEntry(agentOwner);
79    ownerEntry.remove();
80
81    // Add the administrator’s entry
82    adminEntry = mailACL.getEntry(ACL_ADMIN);
83    if ( adminEntry == null )    {
84        adminEntry = mailACL.createACLEntry(ACL_ADMIN, ACL.LEVEL_MANAGER);
85    }
86
87    mailACL.save();
88    }
89    else {
90        statusMsg = "Could not access maildb to update ACL.";
91        return false;
92    }
93  }
94   catch( NotesException e) { System.out.println("Notes Error #" + e.id + ": " + e.text); return false;}
95   catch (Exception e) {e.printStackTrace(); return false;}
96  return true;
97 }
```

We encourage you to experiment with various applications of this technology. Realize that a CORBA-enabled agent can be especially useful for reading or writing information on remote servers. For example, in the sample agent, we could have easily added code to open a centralized logging database on another server and write a status report document there, so as to confirm that the creation of the mail database was either a success or a failure without having to mail messages back and forth.

Further Reading

We have found the following to be invaluable sources of information on Java and CORBA development in Domino:

- As mentioned earlier, for an excellent overview of CORBA as applied to Domino R5, we recommend reading Bob Balaban’s article, “Programming Remote Java Applications and Applets for

- You will find a great Lotusphere presentation on using CORBA with Domino R5 at: http://media.lotus.com/lotusphere99/ad309.pdf

- To get started on understanding how CORBA works with Java, visit: http://developer.java.sun.com/developer/onlineTraining/corba/

- Want to learn more about threading? Here’s a good place to start: http://www.gamelan.com/javaprogramming/threads/

- Here’s a great forum we frequent, hosted by Bob Balaban, where you can find good information on this and all sorts of other Notes technologies: http://www.looseleaf.net

**Conclusion**

We think Java/CORBA agents represent an exciting new technology for Domino developers. Our hope is that by highlighting the benefits of this technology and demonstrating its use, we have encouraged you to consider using remote Java agents in your own enterprise. Like all new technologies, it will proliferate and mature as developers begin to share their experiences with it. To that end, we look forward to hearing from you.

Louis J. Sacco is a member of the ITM Messaging and Groupware Infrastructure Development team at DaimlerChrysler, where his duties involve Domino infrastructure development, including database development, enterprise integration, and programming using LotusScript, Java, and the C/C++ API toolkits. He also helps advocate the use of Lotus Notes through various corporate Notes databases and knowledge transfer sessions. He is a Principal Certified Lotus Professional in Application Development, a Sun Certified Java Programmer, and has recently received his Masters of Science degree in Computer and Information Science from the University of Michigan–Dearborn. Louis can be reached at the local pub or at occasl@yahoo.com (for best results, be sure to address him as Lou!).

Bruce R. Maxim is an Associate Professor of Computer and Information Science at the University of Michigan–Dearborn, College of Engineering and Computer Science. His research interests include software engineering, user interface design, multimedia software, expert systems, and computer game design. Dr. Maxim has published a number of papers on the animation of computer algorithms, knowledge engineering, and educational computing applications. He is also co-author, with Elliot Koffman, of “Turbo Pascal: Problem Solving and Program Design,” a best selling introductory computer science text. His e-mail address is bmaxim@umich.edu.
YES. Sign me up for a one-year subscription (six issues) to THE VIEW.

☐ $335 within the U.S. and Canada
☐ $375 outside the U.S. and Canada

YES. Please send me the following issues of THE VIEW. Each issue is $49.

(For a complete listing of all past articles, visit our Web site at www.eVIEW.com)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan/Feb</td>
<td>Jan/Feb</td>
<td>Jan/Feb</td>
<td>Jan/Feb</td>
<td>Jan/Feb</td>
<td>Jan/Feb</td>
</tr>
<tr>
<td>Mar/Apr</td>
<td>Mar/Apr</td>
<td>Mar/Apr</td>
<td>Mar/Apr</td>
<td>Mar/Apr</td>
<td>Mar/Apr</td>
</tr>
<tr>
<td>May/June</td>
<td>May/June</td>
<td>May/June</td>
<td>May/June</td>
<td>May/June</td>
<td>May/June</td>
</tr>
</tbody>
</table>

Number of past issues ___ x $49 each = $ __________

Total Order __________

Place your order by mail, by fax, or over our Web site:

Mail to: Wellesley Information Services
P.O. Box 370
Newton, MA 02459 USA

Fax to: (781) 329-9186
Or order over our Web site:
http://www.eVIEW.com

Name ____________________________ Company ____________________________
Address ________________________________________________________________
City ____________________________ State/Prov __________ Zip/Postal Code __________
Country ____________________________ E-mail ____________________________
Phone ( ) ____________________________ Fax ( ) ____________________________

☐ Check or PO enclosed payable to Wellesley Information Services (in U.S. dollars)

☐ Master Card ☐ VISA ☐ American Express ☐ Diners Club

Card Number ____________________________ Exp. Date __________

Name on Card ____________________________ Signature ____________________________
(Please print)

Questions? Call (617) 527-5100

Mail to: Wellesley Information Services
P.O. Box 370
Newton, MA 02459 USA
Fax to: (781) 329-9186
Or order over our Web site:
http://www.eVIEW.com

Place your order by mail, by fax, or over our Web site: