NEXTSTEP In Focus, Summer 1993 (Volume 3, Issue 3).

NetInfo Binding and Connecting

Alan M. Marcum and Marc Majka

To get information from NetInfo, a client process must communicate with a server process. There are two ways to contact a NetInfo server: through the domain hierarchy, or directly. For example, NetInfoManager allows you to open a domain using the Open command or the Open by Tag command (both in the Domain menu), and command-line utilities such as **niutil** allow you to specify a domain by path or by tag.

To contact a server directly, all a client needs is the address of the computer running the server and the tag of the database the server is using. contacting the server through the hierarchy is conceptually more complicated, though: The hierarchy must first be built, and then the client must get information about the hierarchy.

This article explains how domains find their parents, how clients find NetInfo servers, and the difference between the two processes.

WHAT IS BINDING?

The NetInfo domain hierarchy is built dynamically, based on the structure of the hierarchy specified in the **serves** properties in the directories in a domain's **/machines** directory. The hierarchy is built from the bottom upÐthat is, from the local domains to their parents, and so on until the root domain is found. (More on this later.)

If there are more than two domains in a NetInfo domain hierarchy, no one process determines the entire hierarchy. Rather, the hierarchy is based on the collected information of *all* the **netinfod**s.

When a **netinfod** locates a server for its parent domain, it's said to *bind* to that server. Exactly how it finds such a server is described below.

Binding and nibindd

Each domain uses binding to find its place in the domain hierarchy. The process that handles binding requests is **nibindd**. This daemon also automatically starts up NetInfo when a computer starts up, by starting the **netinfod** processes that run locally on the computer.

If a domain has a parent domain, the domain's **netinfod** finds a server for the parent by communicating with an appropriate **nibindd**. (If the domain doesn't have a parent, then it's the root domain of the hierarchy.)

Starting netinfod

When a computer starts up, **/etc/rc** invokes **nibindd** to start up NetInfo. As noted above, one of **nibindd**'s tasks is to start the **netinfod** processes that run on a computer. At startup, **nibindd** looks in the directory **/etc/netinfo** for directories that end with the extension **.nidb**, such as **local.nidb**. It starts a **netinfod** daemon for each **.nidb** directory it finds.

When each **netinfod** daemon starts, it checks the consistency of its database. If the database contains a **checksum** file, that means **netinfod** shut down cleanly the last time it ran, and no consistency check is needed. If the file isn't there, though, the daemon examines the database, repairing any damage it finds. This consistency checking can take a long time, up to 20 minutes or even longer for extremely large databases.

Once the database's consistency is verified, the daemon determines whether it's a clone or the master. Then, if the daemon is running the database tagged **local**, it binds to its parent. If **netinfod** is using a database tagged anything other than **local**, it won't bind to its parent yet.

Next, **netinfod** registers itself with **nibindd**. The last step in preparing to provide services depends on whether the server is the master or a clone. If it's the master

server of the domain, it informs all the clones that the master just restarted; if it's a clone, it contacts the master to determine if its database is up to date.

When does a non-**local netinfod** bind to its parent domain? Only when a client process asks for the address and tag of its parent server; see ªContacting the third-level domainº later in this article.

Finding the parent domain

To find its parent domain, **netinfod** examines its database's **/machines** directory, looking for directories with **serves** properties containing the value **../***SomeTag*. Each subdirectory of **/machines** represents a computer, and each computer that has this property and value is a host of one of the parent domain's servers. Once **netinfod** has identified the parent server computerÐor server computers, if there are clonesÐit sends a message to the **nibindd** on it asking to bind to it.

Are you my mother?

In the simplest binding case, the parent domain's computer is explicitly named in the child domain's database. For example:

```
name: cadet
ip_address: 192.42.172.6
serves: ../network
```

In this case, the local computer sends a message to the parent's computer, asking to bind to the parent domain's server. Here, the parent's computer is 192.42.172.6.

The request to bind to a potential parent server is actually a sequence of these ªAre you my mother?º requests (with apologies to P. D. Eastman (Eastman 1988)), one to each potential parent. The child daemon sends a message to **nibindd** at each potential parent, asking if that computer has a **netinfod** that serves its parent domain. It contacts all the potential parent server machines at the same time, rather than contacting one potential parent, awaiting a request, and then contacting another if necessary.

For example, suppose some database is tagged **local**, its parent's database is

tagged **network**, and the child daemon is running on **ranger** whose Internet address is 192.42.172.34. The ªAre you my mother?º message says something like:

> I'm a **netinfod**, on computer **ranger** [192.42.172.34] running with a database tagged **local**. I'm looking for a **netinfod** serving a domain tagged **network** of which my domain is a child. Are you my mother?

The abbreviated version is:

> **local** on 192.42.172.34 binding to **network**; are you my mother?

This request is in the form of a SunRPC to **nibindd**. It's officially called the **NIBIND_BIND** procedure in the NIBIND SunRPC protocol definition.

A brief diversion: protocols and nomenclature

There are two different protocols used by the NetInfo system: the NetInfo Binder Protocol, NIBIND, and the NetInfo Data Protocol, NI. Only messages in the binder protocol are sent to **nibindd**, and only messages in the data protocol are sent to **netinfod**. Both of these protocols use the SunRPC protocol, which can run over either TCP or UDP, depending on which the implementor chose.

Each of the messages in the NetInfo protocols, NIBIND and NI, has a name. As implied above, the ªAre you my mother?º message is the BIND message in the NIBIND protocol. As you might expect in a data access protocol, two of the messages in the NI protocol are **READ** and **WRITE**, performing the expected operations. These messages are **NIBIND_BIND**, **NI_READ**, and **NI_WRITE**, to give you some examples. This nomenclature allows us to distinguish between similar messages for the two different protocols, when those exist.

End of diversion.

Serving binding requests

When an **nibindd** receives an **NIBIND_BIND** request, it first checks to determine whether it has a **netinfod** registered with that tag running on its computer. If there's no appropriate **netinfod**, the receiving **nibindd** ignores the requestÐit never replies

ªno,º only, when appropriate, ªyes.º If instead **nibindd** finds an appropriate **netinfod**, it forwards the binding request along to the **netinfod** process, using the NetInfo protocol's **BIND** request, **NI_BIND**.

The receiving **netinfod** process then checks its **/machines** directory, looking for a directory with the appropriate value of **ip_address** with a **serves** property that includes the value *SomeDomain/tag*. Here, it looks for **/machines/ip_address = 192.42.172.34**, and then looks for a **serves** property value of *SomeDomain***/local**. As with **NIBIND_BIND**, if it doesn't find an appropriate directory, it doesn't send an answer.

If **netinfod** finds the appropriate information in the **/machines** directory, it sends a ªyesº answer back to the caller, which in this case is the **nibindd** on the same computer. The **nibindd** forwards that answer to the child **netinfod** that sent the original request.

When broadcasthost has the serves property

One interesting variation on binding to the parent domain's server is the case where the parent domain is served by **broadcasthost**. The NetInfo directory **/machines/broadcasthost** in a local NetInfo domain normally contains a **serves** property whose value is **../network**. This configuration would appear to indicate that a server with the tag **network** is part of the parent domain of the local domain, and that it runs on the computer named **broadcasthost**. So, does every NetInfo network need to have a computer called **broadcasthost**?

Actually, **broadcasthost** represents not a specific computer, but rather the generic local network broadcast address, 255.255.255.255. This address is used to contact all the computers on the local network or subnet, regardless of their network numbers. When **broadcasthost** is specified as serving the parent domain, the child domain's **netinfod** sends an **NIBIND_BIND** to the **nibindd**s on all computers on the local network implicitly, by sending one message as a broadcast. The message is the usual, ªAre you my mother?º It gets an answer from every computer that's running a parent **netinfod.**

This means that the child **netinfod** might get multiple responses, even many

responses. However, the network won't be flooded with ªnoº answers from computers that don't have appropriate parent domain daemons. The child domain chooses the first reply to arrive.

Multiple explicit parent servers

Another variation on binding to the parent domain is that where a domain has more than one **/machines** subdirectory with a **serves** property indicating a parent domain server. This would be the case when a domain has clone servers. For example, at Rhino Aviation the root domain is served by processes on **super21**, **exec**, **mustang**, **sabre**, and **chaparral**; in the **/mktg** domain, **netinfod**s on both **super21** and **mustang** are listed as parent domain servers.

When **netinfod network** looks for its parent, it sends the ªAre you my mother?º message to each computer with the appropriate **serves** property. This techniqueÐtransmitting the same message to several different computersÐis called *manycasting*. As with broadcasting, the first reply to the binding request wins. (Manycasting is different from *multicasting*, a technique used to address a specific, defined group of computers; see Comer 1991.)

Incidentally, to optimize performance **netinfod** always tries to bind to a local parent if there is one before binding to a remote parent. That is, if the local computer is listed as serving the parent, **netinfod** binds to the parent domain server on the local computer rather than trying to bind to a parent on another computer.

An example of locating the parent domain

Here's what happens when a computer at Rhino Aviation starts up. **ranger** is host to only a server of its local domain. It has the defaults for the **serves** properties and **ip_address** properties set in **/machines**. Both its Internet address and host name are configured automatically.

Packet	Source	Destination	Protocol	Message
1	ranger	broadcast	portmap RPC	Indirect call: nibind(192.42.172.34, local, network).

2	cadet	ranger	nibind reply	I'm your mother; my nibindd is on UDP port 644.
3	exec	ranger	nibind reply	I'm your mother; my nibindd is on UDP port 706.
4	ranger	broadcast	ARP request	What's Ethernet address for Internet address 192.42.172.4?
5	exec	ranger	ARP reply	Ethernet address for IP 192.42.172.4 is 0:aa:0:18:83:2a.
6	ranger	exec	ICMP	Destination port unreachable.

Figure 1: *Simple binding of netinfod local to a parent*

Figure 1 shows the messages exchanged when **ranger**'s **netinfod local** binds to its parent domain server. First, **ranger** broadcasts a message to all the computers on the local network looking for a parent NetInfo server for the **local** domain (packet 1). In packets 2 and 3, two computers answer: **cadet** and **exec**. Since **cadet** was the first computer to answer, its **netinfod network** becomes the parent domain server to **ranger**'s **netinfod local**.

When **exec** and **cadet** reply to the binding request, each sends a message to the same port on **ranger**. Once **ranger** receives the first response in packet 2, it closes its reply port. When subsequent replies arrive, **ranger** sends an ªerrorº message (ICMP Destination port unreachable) back, as shown in packet 6. This message is ignored by **exec**'s **nibindd** at this stage of operation in the NetInfo binding protocol. To send the ICMP message to **exec**, **ranger** needs to know **exec**'s Ethernet address. This is why it sends the ARP message in packet 4, with subsequent ARP reply in packet 5.

BINDING THE REST OF THE HIERARCHY

Now, **netinfod** for the local domain has found a server for its parent domainÐit's *bound* to its parent server. But, it has exchanged no messages with domains other than **local**'s parent, so the rest of the hierarchy remains, in a sense, unknown. Indeed, though there's a binding to **local**'s parent, there's no *connection* yet from a

NetInfo client process to that parent. Only the existence of the parent and the Internet address of its computer have been determined, not even how to contact the parent.

However, each **netinfod** in its turn binds to a parent domain. So, collectively the hierarchy is known. This information is then used when a client needs to *connect* to one of the higher-level domains to obtain information.

CONNECTING TO A DOMAIN

A NetInfo server locates a server for its parent domain by binding. A NetInfo client then *connects* to a server for a domain. When a client connects to a second-level or higher domain, it inherits the bindings of the **netinfod**s.

Figure 2 shows what happens when a client requests a piece of information that isn't available from the local domain. In this example, the account information for the user **eng** has been requested. This example assumes the binding shown in Figure 1 has completed.

Packet	Source	Destination	Protocol	Message
1	ranger	broadcast	ARP request	What is the Ethernet address for Internet address 192.42.172.6?
2	cadet	ranger	ARP reply	Ethernet address for IP 192.42.172.6 is 0:0:f:0:59:82.
3	ranger	cadet	Portmap RPC	What is the UDP port for the netinfobind program, version 1?
4	cadet	ranger	RPC reply	Port 722.
5	ranger	cadet	netinfobind RPC	getregister("network")
6	cadet	ranger	RPC reply	UDP port 724, TCP port 726.
7	ranger	cadet	TCP	Window size 4096?
8	cadet	ranger	TCP	OK; window size 4096?
9	ranger	cadet	TCP	Window size 4096 OK.
10	ranger	cadet	netinfo RPC	root_directory()

11	cadet	ranger	RPC reply	root directory's handle is {0,36} (directory 0, instance 36).
12	ranger	cadet	netinfo RPC	lookup({0, 36}, "name", "machines")
13	cadet	ranger	RPC reply	**/machines** is [1,{0,36}] (directory 1 in {0,36}).
14	ranger	cadet	netinfo RPC	list(**/machines**, "serves")
15	cadet	ranger	RPC reply	Each directory's ID number and serves property values.
16	ranger	cadet	netinfo RPC	listprops(**/machines/cadet**)
17	cadet	ranger	RPC reply	The properties in **/machines/cadet**.
18	ranger	cadet	netinfo RPC	readprop(**/machines/cadet**, **ip_address**)
19	cadet	ranger	RPC reply	192.42.172.6
20	ranger	cadet	netinfo RPC	listprops(**/machines/exec**)
21	cadet	ranger	RPC reply	The properties in **/machines/exec**.
22	ranger	cadet	netinfo RPC	readprop(**/machines/exec**, **ip_address**)
23	cadet	ranger	RPC reply	192.42.172.4
24	ranger	cadet	netinfo RPC	root_directory()
25	cadet	ranger	RPC reply	{0,36}
26	ranger	cadet	netinfo RPC	lookup({0,36}, "name", "users")
27	cadet	ranger	RPC reply	[158, {0,36}]
28	ranger	cadet	netinfo RPC	lookup_read(**/users**, "name", "eng")
29	cadet	ranger	RPC reply	Contents of **/users/eng**.
30	ranger	cadet	TCP	Receipt acknowledged.

Figure 2: *Fetching information from a parent domain*

First, the client process needs to contact the server for the second-level domain. Packets 1 through 4 obtain the address information for the **nibindd** running on **cadet**Ðrecall that **cadet** is where the chosen **netinfod network** is running. Packets 5 and 6 get the addresses of the **netinfod network** running on **cadet**. Packets 7 through 9 are TCP protocol overhead, setting up a TCP connection to **cadet**'s

netinfod network on TCP port 726.

The client process now has a connection established to the chosen server.

After establishing the connection, the client obtains a list of the known servers of the domain, in case the chosen server for the domain fails and a new server must be contacted. Packets 10 through 13 obtain the ªhandleº for the **/machines** directoryÐa handle is how directories are referenced within NetInfo. First, it gets the handle for the root directory, then the handle for the directory **machines** within that root directory. Next, in packets 14 and 15 the client acquires the values for the **serves** properties in each subdirectory of **/machines**. From these, the client can determine which computers provide services for a parent **netinfod**, by finding **serves** properties with a dot (**.**) in the domain portion of the value. Then, in packets 16 through 23, the client on **ranger** requests the Internet address of each of the computers with a server for the parent domain. In this case, those servers are running on machines with Internet addresses 192.42.172.6 (**cadet**) and 192.42.172.4 (**exec**).

Finally, packets 24 through 29 read the properties in **/users/eng**. Again, the client requests the handle for the root directory. It then gets the handle for **/users**, and finally gets the actual data. Packet 30 is TCP overhead acknowledging receipt of the answer. Other TCP acknowledgments were piggybacked on top of request or reply packets (see Comer 1991).

Notice that the actual data lookup required only the last three packets, which include one packet of TCP overhead. Packets 1±23 establish the connection, and 24±27 get the handle for **/users**. In the case of **lookupd**, the primary NetInfo client, the connection is usually already established, and the handles for the root and the **/users** directories are usually cached. In the example in Figure 2, when a client on **ranger** needs to contact **local**'s parent, it inherits the binding between **netinfod local** on **ranger** and the **netinfod network** running on **cadet**.

Contacting the third-level domain

When a client requests information that's in neither the first-level nor the second-level domain, the third level is contacted. Figure 3 shows the sequence of

messages that get the account information for the user **sandy**, this time from the root domain.

Packet	Source	Destination	Protocol	Message
1	ranger	cadet	netinfo RPC	remote_parent
2	cadet	ranger	RPC reply	["Rhino", 192.42.172.4]
3	ranger	exec	Portmap RPC	getport(UDP, netinfobind, 1)
4	exec	ranger	RPC reply	Port 658.
5	ranger	exec	netinfobind RPC	getregister("Rhino")
6	exec	ranger	RPC reply	UDP port 660, TCP port 662.
7	ranger	exec	TCP	Window size 4096?
8	exec	ranger	TCP	OK; window size 4096?
9	ranger	exec	TCP	Window size 4096 OK.
10	ranger	exec	netinfo RPC	root_directory()
11	exec	ranger	RPC reply	{0,69}
12	ranger	exec	netinfo RPC	lookup({0,69}, "name", "machines")
13	exec	ranger	RPC reply	[3,{0,69}]
14	ranger	exec	netinfo RPC	list(**/machines**, "serves")
15	exec	ranger	RPC reply	Each directory's ID number and **serves** property values.
16	ranger	exec	netinfo RPC	listprops(**/machines/super21**)
17	exec	ranger	RPC reply	The properties in **/machines/super21**.
18	ranger	exec	netinfo RPC	readprop(**/machines/super21**, **ip_address**)
19	exec	ranger	RPC reply	192.42.172.2
20	ranger	exec	netinfo RPC	listprops(**/machines/exec**)
21	exec	ranger	RPC reply	The properties in **/machines/exec**.
22	ranger	exec	netinfo RPC	readprop(**/machines/exec**, **ip_address**)
23	exec	ranger	RPC reply	192.42.172.4
24	ranger	exec	netinfo RPC	listprops(**/machines/mustang**)

25	exec	ranger	RPC reply	The properties in **/machines/mustang**.
26	ranger	exec	netinfo RPC	readprop(**/machines/mustang**, **ip_address**)
27	exec	ranger	RPC reply	192.42.172.5
28	ranger	exec	netinfo RPC	listprops(**/machines/sabre**)
29	exec	ranger	RPC reply	The properties in **/machines/sabre**.
30	ranger	exec	netinfo RPC	readprop(**/machines/sabre**, **ip_address**)
31	exec	ranger	RPC reply	192.42.172.66
32	ranger	exec	netinfo RPC	listprops(**/machines/chaparral**)
33	exec	ranger	RPC reply	The properties in **/machines/chaparral**.
34	ranger	exec	netinfo RPC	readprop(**/machines/chaparral**, **ip_address**)
35	exec	ranger	RPC reply	192.42.172.98
36	ranger	exec	netinfo RPC	root_directory()
37	exec	ranger	RPC reply	{0,69}
38	ranger	exec	netinfo RPC	lookup({0,69}, "name", "users")
39	exec	ranger	RPC reply	[308, {0,69}]
40	ranger	exec	netinfo RPC	lookup_read(**/users**, "name", "sandy")
41	exec	ranger	RPC reply	Contents of **/users/sandy**.
42	ranger	exec	TCP	Receipt acknowledged.

Figure 3: *Fetching information from a parent's parent*

The sequence actually begins with a failure to find the requested information in the second-level domain. If the client already has a connection to the second-level domain, the exchange includes an analog to packet 29 from Figure 2, requesting the properties in **/users/sandy**, and a reply stating the information wasn't found. Operations continue with the conversation as shown in Figure 3.

Figure 3 is very similar to Figure 2. Indeed, packets 3 through 27 are exactly the initial connection process, although because the connection is to a different domain

and different servers, the packets themselves are slightly different. Packets 28 through 34 repeat the acquisition of the properties in the **/users** subdirectoryÐ**/users/sandy** in this case.

The new packets here are 1 and 2. These packets are the *remote parent* request and reply. When a NetInfo client communicates with a domain higher than the local domain's parent, it inherits the parent-to-grandparent binding.

If the NetInfo client on **ranger** needs to contact **cadet/network**'s parent, it inherits the binding between **cadet/network** and **netinfod Rhino**, to which **cadet/network** is bound. It obtains this information using the NetInfo Remote Parent, or **NI_RPARENT**, request. **NI_RPARENT** requests the tag and the Internet address of the server of the parent of the receiving **netinfod**.

In this example, **local**'s parent, **cadet/network**, is bound to the **netinfod Rhino** on **exec** (Internet address 192.42.172.4), **exec/Rhino**.

If the client had already obtained the binding to the third level, and if that domain's root and **/users** directories had already been referenced and cached, then the sequence would have been much simpler, involving only a failed attempt to read **/users/sandy** from the second-level domain and a successful attempt to read it from the third-level domain.

BINDING AND CONNECTING FAILURES

Two problems that can come up in regard to binding and connecting are that there might be no servers available for a particular domain, or that a server to which a client had been connected could go down.

Failure binding netinfod local to a parent

When **netinfod local** starts, it tries to contact a parent domain server if it believes one exists and the network is enabled. If the request for a parent times out, the child displays a message on its console:

```
Still searching for parent network
```

```
administration (NetInfo) server.
Please wait, or press 'c' to continue without network user accounts.

See your system administrator if you
need help.
```

The timeout period prior to printing this warning is 30 seconds. This message is displayed only when the initial binding of **netinfod local** fails. It's *not* displayed for other domains binding to their parents, nor is it displayed during rebinding or reconnecting.

netinfod local continues to look for a parent domain server after displaying this message. It continues searching until either it finds a parent, you press **c**, or you shut down the computer.

It's because of this special treatment of the database tagged **local** that this tag is reserved for the local NetInfo domain and the first-level domain should be tagged **local**.

Incidentally, you can customize the searching message by modifying the file **NetInfo.strings** in **/usr/lib/NextStep/Resources/English.lproj**.

Failure binding to a parent's parent

Only **netinfod local** waits for a parent server to respond before continuing. Back in the beginning of this article, we said that servers other than that for **local** only bind to their parent servers when a client requests this binding information. So, what happens if when a client asks a server for its parent (using **NI_RPARENT**), the server hasn't yet bound to a parent, and no parent server responds?

A message is sent to the system log (*syslog*) noting that a NetInfo timeout occurred. As with **local** binding to its parent, the **netinfod** hereÐ**netinfod network**, for exampleÐcontinues sending out **NIBIND_BIND** requests, looking for a parent. ªThe Tough Stuffº describes these messages and their causes in more detail.

A higher-level binding problem is likely to manifest itself first when a third- or higher-level domain is accessed during a request for data. This might happen when a

computer starting up tries to mount a remote file system. Again, unlike binding of the first-level domain to the second-level domain, no message noting the binding failure is displayed.

Failures after initial binding

If, after a client connects to a server, a request to that server times out, you might see messages like this:

```
netinfo timeout, sleeping
```

If the condition persists, you'll see this message:

```
netinfo failure, sleeping
```

Once this message is displayed on syslog, which is normally sent to the console, the NetInfo client attempts to find a server for the domain again.

Remember the packets sent and received when a client first obtains data from a NetInfo server in Figure 2? Some of the communication included the client getting a list of all the servers for the contacted domain. Here's where that list of servers is used. The client attempts to reconnect to the domain by sending a message to each of the machines in that list. Just like when a domain binds to its parent during or after start-up, this request is in the form of a manycast.

The **NIBIND_GETREGISTER** message is sent, just as in the initial connection conversation. And, just like the initial binding or connection, the first reply to the manycast that's received is the winner, and that server is chosen for the connection.

Note: The document references in this and other articles in this issue refer to the books and articles listed in ªNEXTSTEP Networking References.º

FINDING SERVER PROCESSES

Processes communicate over TCP and UDP using ports, which are abstract source and destination points (Comer 1991). How does a client process find the port for a NetInfo daemon? It must either already know the port number, or contact a process that finds port numbers.

Well-known ports
Ports are numbered, and some port numbers are well-knownÐthey're reserved for use by certain services or particular functions. As an analog, the telephone numbers for directory assistance throughout the U.S.Ð411 for local information, 555-1212 in all area codes for long distance informationÐare reserved. Dialing one of them always reaches the directory assistance service. In networking, ports with numbers less than 256 are reserved and can become well-known.

One well-known port, port 111 for both TCP and UDP, allows a client to contact a service called the portmapper. The portmapper is like directory assistance, only instead of matching names to telephone numbers, it translates SunRPC program numbers into port numbers. In portmapper parlance, this is the operation PMAPPROC_GETPORT. The portmapper allows clients to find many SunRPC programs without having to reserve a well-known port for each one.

NetInfo and the portmapper
The NetInfo binding daemon, **nibindd**, checks in with the portmapper when it starts up. To determine the port for contacting nibindd, a client first sends a message to the portmapper on port 111 requesting the port number for **nibindd**. In fact, if you examine the information from the portmapper using the UNIX **rpcinfo** program, you'll see that **netinfobind**, SunRPC program number 200100001, can be found on different ports on different computers. (See the UNIX manual pages to find out about **rpcinfo**.)

When a NetInfo clientÐwhether it's a **netinfod** or some other client, like **lookupd**Ðbegins to communicate with a NetInfo server, it has to determine that server's port number. It does this by asking the **nibindd** running on that server's computer for the port number of the **netinfod** serving that tag. It finds the **nibindd**'s port number by asking the portmapper for it.

An example
For example, to see which daemons are running on a host you would run the **nidomain -l** command:

```
mite-23% nidomain -l
tag=local udp=660 tcp=664
tag=network udp=664 tcp=666
```

nidomain -l communicates with **nibindd** to determine which **netinfod** daemons are running on a

computer. In this instance, **mite** has two **netinfod**s running, or more precisely two that have registered with **nibindd**. The first, serving a database tagged **local**, can be contacted over UDP on UDP port number 660, and over TCP on TCP port number 664. The second, serving a database tagged **network**, can be reached on UDP port 664 and TCP port 666.

Communicating with many portmappers
A child **netinfod** binding to a parent might communicate with scores or hundreds of computers, depending on how many parent servers are listed and whether it's using a broadcast address. How does **netinfod** keep track of all those different port numbers from the portmappers? Sure, one broadcast message can be sent to all the portmappers: portmapper uses a single well-known port. But isn't there an awful lot of bookkeeping needed to track all the answers from all the portmappers?

It turns out that the portmapper can do a little magic. In addition to translating SunRPC program numbers into port numbers, it can invoke a SunRPC indirectly. This operation is PMAPPROC_CALLIT.

The child **netinfod**, when trying to bind, sends the portmapper a request to run the NIBIND_BIND procedure of the **netinfobind** SunRPC program; portmapper then returns the results to the child. The lack of a "no" answer in the NetInfo binding protocol applies to this method of invocation, too.Đ*AMM*

TEMPORARY NETINFO DIRECTORIES

You know about the directories in **/etc/netinfo** with an **.nidb** suffix, which contain NetInfo databases. Sometimes though, you might see directories with two other suffixes: **.temp** and **.move**.

A **.temp** directory holds a temporary database that's being loaded from the master server's database, and will become the actual database. A **.move** directory holds a **.nidb** directory that's moved aside during conversion of a **.temp** directory to a **.nidb** directory. After this conversion is complete, the **.move** directory is deleted. The **.move** directory is created only after the **.temp** directory is completely filled.

These directories are examined during NetInfo startup. If **netinfod** finds a **.temp** directory without a **.move** directory, then it knows the **.temp** directory was incomplete, and it discards the directory since it will get a new database from the master shortly anyway. If **netinfod** finds a **.temp** directory

and a **.move** directory, it knows the **.temp** directory is complete and renames it to the **.nidb** directory; it deletes the **.move** directory. If it finds a **.move** directory without a **.temp** directory, it discards the directory.

You can find **.move** and .temp directories only while downloading a new database from the master server. If a download is interrupted, **netinfod** cleans up the mess.*ĐAMM*