Chapter 14
A Look Ahead to Domino 5.0

We'll wrap up the book by summarizing the major advances in programmability featured in Domino 4.6, and do a bit of crystal ball gazing about where some of these technologies might be headed in the next major feature release, Domino 5.0.

> <note>aside from a personal involvement with the early development of some CORBA prototypes during my tenure at Iris Associates, I have no personal knowledge whatsoever of specific product plans for Domino 5.0 beyond what has been publicly announced by Lotus. What follows stems from my own opinions and research, and is in no way based on any inside information. I have not asked anyone at Lotus or Iris to comment on specific product plans for this book.

Domino 4.6 Programmability: Summary

I'm biased, of course, but I'd rate the following as the major advances in programmability for Domino 4.6:

1. Notes Object Interface Java binding.
2. Java Agents for Domino.
3. Java Servlets for Domino.

Virtually all the new functionality is more relevant to the Domino Server product than to the Notes Client product, although you can run Java Agents in the Client as well as on a server.

Java NOI

Some people have asked me if the Domino 4.6 Java interface was "ported" from the LotusScript classes. The answer is emphatically no. There was some effort made to keep the Java APIs as close in feeling to the LotusScript classes as possible, but no code was

ported from one to the other. Both interfaces (or "bindings" as we sometimes call them) are thin layers on top of a common set of C++ classes that make up the Notes LSX (which stands for LotusScript eXtension module). The behaviors that are the real class implementations are thus common to both LotusScript and Java (and to the OLE Automation interface as well, for that matter). The fact that the architecture still bears the name "LotusScript" is only an historical artifact at this point.

At one level, having the Java binding to the Notes Object Interface in Domino 4.6 is simply a way of allowing people to write standalone API programs using Java instead of C or C++. Java applications as we've described them in this book are really no different from any C API program, other than in the language of implementation. One advantage of using Java over C is that you get all the nice object-oriented constructs. An advantage of using Java over the C++ API is that with Java you can compile a program on one platform and have it run unmodified and without recompiling it on another, even when the operating systems are different. There are, though, differences in the kinds of functionality covered by the Java class library and the other APIs. Being newer, the Java interface (and LotusScript too because remember, they rest on exactly the same code) does not yet have all the functionality of the C API (the lowest level and in most ways most functional API for Notes), but eventually it will. The rule of thumb is, if the Java API gives you the functionality you need to accomplish your task, then use it. If not, consider using a hybrid approach: Java for the functionality that you can implement using Java, and call-outs to C or C++ code for the rest.

I predict (and hope) Lotus will begin some serious investment in broadening the coverage of both the Java and LotusScript interfaces.

In sum, the overall Notes Object Interface represents the programmable face of the product. You get to manipulate real Notes/Domino objects and without having to go through a user interface. You can automate and schedule behaviors that you invent or customize. The fact that there is now a Java binding to that object interface gives you

both more choices and more functionality: you can make use of nice built-in features of the Java language, and integration with other Java technologies (Servlets, for example) becomes much simpler.

Java Agents

LotusScript Agents have been a fixture of Notes programmability since Release 4.0 in January 1996 (that's about a decade in Web years). The Agent Manager module in the Domino server handles schedule-triggering as well as event-triggering of Agent programs, and together with the NOI infrastructure manages Agent identity and security. Without the NOI classes they wouldn't be able to do much, of course, which is why so much of this book focuses on the functionality and interface of the object model.

The new Java Agent functionality in Release 4.6 is best described this way: Anything you could do with a LotusScript Agent you can now also do with a Java Agent. What could be simpler? Naturally you also get to take advantage of features in Java that aren't in LotusScript, such as easy network access and multithreading.

Java Servlets

The Java Servlet interface is a relatively new and as yet under-explored technology from JavaSoft. The basic idea is very simple though: You can write CGI (Common Gateway Interface) programs in Java, have them triggered by your Web server via a URL, and you can pass parameters to them as well.

I've tried to make the point in this book that Agents are better than Servlets for use in production server-based applications, particularly where security and server administration and control over resources is a concern. I've also offered those of you with working Servlets who want to integrate them with the Domino Agent subsystem with a minimum of pain some source code (Chapter 11) for an Agent that knows how to

load and run Servlets. This gives you the best of both worlds: no need to rewrite a line of code in an existing Servlet, but all the security and identity features of an Agent.

This little technique would be impossible without the Java NOI and Agent capabilities. It's an example of how using Java opens up possibilities for the integration of Domino with other useful technologies.

Overview of Forthcoming Enhancements

So what's my best guess for the future of Domino as it relates to Java, NOI, Agents, and so on? I think that there are some obvious areas where these things can (and probably will) be extended:

1. Java IDE.
2. Remote Object Access: Domino server Object Request Broker (ORB) and NOI for ORB clients. Remote Method Invocation (RMI).
3. Java classes for other standard protocols: LDAP, IMAP, etc.

The following sections discuss each in some detail.

Java Development Environment

There are two large holes in the Java offerings in Domino 4.6: There's no Integrated Development Environment (IDE) for Java, and none of the Notes Client user interface scripting features that you get with LotusScript are available for Java.

I hope (though I don't *know*) that both of these will be addressed in Domino 5.0. The biggest need is for an integrated Java debugger so that the contortions you have to go through now (described in detail in Chapter 9) to debug Java Agents will no longer be necessary.

The next thing that Domino customers will demand is the ability to script the Notes Client with Java (currently you can only do it with LotusScript or @function formulas). Once there's a Java IDE in place, it should be possible (I can say this because I'm not

responsible for implementing it) to hook up all the UI events in the current Visual Basic-like model to Java. Java itself, as we've discussed in Chapter 12, has no real event model; you have to use Java Beans to accomplish serious event processing. So perhaps what we'll see will be akin to what we now have with Agents: You'll write a Java Bean that implements a specified interface (or extends a specified lotus.notes class) that allows it to hook into all the Notes Client event processing. That would be, as they say, way cool.

Remote Access Technologies for Domino NOI

I've made the point throughout this volume that in Domino 4.6 the Java NOI APIs are only available on a machine that has Domino installed. There's a lot you can do with that: Agents, Servlets, Applications, and so on. But you can't do Applets. Domino has all kinds of Applet support built in (you can embed Applets in a Notes document, you can have the Domino server serve up an Applet over HTTP, and you can even have Applets invoke Domino Agents with URLs and parse through the HTML response), but you can't write an Applet that will get downloaded to someone's browser and have that Applet use any of the NOI classes.

The reason is simple: NOI requires that a bunch of Notes DLLs (or the platform equivalent) be present on the machine; all the real object behaviors are implemented in the Notes core code. You, the Applet author, have no idea whether anyone downloading your Applet has Domino installed. Furthermore, even if they did, most browsers won't let an Applet load a DLL into memory because that would be a potential security violation.

There are other techniques available for communicating between Applets and Domino:

- You get two programmable "events" that Domino automatically supports: document open and document save. You can designate Agents that the server will run when these events are triggered from a client's browser.
- You can cause an Agent to be run on the server by invoking it with a URL. The Agent can format HTML output that gets returned to the client, and there's no law that says that HTML has to be displayed. It could simply be parsed by the Applet code as returned data.

These techniques, while useful, are not optimal (to say the least) for serious client/server development efforts.

There are, however, some object remoting technologies out there that will probably be making an appearance in Domino 5.0. The primary one, in which a lot of effort has already been invested at Lotus, is called CORBA, which stands for Common Object Request Broker Architecture. CORBA is a specification, not a product. It is "owned" by a consortium called the Object Management Group (OMG). Any vendor is free to implement a conforming Object Request Broker (ORB) however they like. The purpose of CORBA implementations is to allow objects that reside on a server to have their methods invoked remotely, from one or more client machines. It's a bit like Remote Procedure Call (RPC) mechanisms (which is how the Notes Client talks to the Domino server today), although it was designed from the beginning to be used in object oriented systems.

The next couple of sections go into more detail about CORBA and Domino, and what you will probably see in Release 5.0 in this area, and how it applies to Applets.

Another object remoting technology called Remote Method Invocation has been invented by JavaSoft, and released as part of Java 1.1. RMI is a lot like CORBA, but it is more tightly tied to Java. I don't expect to see any official support for RMI in Domino, but as you'll see in the section below on this topic, it won't be very hard for you, if you need it, to write an RMI server that front-ends the Domino classes.

Finally, no discussion of remote object technology would be complete without at least a mention of Microsoft's DCOM.

CORBA and Domino

CORBA is very interesting in the context of Domino: it provides a way for developers to write programs that either live on or are downloaded to a client machine which use a standard on-the-wire network protocol to access objects that live on a server somewhere, and invoke their methods and have results returned to them. Moreover, because the client and server communicate via a standardized protocol (called IIOP, which stands for Internet InterORB Protocol), the client and server can be implemented in different programming languages.

The details of how CORBA works are way beyond the scope of this chapter (and of this book), but there are plenty of published works on it (see the database of useful links on the CD, references.nsf). I'll just give you the basics here.

On the server side, you have a bunch of object classes. You also have an Object Request Broker (ORB), which is a program written in any language you like (C++ is common, but there are Java ORBs available now too). The ORB is itself a server process—it listens for object requests on a TCP/IP port and dispatches any that come in to the object(s) in question. A request is formatted using the rules of the IIOP protocol: It contains an object reference, a method identifier, and (optionally) a bunch of arguments. The ORB passes the method invocation request along to the referenced object, which executes the request. The results, if any, are returned to the client by the ORB, again using the IIOP protocol.

The object interfaces are specified on the server using a language-independent syntax called IDL (Interface Definition Language). You use IDL to define the methods and properties that each object supports. You then use an IDL compiler to translate your language-independent specification into a set of implementations in a specific

target programming language. The IDL compiler generates source code for classes (or whatever it can generate for target languages that aren't object oriented) that implement "skeletons" in the target language (C++, Java, whatever). You (the server object developer) take those skeletons and use them as the basis for implementing your object behaviors. The object definitions also get stored in something called the Interface Respository (IR). The IR can (if the server ORB supports this functionality) be used by client programs to dynamically find out what the interfaces for a given object class look like.

The IDL compiler and server ORB come with (if you buy a commercially available one, if not you write all this yourself) a runtime support library for the server which handles all the IIOP protocol translation and method invocation dispatching. The ORB also handles various "services" such as security (though there are all kinds of different schemes, from SSL to Kerberos), naming, and location (finding out what machine a particular object class might reside on). See the OMG specs at http://www.omg.org if you need the details.

The most common way you would make use of server objects from a client is to get a copy of those object's IDL definitions. You can then use an IDL compiler (doesn't have to be the same one that was used to create the server objects) to create client-side "stub" classes. These are very lightweight objects whose only purpose is to serve as "proxies" on the client machine for the "real" objects that live on the server. Each defined server class has an identical looking proxy class on the client side, with the identical set of methods and properties. The job of the proxy class when invoked from a client program is to collect the method arguments (if any), package them up as an IIOP request, and ship them off to the server. The proxy then waits (most of the time, there are options supported by some IDL compilers that allow for asynchronicity on the client) for a response from the server. It takes the returned IIOP packets and formats them as returned values to the caller.

The language generated by the IDL compiler for the client stubs must match the language used for the client program, but it can be a totally different language from the one the server skeletons use. That's because the CORBA and IIOP specifications tell you exactly what data types are allowed and how to format them for language-independent processing. The ORBs on both the client and server sides handle any required data format translations.

That's the simple explanation. There's one additional important wrinkle: If your client-side stub objects are implemented in Java, then they can be downloaded to the client machine at runtime and don't need to be permanently installed there. This works because Java has sophisticated dynamic class loading capabilities. The stub classes (together with their underlying runtime support classes that handle sockets, IIOP, and so on) can be packaged together in a .jar file and await your pleasure on some HTTP server somewhere. *Voila:* Applets that can be downloaded to any browser and use IIOP to talk back at an object-to-object level to the server from whence they came.

Domino Server ORB

Domino 5.0 is expected to ship with an ORB developed by IBM, Lotus's parent company. I'm not aware of (and Lotus isn't talking publicly about it yet) what arrangements they plan to make for security (though SSL support would be a good bet), naming services, and so on. The existing LSX architecture fits nicely into the CORBA scheme: Each NOI class already has a C++ implementation; all you'd need is an interface layer for the ORB to dispatch method invocations to (big hand wave here, there are many details that make it a bit less than the slam-dunk I'm portraying).

I would expect to see all the NOI classes available for remote use via CORBA. I also hope Lotus/Iris will decide to publish the actual IDL definitions of the interfaces. Will the remote interface to the Domino NOI look the same as the current LotusScript and Java bindings? It could, but there are some good reasons why it shouldn't.

The easy implementation (I know, because I prototyped one when I worked at Iris) would be to take the existing interface for either Java or LotusScript, and translate it directly into IDL. From there, you can generate the C++ skeletons that get bolted onto the Notes LSX, and you're pretty much done (there are some issues with data type translation and persistent object references, but we'll hand wave that for now). Then you take the same IDL and use another IDL compiler to generate client-side Java stubs. Those stubs can then be used from remote machines in either Applets or applications to talk to NOI objects on a Domino server using IIOP. So far so good.

The problem with this model is that the network performance of this implementation would be pretty bad. Each method/property invocation has to make a round trip from the client to the server and back. For example, let's count the number of trips to the server that the following few lines of client code would have to make:

```
Session s = Session.newInstance();
Database db = s.getDatabase("names.nsf");
View v = db.getView("People");
Document doc = v.getDocumentByKey("Balaban");
String name = doc.getItemValueString("lastname");
```

I count five, one for each line. The one nice optimization that you get for free here is that the View.getDocumentByKey() call does all its work on the server, but this program would perform rather poorly in terms of network i/o, especially if we were doing something like coding a loop over a thousand Documents in a View.

The solution (assuming that we're optimizing for network traffic, which might or might not be the right decision in all cases) is to modify the interface to do as much work as possible on the server. That means changing the object model somewhat, and implementing some different code on the server side. In the Document iteration case, for example, you might want an interface that lets you get a big chunk of data all at once, maybe 10, 20, or 100 Document's worth in one round trip. Then you'd cache all

those bytes on the client, making for better access times and user interaction response overall.

Such a redesign is a two-part effort. First, you need to redesign (or add on to the existing one) the server interfaces to be more optimized for network traffic. Second, you need to do some work on the client side, which is covered in the next section. I expect some changes along these lines in Domino 5.0.

Applet Access to Domino Server ORB: NOI for ORB Clients

On the client side, the simple thing to do would be, as above, to just compile the objects' IDL into Java stubs, and write Applet programs on top of that interface. But as I just said above, this will result in poor performance, especially over the Web, which has relatively high messaging latency (meaning, it's slow).

The real idea is to write a set of Java classes that can be downloaded for Applet use, but which is more optimized for networking. They would sit atop the default set of stubs generated from the IDL (you still need those) and do some intelligent caching of data, along the lines of the 100-Document chunks example above. This would not only reduce overall network traffic, but also improve user response time, which is particularly important for an Applet.

These new client-side classes might well have a significantly different interface from what we now see in NOI, but they might not. One advantage in this sort of architecture is that the client classes don't need to be constrained by what you can implement using IDL: They're a *real* set of Java classes that know how to talk to the IDL-generated stubs. One disadvantage is that doing it this way increases the number of bytes of Java code that have to be downloaded before you can run your Applet: You need to pull down the actual Applet code, the client-side classes, the IDL stubs, and all the required IIOP and networking runtime support code. This situation can be improved somewhat by using .jar files (especially compressed ones); we'll have to see how it goes.

Regardless of the exact implementation that Lotus/Iris settles on, you can expect that in Domino 5.0 you'll be able to write Applets in Java that are stored on a Domino server and downloaded to any Web browser, and that you'll be able to write those Applets to include NOI-like calls to server objects. The possibilities for such an enabling technology are truly wide open. You can imagine, for instance, that people somewhere at Iris and/or Lotus are busily writing some Java Beans that implement slices of Notes Client functionality. There's no reason why these Beans shouldn't be usable by downloadable Applets.

One could also picture the Lotus Kona components (a set of Java Beans designed from scratch to be used by downloaded Applets that implement utility functions such as spreadsheets, word processing, and charting) gaining remote NOI capabilities and talking directly to Notes databases on Domino servers using IIOP. The Domino server could become not only an application platform and data server, but a true object server as well.

On the server side, it would be awfully nice to be able to write your own server objects, register or install them in some way on the Domino server, and have the Domino ORB handle remote access to them automatically. I don't know if this will happen in Domino 5.0, as there are significant architectural hurdles that need to be overcome beyond what it will take to support just the NOI classes remotely.

Remote Method Invocation (RMI)

CORBA isn't the only remote object manipulation architecture out there. JavaSoft has released a set of classes in the java.rmi (and subsidiary) package that essentially do the same sort of job that CORBA does, only it's for Java only and uses a different networking protocol.

RMI is a "lighter weight" technology than CORBA in a couple of ways: It is much easier to install and set up (it comes with the Java Development Kit); it is easier to

develop remotable interfaces; and there is a smaller networking footprint for Applets because less code has to be downloaded.

If you have some existing classes for which you want to develop a remote interface, it's pretty easy to do using RMI (see the RMI documentation for the details on http://java.sun.com/products/jdk):

1. Create a remote interface for each class. Your interface must extend the java.rmi.remote interface, which has no methods. This "marks" your class as being available for remove invocation.
2. For each method in your class, decide whether you want it to be remotely invoked. If you do, add it to your remote interface. Modify your class to *implement* your new interface.
3. Use the RMI compiler (the rmic utility comes with the JDK) to process your remote classes. RMIC generates both skeletons and stubs, along the lines of the CORBA model.
4. Write an RMI server. Unless you need a lot of control over how it works, you can do this very easily by just writing a Java application (with a static main() method) and extending the java.rmi.RemoteServer class. Your server will field remote method calls and dispatch them to the proper classes on the server. See the RMI documentation for details on registration and so on.
5. Write the client program. The client-side code will use the stubs generated by RMIC, which are (as in CORBA) remoted versions of the server-side implementations. You essentially have, for every server class, a remote client proxy class (they have to have different names, unfortunately). Your client code must reference the proxy classes, each of which has the same method signatures as its corresponding server class.
6. The client program can be a downloadable Applet, no problem there. In fact, the amount of code that needs to get downloaded to the client browser is smaller for RMI than for CORBA, because a bunch of the RMI code is already installed on the client as part of the base Java classes (assuming you have a browser that supports a version of Java that contains RMI; not all of them do yet).

7. The client proxies and the server communicate using a private wire protocol, also called RMI.

Can the RMI server classes use NOI? Yes they can, so long as you have Domino installed on the server machine. Does this mean you can write Applets that talk remotely to the Domino 4.6 Java NOI using RMI? Yes, but you have to write the RMI server yourself; Lotus doesn't provide one.

So which is better, CORBA or RMI? "Better" being a slippery concept sometimes, I'll offer the following pros and cons of each:

- **CORBA pros:** Supports multiple languages. Can (in theory) mix and match different vendor client and server ORBs, because all support IIOP. Robust set of services available (though it varies from vendor to vendor). Well understood technology, it has been around for many years. Lots of production systems in existence, fairly large community of practitioners available for assistance and support.
- **CORBA cons:** Not the easiest technology to bring into a production environment, there's a fairly steep learning curve to climb before you're up and running. It is even more difficult to implement your own ORB (though most people never need to). Each vendor's ORB will have different integration techniques for remoting your object classes. This means that you can't count on easily switching vendors for server-side ORBs. Unless you have an ORB implemented in Java, and unless your server objects are also implemented in Java, you can't count on being able to switch platforms easily. It can sometimes be hard to translate your application's data types into the base types supported by IDL.
- **RMI pros:** Very easy to get something working. Everything is in Java, so there's less code overhead, smaller footprint on the client and over the network (in terms of how much code has to be downloaded). No IDL to deal with, all native Java data types are supported. Works on all platforms that can run a Java VM.
- **RMI cons:** It isn't a "standard" in the way that CORBA is. The technology belongs to Sun/JavaSoft, and is controlled by them. You're reliant on a single vendor for technical support and bug fixes. There is not yet a large community of RMI users on which to lean for assistance. When I

> implemented an RMI prototype for Notes in 1996 (using JDK 1.02, before the newer 1.1x became available), I found that I wasn't able to get all the information I needed on the server's multithreading behavior. RMI only works for Java; if you have classes implemented in some other language, you either have to use "native" calls from Java to C or invent some other solution.

In the end, my own opinion is that Lotus made the right decision by going with CORBA over RMI, mainly because of issues like openness and standardization.

DCOM

Microsoft has, of course, a competing technology for remote object access. Anyone who has done any OLE programming in the past few years is intimately familiar with COM, Component Object Model. COM is the technology Microsoft invented to let objects present interfaces to the world, to have those interfaces be standardized and discoverable at run time. COM is itself intimately tied up with C++, but it is a powerful framework. Given a pointer to any COM object, you can find out what interfaces it supports. The object does not necessarily have to implement all its interfaces itself, it can contain child objects and use a delegation model to present contained objects' interfaces as its own.

DCOM simply stands for Distributed COM, and is meant to be the machine-to-machine version of COM, where the invoker of an interface's method does not have to reside on the same machine as the implementor of the interface. OLE already handles a form of remote method invocation: You can invoke a method on an OLE interface from a "container" object's process and have the "server" object which implements the method be in a different process with its own address space. OLE handles the proxying, marshalling, and de-marshalling (packing and unpacking buffers full of argument data) involved in remoting calls across the process boundary, something that is ordinarily

fairly difficult to do. DCOM is (at least conceptually) just an extension of that technology from cross-process to cross-machine, using a network as the link.

DCOM comes for free with Windows NT 4.0, and is supported by a bunch of new classes in the Microsoft Foundation Classes (MFC) framework. DCOM uses YAWP (Yet Another Wire Protocol) to communicate DCOM messages across a network link.

Microsoft has positioned DCOM as a direct competitor to both CORBA and RMI, and can give you chapter and verse on why DCOM is the superior technology (I myself don't necessarily believe that it is superior, as I probably use a definition of the term somewhat different from Microsoft's.) Since I'm not an expert on DCOM I can neither confirm nor deny Microsoft's claims. I do, however, question their motives in conducting what at times looks like a food fight.

A couple of points I will make, however: If you adopt DCOM you pretty much tie yourself to both C++ (Microsoft's version of it) and Windows NT operating systems. I wouldn't be surprised to see Microsoft come out with a Java interoperability story for DCOM (they may already have done, so for all I know), making it a direct competitor to RMI. At this point all I can say is that, like RMI, DCOM is a fairly new technology, and the issues pro and con need some time to settle out.

Summary: Whither Remote Objects for Domino?

I suppose I could descend into punditry and position this section as a "Who will win?" discussion, but I won't. The real question of interest here is: What makes the most sense for Domino, and what are we likely to see in the next revision?

Again, not having any inside information to draw on, I can only make observations from what I know and what I've seen in the press. I know for sure that both a server ORB and some kind of remote client Java classes extending NOI are in the works for Domino Release 5.0. I don't know exactly what they'll look like, but the ORB will most likely conform to all the relevant standards. The new downloadable Java classes will

most likely bear some resemblance to the 4.6 Java NOI, but will probably include some differences as well. They'll be tuned for network performance, and they will be be programmed on top of the "bare" IDL-generated stubs.

Somewhere around the release of Domino 5.0 I would expect to see some cool Java Beans that use the remote interfaces to do interesting bits of Domino functionality within Applet frameworks. Since Beans are mostly self-contained, they don't have to depend on features in the base product, and could ship independently.

Will Domino support either DCOM or RMI? I doubt that we'll see an officially supported RMI in Domino, but I wouldn't be surprised to see some kind of DCOM support. Notes has always been an excellent OLE container, and the expertise within the organization on COM goes fairly deep. There have been rumblings in the press about this too. The issue for Iris and Lotus is likely to be more one of engineering resources and product ship deadlines than one of remote object religion. If they can get the code done in time, they probably will do it (again, just my opinion).

The big issue around supporting multiple remote object technologies for the customer is likely to be one of installation and deployment. There are currently very few network firewalls that will allow IIOP, DCOM, or RMI to pass through; most are programmed to allow HTTP only. This means that if you want to allow users outside your firewall to download Applets that remotely manipulate server objects using one of these protocols, you might have to convince your network administration bureaucracies to allow some new stuff through the firewall. This is usually not an easy task. No problem if you' re just doing your deployment within a corporate intranet, though, as there's usually no firewall involved. Another possibility is to use ORB (or whatever) software that tunnels its protocol through HTTP. There are ways to take (almost) any network protocol and wrapper it with HTTP so that a firewall will let it through. The downside of doing this is that it puts more of a burden on both the server and the client software (they have to catch HTTP requests and responses, decode from them the real

protocol, and then dispatch that message). Furthermore, if everyone starts tunneling all kinds of protocols through HTTP, the firewall vendors will make their firewalls smart enough to figure out what's really going on, and then they'll start refusing tunneled protocols as well. We'll be right back where we are now.

This is not an easy problem, and I expect that real and practical solutions will emerge over time.

Other Programmable Interfaces for Domino

The other big news relating to Domino 4.6 besides the Java NOI (okay, okay, but that's how I see it) was that the Domino server now supports standard Internet application protocols such as Lightweight Directory Access Protocol (LDAP) and Internet Mail Access Protocol (IMAP). LDAP support in the server means that any client who also knows the protocol can gain access to the Domino directory: browse user entries, look up mail addresses, and so on. IMAP support allows clients to read and create mail messages. Thus the two protocols work together so that, for example, from a Web browser you can use a Domino server to create new mail documents, complete type-ahead on recipient names, and send the messages.

These interfaces are both programmable" today, in the sense that you can write a Java (or C++) program on a client machine that opens a socket to the LDAP or IMAP port on a Domino machine, and talks to it using the standard protocol. What would be much nicer, though, would be to have some Java class libraries that handle all the grungy bit packing and communications back and forth for us.

JavaSoft has announced that they are working on a mail package for Java, but design and development are still in progress, and there isn't much we can say about it at this stage. Java support for LDAP is somewhat further along, though, so we can talk a bit about the Java Naming and Directory Interface (JNDI).

JNDI is, as of this writing, in beta. you can download a published specification and pre-release software from http://java.sun.com/products/jndi. JNDI is meant to cover a wide range of naming and directory service functionality, and to do much more than just LDAP, though LDAP is one of the protocols that JNDI will support.

The basic idea of the Naming and Directory Interface is to provide a one-size-fits-all API for network based naming and directory services. The API is defined in two packages: java.naming and java.naming.directory. There are several interfaces that would be used by implementors to expose support for a given naming or directory service. One could, for example, use the naming package interface to implement a service that finds, say, printers on the local network. You'd write a class that *implements* the java.naming.Context interface, and have the Context.lookup() call return objects of class Printer. Other methods on the Context interface allow you to iterate through names that the context knows about, and to define new bindings of names to objects.

The java.naming.directory.DSContext is a special case extension of the java.naming.Context interface. It deals specifically with directory objects and their attributes. The set of attributes belonging to a directory entry is arbitrary: Each attribute has a name and a value. Sounds just like a Notes Document, doesn't it? You'd implement a class using the DSContext interface for each kind of directory you wanted to support, and LDAP could (and probably will) be one of them.

If the API sounds kind of vague (I've left out a bunch of the details, but the JavaSoft spec is pretty complete), that's because JavaSoft wanted the interfaces to be completely general purpose; their aim is to eventually provide implementations of the interface (and allow other vendors to write them as well) that wrapper all kinds of naming and directory services: CORBA, Novell NDS, Sun's NIS, DNS, as well as LDAP. Support for each kind of naming or directory service is "plugged in" to the architecture (JavaSoft's beta release contains support only for LDAP and NIS) by writing classes that implement the required interfaces.

This is a great overall architecture, it's open in the sense that we (the masses of software developers) are not dependent on JavaSoft to support our favorite naming or directory service. We can write our own, and it works in the overall framework just fine, assuming that the design of the basic object model is flexible enough to handle the idiosyncrasies of our particular product.

So, what does this mean for Domino? For one thing, it means that when JNDI is released, you'll be able to browse and update Domino directories from a Java program via LDAP, with no enhancements required on the server side (it already supports LDAP in Release 4.6). There could also be a native Domino implementation of JNDI, but it's probably too soon to say whether there would be significant functional advantages to doing it that way. The same is most likely true of the forthcoming Java mail APIs: I suspect that an early release of whatever JavaSoft defines will support the IMAP protocol, and you'll therefore be able to use Domino mail services that way. Still, the name of the game is choice: You can code to the "raw" protocols, or use a nicely packaged class library instead.

Of course you can also use the lotus.notes package to do anything with Domino that you can do via either LDAP or IMAP, without having to deal with a generic interface that may not support all the functionality of the underlying product. Choice is the name of the game these days, and you can pick the interface that suits your needs best; that's why Domino supports the appropriate Internet standards (such as LDAP and IMAP) in the first place.

Conclusion

If the past is prelude, then I think we can reasonably expect Domino/Notes to continue to evolve as a product for a long time to come. The trend in the programmability area will almost certainly continue toward more openness in several areas:

- More and more of the functionality of the product will become available to developers as Java (and LotusScript) interfaces. The direction toward making as much of the functionality of the older C API available through NOI will continue.
- The Internet standard protocols that Domino already supports will become conveniently programmable over time, and new standards and interfaces will emerge. The synergy between Lotus and JavaSoft may well grow as Domino continues to support protocols and JavaSoft continues to develop standardized Java interfaces to those protocols.
- New remoting technologies such as CORBA and RMI will also continue to evolve. Domino will become tightly integrated with some or all of them, beginning with Release 5.0. Initially we can expect to be able to use CORBA/IIOP to remotely program Domino objects. Future releases are likely to add even more functionality by supporting technologies like automatic failover, naming and server location services, and so on.
- Integrated Java development environment. Badly needed, almost certain to get done one way or another.
- Some of the current Notes Client functionality will very likely be made available in the form of a library of Java Beans. A “Bean-ized” version of the Notes Client view interface is one obvious possibility. These Beans would be available for use in downloadable Applets as embeddable user interface widgets that know how to talk directly back to the Domino server, using the CORBA/IIOP communications facility that we know is already in the works. The nice thing about this is that it gives third-party developers the opportunity to embed slices of the Notes Client user experience into the browser environment with minimal effort.

That about wraps up my list of speculations and prognostications for what's coming in Domino Release 5.0 and beyond. Of course it could all be different next year as the world continues to change at accelerating speed. That's the exciting (and sometimes bewildering) thing about the software industry these days.