NX Client Developer's Guide

version 1.0

updated 03/20/2005

Table of Contents

Introduction

This guide will provide all the documentation necessary to develop NX client applications. Specifically, the guide will cover:

- An overview of the NX technology and history of the open source clients
- How to integrate NX into your applications using the nxrun client
- How to use the nxdriver c++ library to create custom clients
- A guide to using moznx to create browser-based applications
- An overview of all the components in the nxcompsh/nxrun libraries
- Detailed build instructions for each platform

Overview of NX technology

Before digging into how to develop applications using the NX client tools, it is important to understand what NX is and what the related products are. The best documentation can be found at nomachine.com, so this overview will be brief.

The core NX technology is a set of custom libraries that compress X-Windows sessions and make them perform well over slow connections. These libraries are used in the NX applications, which behave like Citrix or VNC. However, how they work is a little different.

Normally in X-Windows applications, there is a server on the end user's machine and a client that the user wants to run. The client can be on the user's machine or on another machine. When the user invokes the client application, it runs on its machine, but it sends all the ui messages to the server which provides the screen, pointer, and keyboard interfaces to the user. With applications like Citrix and VNC, these messages are trapped some way and converted to a different protocol and sent to a custom client to render for the user.

With NX technology, NX components are inserted between the server and the client. There is one component at each end. While the server and client think they are talking directly to each other, they are really talking through the NX

libraries. These libraries optimize the X-Windows messages using caching, compression, and nesting so the traffic between the two is lighter and faster. Unlike other technologies, NX doesn't translate the messages to a proprietary protocol. It uses the native X protocol and is transparent to the X server. This is why the NX components are referred to as proxies. A web proxy works in a similar fashion - the web servers and browsers talk the same way, but the proxy adds additional caching and security by stepping into the middle of the connection.

This core proxy technology is released by Nomachine as open source libraries. The components have names like nxagent, nxcomp, and nxproxy. These libraries are supported by utility applications like nxssh and nxwin. See the documentation on nomachine.com for more details about these components.

So, how does a user tell its X environment to run an application through the proxies instead of normally? Well, it could be done manually through a series of steps on the user's client machine and the server node. But Nomachine developed NX client and NX server to make it easier.

The nxclient calls the nxserver. It authenticates and requests a session. The nxserver launches an instance of the proxy agent on the server and tells the nxclient how to access it. The nxclient then starts an instance of the proxy on the client machine. The two proxies talk to each other and the session is established. Then the desired application is launched by nxserver and run through the proxies. This communication protocol has been reverse engineered with a lot of help from nomachine. While some open source clients invoke the proxies manually, most use this protocol to talk to a commercial server, or to the freenx server.

NXCompsh and NXDriver

Because the nxserver protocol is complicated and changes over time, it is better that the open source community use a common implementation of the protocol in their applications. This implementation is nxcompsh and nxdriver. They were originally released under GPL by nomachine as part of 1.3. The client consisted of a low level communication library called nxcompsh and a command line client called nxrun. The client was going to be used as the core

of the Nomachine 2.0 NX Client. But it was buggy to start with and was never maintained. I picked it up and started working with it because I wanted to create a browser plug-in client and I really didn't want to figure out how the protocol really worked. I tore out the command line interface and turned it into a high-level API called NXDriver. I also updated all the code to support the 1.4.0 servers and fixed some bugs. So what happened to NXRun? Well, using NXDriver it was trivial to implement a command line interface. So nxrun has been recreated and it represents the easiest way for developers to create nx-enabled applications.

NXRun

NXRun Overview

NXRun is designed to be either a standalone client or to be embedded in scripts or applications. Because it is just a wrapper around nxdriver, nxrun is a great way to understand the functionality of the library. All of the functionality described in this section is also available through nxdriver.

NXRun takes as input a valid nx client configuration file and some command line parameters.

The easiest way to use nxrun is to start with a configuration file created by the nomachine client.

now you can invoke nxrun from the command line:

./nxrun test.conf -p onion

nxrun does not support the way passwords are saved in the nx client configuration file, so you have to use the -p parameter to define your password.

There are other parameter as well. The table below lists them all.

-u : prompt for user

-p: prompt for password
-i: interactive. Prompt for user or password if they are not set
-q : quiet. Do not display messages on console
-l: enable logging. See details below

Custom Configuration Attributes

In addition to the standard configuration file, nxrun also supports a few custom configuration items. These are listed below:

Group: Login
Password - Password in md5 format
ClearPassword - Password in clear text
KeyFile - path to an ssh keyfile (overrides default of NXDIR/share/client.id_dsa.key)

Getting Feedback

NXRun provides feedback on its processing over STDOUT and STDERR. The format and content is outlined below;

STDOUT
I> <message>. An informational messgae
S> <session id>. The session id assigned to the session
C> Complete. NX client initiated successfully.
STDERR
E> <message>. An error occurred. This is the exception message.

In the case of an error, you get an informational message saying there is an error on STDOUT and the detailed error message on STDERR.

You can suppress these messages by using the -q parameter.

Logging

There is a logging facility in nxrun. You enable it from the command line by using the -l flag. For example, you can log errors to a file with the following:

./nxrun test.conf -l fe

There are two modes for logging. One is logging to the console ('c'). This logs info and debug messages to STDOUT and errors to STDERR. The other is logging to a file ('f'). This option will log all messages to a file called nxcompsh.log in the current directory.

There are three types of messages to be logged. They are info, debug, and errors ('i', 'd', and 'e' respectively). You decide which ones to log by setting the appropriate flags. to log all three, just do:

./nxrun test.conf -l ide

NXDriver

NXDriver Overview

Once you understand how NXRun works, you will find it very easy to use nxdriver. We invoked nxrun from the command line using the following:

./nxrun test.conf -p onion

To do the same from a c++ progam using nxdriver, we do the following:

```
#include "NXDriver.h"

NXDriver dr;
dr.SetConfigFile("test.conf");
dr.SetClearPassword("onion");
dr.Run();
```

Simple! How about the other options we used? They are all in NXDriver as well. The table below shos the equivalent methods for each of the command line parameters:

Getting Feedback

While NXDriver is easy to invoke, you are probably also going to want to keep track of what it is doing so you can provide feedback to your users, handle errors, etc. There are two ways of doing this with NXDriver.

The Callback Interface

NXDriver has a very simple callback interface that can be implemented by any c++ class. It has a single method that is invoked whenever nxdriver has information to share.

The method is Callback(int message, void *ptr);

The message types that are supported are in CallbackMessageTypes.h.

The messages are defined below:

CB_MSG_STATUS. Informational message. ptr is a const char * to the informational message.

CB_MSG_SESSION. Once a session id has been assigned by the server this message is sent. ptr is a const char * to the session id. The session id is in the form hostname-display-unique id

CB_MSG_COMPLETE - This message is sent when NXDriver completes successfully. ptr is null.

CB_MSG_ERROR - This message is sent when an error occurs. ptr is a const char * to the error message.

For a simple example of a class implementing the callback interface, see ConsoleCallback in nxrun.

Deriving NXDriver

The second way to get feedback to your application from NXDriver is to derive your application from it. There are two methods that can be used to get messages and errors from NXDriver. They are:

void NXDriver::SetConnectionMessage(string message)
void NXDriver::SetConnectionError(string error)

Logging in NXDriver/NXCompsh

NXDriver and NXCompsh share a logging interface. It is nxcompsh/Logger.h.

The logging options are the same as those presented in nxrun. There are a set of macros you can use to log events.

The macros serve two purposes. One is to keep the logging interface simple. The other is to allow you to compile with or without logging.

To enable logging, you need to define NX_ENABLE_LOGGING in your compile.

Then you can use the commands below:

NX_LOG_SETFLAGS(flags) - set the logging level. Flags is a string containing some combination of c,f,i,d, and e. See the description under nxrun for more details
NX_LOG_LOGINFO(message) - Log an informational message
NX_LOG_LOGDEBUG(message) - log a debug message
NX_LOG_LOGERROR(message) - log an error message

Tips for using the Logger.h macros:

If your debug logic is complicated and you don't want it compiled in unless debug is on, then use the NX_ENABLE_LOGGING macro. For example:

```
#ifndef NX_ENABLE_LOGGING
    stringstream ss;
    ss << "log this: " << foo() << "from " << bar();
    NX_LOG_LOGDEBUG(ss.str);
#endif
```

If you do compile with logging enabled, remember that these macros can be compiled out. For example, the following code would be bad:

```
int i=0;
while (i < 10)
  NX_LOG_LOGDEBUG("element:" + e[i++]);
```

Not only would the while statement have no body when the macro is compiled out, but i will never be incremented.

Moznx

Moznx Overview

You can use moznx to embed nx services in a browser application. This is great for creating a portal to allow acces to servers or applications. Also/ browser clients can be more lightweight and cross-platform.

Moznx is a mozilla plugin. That means it will work with Mozilla or Firefox browsers. It is native code, so it must be installed for your particular platform. Currently, moznx is available for Windows and Linux.

To embed moznx in a web page, you use the embed tag in html. The src attribute will point to the nx configuration file. This can be a local file, a static file on your web server, or a file generated dynamically from a database or directory.

Getting feedback from moznx

Moznx prints messages to the plugin window, but your application may want feedback as well. There is a browser callback mechanism in moznx. It is intended to provide callbacks to a web server but could be adapted to client-side callbacks as well.

To use the callbacks, you need to set the postback_target attribute. This attribute will point to the frame you want the callback messages sent to. I use a hidden iFrame on my page. Then you need to set the postback_session attribute. This tells moznx what url to call (with a get request) when an event occurs. The events are either that a session was successfully established or that an error occurred. It also gets called when the session id is assigned.

Because moznx is based on NXDriver, it has the same features as nxrun. A list of the custom attributes for moznx is below:

postback_session -If you provide this parameter, this url will be called with a GET to give you feedback on your session. The data is appended to your url. There are 3 scenarios where this is called:
op=postsessionid&session_id={the session id}. Once a session id is established it will be sent to your url
op=postconnected. The url will be called when the negotiation is completed successfully.
postback_target - This is the name of the frame to send the callback message to.
restore_session_id - If this session id is set/ moznx will restore that session instead of starting a new one
session_name - The name to give to the session. Important. The nomachine server will use the session name to determine if you are requesting a suspended session and will restore a matching session even if you do not explicitly request it.
log_flags - if you want to enable logging, you can set the logging flags (c,f,i,d,e).

Installation

The mozilla plugin can be installed using the xpi installer for mozilla. This technology uses a simple script to install the plugin on the client machine. The

xpi files I have created include both the plugin and the latest version of the nomachine executables. The windows install alslo includes the pthread dll. Please note that on the Windows platform, the moznx install may overwrite some of your programs if you have the nomachine client installed.

Paths

Moznx needs to define paths for three components. One is the plugin directory for the browser. The second is the location of the core X binaries. The third is the location of the temp and session files.

On both platforms, it is possible to control where the binaries are stored. First, it will look for the environment variable NXDIR. If this is not set, it will use the default location for each platform. On Linux, it goes one step further. If the binaries are not int the default location, it will look to where moznx installs the binaries. This is to make it easier to package moznx to share binaries with other clients.

On Windows, moznx installs the plugin to the mozilla plugin directory. It installs the core nx components to c:\Program Files\NX Client for Windows\, which is the default location for the commercial client. Currently, the temp files are stored in c:\.nx.

On Linux, it is more likely that end users will be set up properly, meaning that they will not be allowed to install shared programs. The default moznx install takes this into account. The plugin is installed to the user's browser plugin directory (for example ~/.mozilla/plugins). The core nx components are stored in ~/.nx. This is different then the standard location of /usr/NX. The temporary and session files are stored in the default location of ~/.nx/.

Changing the install package is very straightforward. The xpi archive is just a zip file and can be unzipped with any standard zip utility. The install.js file in the root of the archive is a standard javascript file with the install commands in it. For more information see the xpi project on mozilla.org.

The NXCompsh/NXDriver Implementation

To be added

Appendix A - Installing the NX Client Tools

To be added

Appendix B - Building The Client Tools

- Linux
  - nxrun
    - Download the client from cvs
      - For each directory do the following:
        - ./configure
        - make
      - Build in the following order:
        - nxcompsh
        - nxdriver
        - nxrun
  - moznx
    - Download the samonkey source from mozilla
    - configure and build all of mozilla
    - download the client from cvs into the following directory
      - mozilla/modules/plugin/tools/sdk/samples/
    - For each directory do the following:
      - ./configure
      - make
    - Build in the following order
      - nxcompsh
      - nxdriver
      - moznx
- Windows
  - Windows is a little more complicated
  - You must use Microsoft Visual C++ 6.0
  - Download the gecko sdk from the mozilla site
  - download the client from cvs into the samples directory
  - download expat into the nxc directory
  - download pthreads for win32 into the nxc directory

open the nxc workspace and build

Appendix C - The Protocol - Sample Session

1. Connect to the server using nxssh

nxssh -nx -i /usr/NX/share/client.id_dsa.key nx@<host address>

If you are using encrypted session:

nxssh -nx -i /usr/NX/share/client.id_dsa.key nx@<host address> -B

for Windows client, you have to include the -v switch for encrypted sessions to work.

nxssh -nx -i /usr/NX/share/client.id_dsa.key nx@<host address> -v -B

You will get the following response:

NX> 203 NXSSH running with pid <some pid>
NX> 285 Enabling check on switch command
NX> 200 Connected to address: <address> on port: <port>
NX> 202 Authenticating user: nx
NX> 208 Using auth method: publickey
HELLO NXSERVER - Version 1.4.0-02 OS_(GPL)
NX> 105

3. NX> 105 is kind of like a shell prompt. Now you respond with the client version

type: hello NXCLIENT - Version 1.4.0

You will get the following response:

NX> 105 hello NXCLIENT - Version 1.4.0
NX> 134 Accepted protocol: 1.4.0

NX> 105

4. I think the production client then sends the following:

SET SHELL_MODE SHELL

response:
NX> 105 SET SHELL_MODE SHELL
NX> 105

SET AUTH_MODE PASSWORD
NX> 105 SET AUTH_MODE PASSWORD
NX> 105

5. Then you send the login command

type: login

response:
NX> 105 login
NX> 101 User:

type: <username>

repsonse:

NX> 102 Password:

type: <your password>

If you type <enter> instead, you will get the following from the commercial server (but not freenx)

NX> 109 MD5 Password:

type: <md5 of usernamepassword>

You can get this password value by using the nxpassgen utility I have for moznx

response:
NX> 103 Welcome to: <host> user: <username>
NX> 105

6. Now you can request a session

type: startsession --session="<session>" --type="unix-kde" --
cache="8M" --images="32M" --
cookie="6726ad07a80d73c69a74c5f341b52a68" --link="adsl" --
render="1" --encryption="0" --backingstore="when_requested" --
imagecompressionmethod="2" --geometry="1024x768+188+118" --
keyboard="defkeymap" --kbtype="pc102/defkeymap" --media="0" --
agent_server="" --agent_user="" --agent_password="" --
screeninfo="1024x768x16+render"

For encrypted session, send --encryption="1"

note: I have always had trouble getting this to work and have to use '&' as a delimeter instead of ' --'. It seems this issue is solved if you SET SHELL_MODE and SET AUTH_MODE as described above. I have not confirmed yet.

response:

NX> 105 startsession --session="<session>" --type="unix-kde" --
cache="8M" --images="32M" --
cookie="6726ad07a80d73c69a74c5f341b52a68" --link="adsl" --
render="1" --encryption="0" --backingstore="when_requested" --
imagecompressionmethod="2" --geometry="1024x768+188+118" --
keyboard="defkeymap" --kbtype="pc102/defkeymap" --media="0" --
agent_server="" --agent_user="" --agent_password="" --
screeninfo="1024x768x16+render"

you can also just type startsession<enter> then the response will be

NX> 106 Parameters:

Then you type all the parameters

You can replace startsession with restoresession if you want to restore an

existing session. You add the additional attribute --id="<session id you want to restore>".

A good explanation of restoring sessions is here:
http://www.nomachine.com/developers/archives/nxdevelopers/0323.php

7. Now the server sends back all of its parameters followed by a 105

NX> 700 Session id: <hostname>-1058-CA3511103B37ADB2ABDAAF3EB510E99D
NX> 705 Session display: 1058
NX> 703 Session type: unix-kde
NX> 701 Proxy cookie: A4BFD3EAE09B28A0EB0399A3EFD26392
NX> 702 Proxy IP: 127.0.0.1
NX> 706 Agent cookie: 6fff2cd4222776acd605d42fbb4bdfb5
NX> 704 Session cache: unix-kde
NX> 707 SSL tunneling: 0
NX> 710 Session status: running
NX> 105

For encrypted sessions, NX> 707 SSL tunneling: 1

8. Now in another session invoke nxproxy with the proper parameters on the command line and in the options file.

nxproxy -S options=<path to options file>/options:<Session display>

for example above: nxproxy -S options=/.nx/S-hostname-1058-CA3511103B37ADB2ABDAAF3EB510E99D/options:1058

Then, in the options file:

nx,session=<sesname>,cookie=A4BFD3EAE09B28A0EB0399A3EFD26392,root=/.nx,id=hostname-1058-CA3511103B37ADB2ABDAAF3EB510E99D,listen=33057:1058

listen=<port:display> is only needed for encrypted sessions. Also, all these parameters can be sent on the command line instead of the options file.

For the listen=<port:display>, I always just hardcode a port number. I am not sure where the commercial client gets the port number. I have asked but not gotten a response.

If the session is not encrypted, you say connect=<address:port> instead. Address is the address of the NX Server. Port is the proxy port.

9. Now back to our NXSSH session.

type 'bye'

Response:

999> Bye

10. For encrypted sessions, now enter the switch command

type: NX> 299 Switching connection to 127.0.0.1:33507 cookie: A4BFD3EAE09B28A0EB0399A3EFD26392

Appendix D - The Protocol - Reference

This section is in csv format.

Server responses,,
,HELLO NXSERVER - Version 1.4.0-02 OS_(GPL) ,Hello from server. Comes after ssh authentication. Includes server version number
100,NXSERVER - Version <version> <license>,version
101,user:,enter the nx user name. (response to login comand)
102,password:,enter the nx password in plain text (response to login command)
103,Welcome to: <host> user: <username>,welcome indicates nx user and password have been accepted
105,,This is kind of like a shell prompt. It means the server is expecting a command from the client. It may also be sent to echo a command back to the client
106,parameters:,parameters for the session (response to startsession or

restoresession command)
109,MD5 password,enter the nx password in md5 <usernamepassword). Commercial only
110,NX Server is <status>,status (running|stopped)
113,Changing password of user <user>,change password
114,Password of user <user> changed,password changed
122,Service started,nxserver started
123,Service stopped,nxserver stopped
127,Available sessions:,lists out all the sessions meeting the parameters. Response to listsession command from the client
134,Accepted protocol: 1.4.0,Indicate that the client protocol has been matched and accepted (response to hello command from client)
148,Server capacity: not reached for user: <user>,Info - server capacity was checked and has not been reached
200,Connected to address: <address> on port: <port>,nxssh connected
202,Authenticating user: nx,authenticating the ssh user
203,NXSSH running with pid <some pid>,Info - process id of nxssh session
208,208 Using auth method: <method>,Info - auth method for ssh session
285,Enabling check on switch command,Info - used for informational messages related to encryption/switch
404,ERROR: wrong password or login,response to login command with invalid credentials
500,ERROR: <error>,Server encountered an error
537,ERROR: passwords do not match,Password does not match confirm password on password change
700,Session id: <hostname>-<display>-<id>,Session id assigned by the server
701,Proxy cookie: <cookie>,Cookie assigned to the proxy by the server. Used to authenticate to the proxy
702,Proxy IP: <ip>,IP Address used to communicate to the client proxy
703,Session Type: <type>,Session type requested by the client
704,Session cache: <cache>,
705,Session display: <display>,"On the server, every nx instance is associated with a display"
706,Agent Cookie: <cookie>,Session cookie suggested by the client
707,SSL tunneling: (0|1),Flag whether to tunnel all traffic through ssh. Based on encryption parameter sent by the client,
710,Session status: <status>,good status is 'running',
999,Bye,end interactive session,
1002,Commit,,

1006,Session status: running,,

Client commands,,,
,hello NXCLIENT - Version 1.4.0,hello. Include the client version,NX> 105 hello NXClient - Version 1.4.0\nNX> 134 Accepted Protocol 1.4.0
,SET SHELL_MODE SHELL,set the shell mode (for backward compatibility),NX> 105 SET SHELL_MODE SHELL
,SET AUTH_MODE PASSWORD,,NX> 105 SET AUTH_MODE PASSWORD
,login,request nx login,NX> 105 login\nNX> 101 user:
,startsession,request to start a new session,if parameters are not included: NX> 106 parameters:. Otherwise NX> 105 startsession ? followed by server reporting its parameters (700 series)
,restoresession,request to restore an existing session,if parameters are not included: NX> 106 parameters:. Otherwise NX> 105 restoresession ? followed by server reporting its parameters (700 series).
,bye,Ends the interactive session,999> Bye
,"listsession --user=""<nx user>"" --status=""suspended,running"" --geometry=""<geometry>"" --type=""<session type>"" ",List the sessions that meet the criteria. Used to identify sessions to restore, NX> 127 Available sessions?
,NX> 299 Switching connection to <proxy ip>:<proxy display> cookie: <proxy cookie>,switch command for encrypted sessions. This tells the nxssh client to forward stdin & stdout to the nxproxy instance,

session parameters,,
,session,session name. Derived from conf file name in the commercial client.
,type,"unix-kde, unix-gnome, unix-application, windows, vnc. If unix-application, then application is required"
,cache,
,images,
,cookie,unique cookie for the session
,link,"modem, adsl, etc. Tells the bandwidth. It makes some decisions about compression and quality based on this"
,render,use rendering extension (?)
,encryption,0 means do not tunnel all traffic over ssh. 1 means do.
,backingstore,
,imagecompressionmethod,
,geometry,screen resolution
,keyboard,keyboard mappings. Default is defkeymap

,kbtype,keyboard type. Default is pc102/defkeymap
,media,forward sound (0|1)
,samba,map samba shares (0|1)
,agent_server,"vnc or rdp server, if applicable"
,agent_user,"rdp user, if applicable"
,agent_password,vnc or rdp password if applicable
,screeninfo,
,id,id of the session to restore

Proxy Parameters,,
,nxproxy -s,
,options=<path to options file>:<display>,path to an options file containing the remaining options
,nx,
,session=<session name>,session name. Derived from conf file name in the commercial client.
,root=<path>,where to keep session related files (?)
,id=<hostname>-<display>-<id>,
,cookie=<proxy cookie>,Cookie used for proxy authentication. Set by server and communicated to client with NX> 701
,listen=<port:display>,Port and display to listen for forwarded NXSSH connection. The port is the one defined in the NX> 299 command and the display is the same as display in the options parameter