# XPA: Public Access to Data and Algorithms

## Summary

This document is the Table of Contents for XPA.

## Description

The XPA messaging system provides seamless communication between many kinds of Unix programs, including X programs and Tcl/Tk programs. It also provides an easy way for users to communicate with XPA-enabled programs by executing XPA client commands in the shell or by utilizing such commands in scripts. Because XPA works both at the programming level and the shell level, it is a powerful tool for unifying any analysis environment: users and programmers have great flexibility in choosing the best level or levels at which to access XPA services, and client access can be extended or modified easily at any time.

A program becomes an XPA-enabled server by defining named points of public access through which data and commands can be exchanged with other client programs (and users). Using standard TCP sockets as a transport mechanism, XPA supports both single-point and broadcast messaging to and from these servers. It supports direct communication between clients and servers, or indirect communication via an intermediate message bus emulation program. Host-based access control is implemented, as is as the ability to communicate with XPA servers across a network.

XPA implements a layered interface that is designed to be useful both to software developers and to users. The interface consists of a library of XPA client and server routines for use in C/C++ programs and a suite of high-level user programs built on top of these libraries. Using the XPA library, access points can be added to Tcl/Tk programs, Xt programs, or to Unix programs that use the XPA event loop or any event loop based on select(). Client access subroutines can be added to any Tcl/Tk, Xt, or Unix program. Client access also is supported at the command line via a suite of high-level programs.

Choose from the following topics:


- Introduction to XPA . . . . . 1
- Access Point Names and Templates . . . . . 3
- Getting Common Information About Access Points . . . . . 5
- Communication Methods . . . . . 8
- Communication Between Hosts . . . . . 10
- Distinguishing Users . . . . . 14
- XPA User Programs . . . . . 15
  - xpaget: get data and info . . . . . 16
  - xpaset: send data and info . . . . . 15
  - xpainfo: send info alert . . . . . 16
  - xpaaccess: get access point info . . . . . 17
  - xpamb: message bus emulation . . . . . 19
  - xpans: the XPA name server . . . . . 22

- XPA Server Routines . . . . . 25
  - XPANew: define a new access point . . . . . 26
  - XPACmdNew: define a new command access point . . . . . 29
  - XPACmdAdd: add a command . . . . . 30
  - XPACmdDel: delete a command . . . . . 30
  - XPAInfoNew: define an info access point . . . . . 31
  - XPAFree: free an access point . . . . . 32
  - XPAMainLoop: event loop for select server . . . . . 32
  - XPAPoll: poll for XPA events . . . . . 33
  - XPACleanup: release reserved XPA memory . . . . . 34
  - XPA Server Macros: accessing structure internals . . . . . 34
  - XPA Race Conditions: how to avoid them . . . . . 35
  - XPA Out of Memory (OOM) errors . . . . . 37
- XPA Client Routines . . . . . 38
  - XPAOpen: open a persistent client connection . . . . . 46
  - XPAClose: close persistent client connection . . . . . 46
  - XPAGet: get data . . . . . 39
  - XPASet: send data or commands . . . . . 40
  - XPAInfo: send an info alert . . . . . 42
  - XPAGetFd: get data and write to an fd . . . . . 43
  - XPASetFd: read data from and fd and send . . . . . 44
  - XPANSLookup: look up an access point . . . . . 47
  - XPAAccess: get access info . . . . . 48
  - The XPA/Xt Interface: Xt interface to XPA . . . . . 50
  - The XPA/Tcl Interface: Tcl interface to XPA . . . . . 51
- Tailoring the XPA Environment
  - Environment Variables . . . . . 55
  - Access Control . . . . . 61
- Miscellaneous
  - XPA ChangeLog . . . . . 63
  - Where to Find Example/Test Code . . . . . 75
  - User Changes Between XPA 1.0 and 2.0 . . . . . 76
  - API Changes Between XPA 1.0 and 2.0 . . . . . 77
  - What Does XPA Stand For, Anyway? . . . . . 80


**Last updated: September 10, 2003**

# XPAIntro: Introduction to the XPA Messaging System

## Summary

A brief introduction to the XPA messaging system, which provides seamless communication between all kinds of Unix event-driven programs, including X programs, Tcl/Tk programs, and Perl programs.

## Description

The XPA messaging system provides seamless communication between all kinds of Unix programs, including X programs, Tcl/Tk programs, and Perl programs. It also provides an easy way for users to communicate with these XPA-enabled programs by executing XPA client commands in the shell or by utilizing such commands in scripts. Because XPA works both at the programming level and the shell level, it is a powerful tool for unifying any analysis environment: users and programmers have great flexibility in choosing the best level or levels at which to access XPA services, and client access can be extended or modified easily at any time.

A program becomes an XPA-enabled server by defining named points of public access through which data and commands can be exchanged with other client programs (and users). Using standard TCP sockets as a transport mechanism, XPA supports both single-point and broadcast messaging to and from these servers. It supports direct communication between clients and servers, or indirect communication via an intermediate message bus emulation program. Host-based access control is implemented, as is as the ability to communicate with XPA servers across a network.

XPA implements a layered interface that is designed to be useful both to software developers and to users. The interface consists of a library of XPA client and server routines for use in programs and a suite of high-level user programs built on top of these libraries. Using the XPA library, access points can be added to Tcl/Tk programs, Xt programs, or to Unix programs that use the XPA event loop or any event loop based on select(). Client access subroutines can be added to any Tcl/Tk or Unix program. Client access also is supported at the command line via a suite of high-level programs.

The major components of the XPA layered interface are:

- A set of XPA server routines, centered on XPANew(), which are used by XPA server programs to tag public access points with string identifiers and to register send and receive callbacks for these access points.
- A set of XPA client routines, centered on the XPASet() and XPAGet(), which are used by external client applications to exchange data and commands with an XPA server.
- High-level programs, centered on xpaset and xpaget, which allow data and information to be exchanged with XPA server programs from the command line and from scripts. These programs have the command syntax:

```
[data] | xpaset  [qualifiers ...]
         xpaget  [qualifiers ...]
```

- An XPA name server program, xpans, through which XPA access point names are registered by servers and distributed to clients.

Defining an XPA access point is easy: a server application calls XPANew(), XPACmdNew(), or the experimental XPAInfoNew() routine to create a named public access point. An XPA service can specify "send" and "receive" callback procedures (or an "info" procedure in the case of XPAInfoNew()) to be executed by the program when an external process either sends data or commands to this access point or requests data or information from this access point. Either of the callbacks can be omitted, so that a particular access point can be specified as read-only, read-write, or write-only. Application-specific client data can be associated with these callbacks. Having defined one or more public access points in this way, an XPA server program enters its usual event loop (or uses the standard XPA event loop).

Clients communicate with these XPA public access points using programs such as xpaget, xpaset, and xpainfo (at the command line), or routines such as XPAGet(), XPASet(), and XPAInfo() within a program. Both methods require specification of the name of the access point. The xpaget program returns data or other information from an XPA server to its standard output, while the xpaset program sends data or commands from its standard input to an XPA application. The corresponding API routines set/get data to/from memory, returning error messages and other info as needed. If a template is used to specify the access point name (e.g., "ds9*"), then communication will take place with all servers matching that template.

Please note that XPA currently is not thread-safe. All XPA calls must be in the same thread.

Go to XPA Help Index

**Last updated: March 10, 2007**

# XPATemplate: Access Point Names and Templates

## Summary

XPA access points are composed of two parts: a general class and a specific name. Both parts accept template characters so that you can send/retrieve data to/from multiple servers at one time.

## Description

When XPA servers call XPANew(), or XPACmdNew() to define XPA access points, they specify a string identifier composed of a class and a name. When clients communicate with XPA access points, they specify which access points to communicate with using an identifier of the form:

```
class:name
```

All registered XPA access points that match the specified identifier will be available for communication (subject to access control rules, etc.)

As of XPA 2.1.5, the length of both the class and name designations are limited to 1024 characters.

The XPA class:name identifier actually is a template: it accepts wild cards in its syntax, so a single specifier can match more than one XPA access point. (Note that the class is optional and defaults to "*".) The allowed syntax for clients to specify the class:name template is of the form shown below. (Note that "*" is used to denote a generic wild card, but other wild cards characters are supported, as described below).

```
template      explanation
--------      -----------
class:name    exact match of class and name
name          match any class with this name
*:name        match any class with this name
class:*       match any name of this class
*:*           match any access point
```

In general, the following wild-cards can be applied to class and name:

```
wildcard      explanation
--------      -----------
?             match any character, but there must be one
*             match anything, or nothing
[...]         match an inclusive set
```

Although the class:name template normally is used to refer to XPA access points, these also can be specified using their individual socket identifiers. For inet sockets, the socket identifier is **ip:port**, where ip can be the DNS-registered name, the ASCII IP number (e.g. 123.45.67.890) or the hex IP number (e.g. 838f3a60). For unix sockets, the identifier is the **socket file name**. These socket identifiers are displayed as the fourth argument in the xpans display of registered access points. For example, consider the ds9 program started using inet sockets. The xpans name server will register something like this:

```
csh> xpaget xpans
DS9 ds9 gs saord.harvard.edu:3236 eric
```

You can access ds9 using ip:3236 in any of the three forms:

```
csh> xpaget saord:3236 file
/home/eric/data/snr.ev

csh> xpaget 123.45.67.890:3236 file
/home/eric/data/snr.ev

csh> xpaget 838f3a60:3236 file
/home/eric/data/snr.ev
```

In the case of unix sockets, the socket identifier is a file:

```
csh> xpaget xpans
DS9 ds9 gs /tmp/.xpa/DS9_ds9.2631 eric

csh> xpaget /tmp/.xpa/DS9_ds9.2631 file
/home/eric/data/snr.ev
```

This feature can be useful in distinguishing between multiple instances of a program that all have the same class:name designation.

Go to XPA Help Index

**Last updated: September 10, 2003**

# XPACommon: Getting Common Information About Access Points

## Summary

There are various kinds of generic information you can retrieve about an XPA access point by using the xpaget command.

## Description

You can find out which XPA access points have been registered with the currently running XPA name server by executing the xpaget command to retrieve info from the XPA name server:

```
xpaget xpans
```

If, for example, the stest test server program is running, the following XPA access points will be returned (the specifics of the returned info will vary for different machines and users):

```
XPA xpa gs 838e2f67:1262 eric
XPA xpa1 gs 838e2f67:1266 eric
XPA c_xpa gs 838e2f67:1267 eric
XPA i_xpa i 838e2f67:1268 eric
```

Note that access to this information is subject to the usual XPA Access Control restrictions.

Each XPA access point supports a number of reserved sub-commands that provide access to different kinds of information, e.g. the access control for that access point. These sub-commands can be executed by using xpaset or xpaget at the command line, or XPAGet() or XPASet() in programs, e.g:

```
xpaget ds9 -acl
xpaget ds9 -help
xpaget ds9 env FOO

xpaset -p ds9 env FOO foofoo
```

With the exception of **-help** and **-version**, reserved sub-commands are available only on the machine on which the XPA server itself is running. The following reserved sub-commands are defined for all access points:

**-acl** get (set) the access control list [options: host type acl, for set]
: The 'xpaset' option allows you to add a new acl for a given host, or change the acl for an existing host. See XPA Access Control for more information. This access point is available only on the server machine.

**-env** get (set) an environment variable [options: name (value, for set)]
: The 'xpaget' option will return the value of the named environment variable. The 'xpaset' option will set the value of the names variable to the specified value. This access point is available only on the server machine. (Please be advised that we have had problems setting environment variables in static

Tcl/Tk programs such as ds9 running under Linux.)

**-clipboard** set(get) information on a named clipboard
Clients can store ASCII state information on any number of named clipboards. Clipboards of the same name created by clients on different machines are kept separate. The syntax for creating a clipboard is:

```
[data] | xpaset [server] -clipboard add|append [clipboard_name]
xpaset -p [server] -clipboard delete [clipboard_name]
```

Use "add" to create a new clipboard or replace the contents of an existing one. Use "append" to append to an existing clipboard.

Information on a named clipboard is retrieved using:

```
xpaget [server] -clipboard [clipboard_name]
```

**-exec** set: execute commands from buffer [options: none]
If -exec is specified in the paramlist of an 'xpaset' call, then further sub-commands will be retrieved from the data buffer.

**-help** get: return help string for this XPA or sub-command [options: name (for sub-commands)]
Each XPA access point and each XPA sub-command can have a help string associated with it that is specified when the access point is defined. The -help option will return this help string. For XPA access points that contain user-defined sub-commands, you can get the help string for a particular sub-command by specifying its name, or else get the help strings for all sub-commands if not name is specified.

**-ltimeout** get (set) the long timeout value [options: seconds|reset]
The 'xpaget' option will return the value of the long timeout (in seconds). The 'xpaset' option will set the value of the long timeout. If "reset" is specified, then the timeout value will be reset to the default value.

**-nsconnect** set: re-establish name server connection to all XPA's [options: none]
If the XPA Name Server (xpans) process has terminated unexpectedly and then re-started, this sub-command can be used to re-establish the connection. You use it by sending the command to the [name:port] or [file] of the access point instead of to the XPA name (since the latter requires the xpans connection!):

```
xpaset -p 838e2f67:1268 -nsconnect
```

See xpans for more information.

**-nsdisconnect** set: break name server connection to all XPA's [options: none]
This sub-command will terminate the connection to the XPA Name Server (xpans), thereby making all access points inaccessible except through their underlying [name:port] or [file] identifiers. I forget why we added it, it seems pretty useless.

**-stimeout** get (set) the short timeout value [options: seconds|reset]

The 'xpaget' option will return the value of the short timeout (in seconds). The 'xpaset' option will set the value of the short timeout. If "reset" is specified, then the timeout value will be reset to the default value.

**-remote** set: register xpa with remote server [options: host[:port] [acl]] [-proxy]

This sub-command will register the XPA access point with the XPA name server (xpans) on the specified host (which must already be running). The specified host also is given access control to the access point, using the specified acl or the default acl of "+" (meaning the remote host can xpaset, xpaget, xpainfo or xpaaccess). If the acl is specified as "-", then the access point is unregistered. See Communication Between Machines for more information on how this sub-command is used.

**-version** get: return XPA version string [options: none]

The version refers to the version of XPA used to define this access point (currently something like 2.0).

You can add your own reserved commands to all XPA access points by using the XPACmdAdd() routine, passing the XPA handle returned by *XPA XPAGetReserved(void)* as the first argument. Note again that these will only be available on the machine where the XPA service is running.

Go to XPA Help Index

## Last updated: September 10, 2003

# XPAMethod: XPA Communication Methods

## Summary

XPA supports both inet and unix (local) socket communication.

## Description

XPA uses sockets for communication between processes. It supports three methods of socket communication: inet, localhost, and unix. In general, the same method should be employed for all XPA processes in a session and the global environment variable XPA_METHOD should be used to set up the desired method. By default, the preferred method is "inet", which is appropriate for most users. You can set up a different method by typing something like:

```
setenv XPA_METHOD local              # unix csh
XPA_METHOD=local; export XPA_METHOD  # unix sh, bash, windows/cygwin
set XPA_METHOD=localhost             # dos/windows
```

The options for XPA_METHOD are: **inet**, **unix** (or **local**), and **localhost**. On Unix machines, this environment setup command can be placed in your shell init file (.cshrc, .profile, .bashrc, etc.) On Windows platforms, it can be placed in your AUTOEXEC.BAT file (I think!).

By default, **inet** sockets are used by XPA. These are the standard Internet sockets that are used by programs such as Netscape, ftp. etc. Inet sockets utilize the IP address of the given machine and a (usually random) port number to communicate between processes on the same machine or between different machines on the Internet. (Note that XPA has an Access Control mechanism to prevent unauthorized access of XPA access points by other computers on the Net). For users connected to the Internet, this usually is the appropriate communication method. For more information about setting up XPA communication between machines, see Communication Between Machines.

In you are using XPA on a machine without an Internet connection, then inet sockets are not appropriate. In fact, an XPA process often will hang for many seconds while waiting for a response from the Domain Name Service (DNS) when using inet sockets. Instead of inet sockets, users on Unix platforms can also use **unix** sockets (also known as local sockets). These sockets are based on the local file system and do not make use of the DNS. They generally are considered to be faster than inet sockets, but they are not implemented under Windows. Use local sockets as a first resort if you are on a Unix machine that is not connected to the Internet.

Users not connected to the Internet also can use **localhost** sockets. These are also inet-type sockets but the IP address used for the local machine is the **localhost** address, 0x7F000001, instead of the real IP of the machine. Depending on how sockets are set up for a given platform, communication with the DNS usually is not required in this case (though of course, XPA cannot interact with other machines). The localhost method will generally work on both Unix and Windows platforms, but whether the DNS is required or not is subject to individual configurations.

A final warning/reminder: if your XPA-enabled server hangs at startup time and your XPA_METHOD is **inet**, the problem probably is related to an incorrect Internet configuration. This can be confirmed by using the **unix** method or (usually) the **localhost** method. You can use these alternate methods if other hosts do not need access to the XPA server.

Go to XPA Help Index

**Last updated: September 10, 2003**

# XPAInet: XPA Communication Between Hosts

## Summary

XPA uses standard inet sockets to support communication between two or more host computers.

## Description

When the Communication Method is set to **inet** (as it is by default), XPA can be used to communicate between different computers on the Internet. INET sockets utilize the IP address of the given machine and a (usually random) port number to communicate between processes on the same machine or between different machines on the Internet. These standard Internet sockets are also used by programs such as Netscape, ftp. etc.

XPA supports a host-based Access Control mechanism to prevent unauthorized access of XPA access points by other computers on the Net. By default, only the machine on which the XPA server is running can access XPA services. Therefore, setting up communication between a local XPA server machine and a remote client machine requires a two-part registration process:

- the XPA service on the local machine must be made known to the remote machine
- the remote machine must be given permission to access the local XPA service

Three methods by which this remote registration can be accomplished are described below.

## Manual Registration

The first method is the most basic and does not require the remote client to have xpans running. To use it, the local server simply gives a remote client machine access to one or more XPA access points using xpaset and the **-acl** sub-command. For example, consider the XPA test program "stest" running on a local machine. By default the access control for the access point named "xpa" is restricted to that machine:

```
[sh]$ xpaget xpa -acl
*:* 123.456.78.910 gisa
*:* localhost gisa
```

Using xpaset and the **-acl** sub-command, a remote client machine can be given permission to perform xpaget, xpaset, xpaaccess, or xpainfo operations. For example, to allow the xpaget operation, the following command can be issued on the local machine:

```
[sh]$ xpaset -p xpa -acl "remote_machine g"
```

This results in the following access permissions on the local machine:

```
[sh]$ xpaget xpa -acl
XPA:xpa 234.567.89.012 g
*:* 123.456.78.910 gisa
*:* localhost gisa
```

The remote client can now use the local server's xpans name server to establish communication with the local XPA service. This can be done on a call-by-call basis using the **-i** switch on xpaset, xpaget, etc:

```
[sh]$ xpaget -i "local_machine:12345" xpa
class: XPA
name: xpa
method: 88877766:2778
sendian: little
cendian: big
```

Alternatively, the XPA_NSINET variable on the remote machine can be set to point directly to xpans on the local machine, removing the need to override this value each time an XPA program is run:

```
[csh]$ setenv XPA_NSINET 'karapet:$port'
[csh]$ xpaget xpa
class: XPA
name: xpa
method: 88877766:2778
sendian: little
cendian: big
```

Here, '$port' means to use the default XPA name service port (14285). not a port environment variable.

Access permission for remote client machines can be stored in a file on the local machine pointed to by the **XPA_ACLFILE** environment variable or using the **XPA_DEFACL** environment variable. See XPA Access Control for more information.

# Remote Registration

If xpans is running on the remote client machine, then a local xpaset command can be used with the **-remote** sub-command to register the local XPA service in the remote name service, while at the same time giving the remote machine permission to access the local service. For example, assume again that "stest" is running on the local machine and that xpans is also running on the remote machine. To register access of this local xpa on the remote machine, use the xpaset and the **-remote** sub-command:

```
[sh]$ ./xpaset -p xpa -remote 'remote_machine:$port' +
```

To register the local xpa access point on the remote machine with xpaget access only, execute:

```
[sh]$ ./xpaset -p xpa -remote 'remote_machine:$port' g
```

Once the remote registration command is executed, the remote client machine will have an entry such as the following in its own xpans name service:

```
[csh]$ xpaget xpans
XPA xpa gs 88877766:2839 eric
```

The xpa access point can now be utilized on the remote machine without further setup:

```
[csh]$ xpaget xpa
class: XPA
name: xpa
method: 838e2f68:2839
sendian: little
cendian: big
```

To unregister remote access from the local machine, use the same command but with a '-' argument:

```
[sh]$ xpaset -p xpa -remote 'remote_machine:$port' -
```

The benefit of using remote registration is that communication with remote access points can be mixed with that of other access points on the remote machine. Using Access Point Names and Templates, one XPA command can be used to send or receive messages to the remote and local services.

# XPANS Proxy Registration

The two methods described above are useful when the local and remote machines are able to communicate freely to one another. This would be the case on most Local Area Networks (LANs) where all machines are behind the same firewall and there is no port blocking between machines on the same LAN. The situation is more complicated when the XPA server is behind a firewall, where outgoing connections are allowed, but incoming port blocking is implemented to prevent machines outside the firewall from connecting to machines inside the firewall. Such incoming port blocking will prevent xpaset and xpaget from connecting to an XPA server inside a firewall.

To allow locally fire-walled XPA services to register with remote machines, we have implemented a proxy service within the xpans name server. To register remote proxy service, xpaset and the **-remote** sub-command is again used, but with an additional **-proxy** argument added to the end of the command:

```
[sh]$ ./xpaset -p xpa -remote 'remote_machine:$port' g -proxy
```

Once a remote proxy registration command is executed, the remote machine will have an entry such as the following in its own xpans name service:

```
[csh]$ xpaget xpans
XPA xpa gs @88877766:2839 eric
```

The '@' sign in the name service entry indicates that xpans proxy processing is being used for this access point. Other than that, from the user's point of view, there is no difference in how this XPA access point is contacted using XPA programs (xpaset, xpaget, etc.) or libraries:

```
[csh]$ xpaget xpa
class: XPA
name: xpa
method: 88877766:3053
sendian: little
cendian: big
```

Of course, the underlying processing of the XPA requests is very much different when xpans proxy is involved. Instead of an XPA program such contacting the XPA service directly, it contacts the local xpans. Acting as a proxy server, xpans communicates with the XPA service using the command channel established at registration time. Commands (including establishing a new data channel) are sent between

xpans and the XPA service to set up a new message transfer, and then data is fed to/from the xpa request, through xpans, from/to the XPA service. In this way, it can be arranged so that connections between the fire-walled XPA service and the remote client are always initiated by the XPA service itself. Thus, incoming connections that would be blocked by the firewall are avoided. Note that there is a performance penalty for using the xpans/proxy service. Aside from extra overhead to set up proxy communication, all data must be sent through the intermediate proxy process.

The xpans proxy scheme requires that the remote client allow the local XPA server machine to connect to the remote xpans/proxy server. If the remote client machine also is behind a port-blocking firewall, such connections will be disallowed. In this case, the only solution is to open up some ports on the remote client machine to allow incoming connections to xpans/proxy. Two ports must be opened (for command and data channel connections). By default, these two ports are 14285 and 14287. The port numbers can be changed using the **XPA_NSINET** environment variable. This variable takes the form:

```
setenv XPA_NSINET machine:port1[,port2[,port3]]
```

where port1 is the main connecting port, port2 is the XPA access port, and port3 is the secondary data connecting port. The second and third ports are optional and default to port1+1 and port1+2, respectively. It is port1 and port3 that must be left open for incoming connections.

For example, to change the port assignments so that xpans listens for registration commands on port 12345 and data commands on port 28573:

```
setenv XPA_NSINET myhost:12345
```

Alternatively, all three ports can be assigned explicitly:

```
setenv XPA_NSINET remote:12345,3000,12346
```

In this case 12345 and 12346 should be open for incoming connections. The XPA access port (which need not be open to the outside world) is set to 3000.

Finally, note that we currently have no mechanism to cope with Internet proxy servers (such as SOCKS servers). If an XPA service is running on a machine that cannot connect directly to outside machines, but goes through a proxy server instead, there currently is no way to register that XPA service with a remote machine. We hope to implement support for SOCKS proxy in a future release.

Go to XPA Help Index

### Last updated: September 10, 2003

# XPAUsers: Distinguishing Users

## Summary

XPA normally distinguishes between users on a given host, but it is possible to send data to access points belonging to other users.

## Description

A single XPA name service typically serves all users on a given machine. Two users can register the same XPA access points on the same machine without conflict, because the user's username is registered with each access point and, by default, programs such as xpaget and xpaset only process access points of the appropriate user. For example:

```
XPA xpa1 gs 838e2f67:1262 eric
XPA xpa2 gs 838e2f67:1266 eric
XPA xpa1 gs 838e2f67:2523 john
XPA xpa2 gs 838e2f67:2527 john
```

Here the users "eric" and "john" both have registered the access points xpa1 and xpa2. When either "john" or "eric" retrieves information from xpa1, they will process only the access point registered in their user name.

If you want to access another user's XPA access points on a single machine, use the -u [user] option on xpaset, xpaget, etc. For example, if eric executes:

```
xpaget -u john xpa1
```

he will access John's xpa1 access point.Use "*" to access all users on a given machine:

```
xpaget -u "*" xpa1
```

Note that the XPA Environment Variable XPA_NSUSERS can be used to specify the default list of users to process:

```
setenv XPA_NSUSERS "eric,john"
```

will cause access points from both "eric" and "john" to be processed by default.

Go to XPA Help Index

**Last updated: September 10, 2003**

# XPA Programs

# Summary

Use the XPA programs to send/receive data to/from XPA servers from the command line or from scripts.

```
<data> | xpaset  [-h] [-i nsinet] [-m method] [-n] [-p] [-s] [-t sval,lval] [-u users] [-v] <template> [paramlist]

xpaget  [-h] [-i nsinet] [-m method] [-s] [-t sval,lval] [-u users] <template> [paramlist]

xpainfo [-h] [-i nsinet] [-m method] [-n] [-s] [-t sval,lval] [-u users] <template> [paramlist]

xpaaccess [-c] [-h] [-i nsinet] [-m method] [-n] [-u users] [-v|-V] <template> [type]
```

# xpaset: send data to one or more XPA servers

```
<data> | xpaset  [-h] [-i nsinet] [-m method] [-n] [-p] [-s] [-t sval,lval] [-u users] [-v] <template|host:port> [paramlist]
```

```
  -h            print help message
  -i            access XPA point on different machine (override XPA_NSINET)
  -m            override XPA_METHOD environment variable
  -n            don't wait for the status message after server completes
  -p            don't read (or send) buf data from stdin
  -s            enter server mode
  -t [s,l]      set short and long timeouts (override XPA_[SHORT,LONG]_TIMEOUT)
  -u [users]    XPA points can be from specified users (override XPA_NSUSERS)
  -v            verify message to stdout
  --version     display version and exit
```

Data read from stdin will be sent to access points matching the template or host:port. A set of qualifying parameters can be appended.

Normally, xpaset reads data input from stdin until EOF and sends those data to the XPA target, along with parameters entered on the command line. For example to send a FITS file to the ds9 image display:

```
cat foo.fits | xpaset ds9 fits
```

Sometimes, however, it is desirable to send only parameters to an XPA access point, without sending data. For such cases, use the -p switch to indicate that there is no data being send to stdin. For example, to change the colormap used by the ds9 image display program, use:

```
csh> xpaset -p ds9 cmap Heat
```

Of course, this also can be accomplished by sending EOF to stdin in any of the usual ways:

```
csh> echo "" | xpaset ds9 cmap Heat
csh> xpaget ds9 cmap Heat < /dev/null
csh> xpaset ds9 cmap Heat
^D                    # Ctl-D signals EOF
```

The -s switch puts xpaset into server mode, in which commands and data can be sent to access points without having to run xpaset multiple times. (Its not clear if this buys you much!) The syntax for sending commands in server mode is:

```
  csh> xpaset -s
  xpaset ds9 colormap I8
  ^D
  xpaset ds9 regions
  circle 200 300 40
  circle 300 400 50
  ^D
etc.
```

After the required "xpaset" command is specified, optional ASCII data can be appended (as in the region example). A single data/command set is delimited by ^D. Note that typing ^D when a command is expected terminates the program.

NB: server mode only works from the terminal and only ASCII data can be sent in this way.

**Examples:**

```
  csh> xpaset ds9 file < foo.fits
  csh> echo "stop" | xpaset myhost:12345
```

# xpaget: retrieve data from one or more XPA servers

```
xpaget [-h] [-i nsinet] [-m method] [-s] [-t sval,lval] [-u users] <template|host:port> [paramlist]

  -h            print help message
  -i            access XPA point on different machine (override XPA_NSINET)
  -m            override XPA_METHOD environment variable
  -n            don't wait for the status message after server completes
  -s            enter server mode
  -t [s,l]      set short and long timeouts (override XPA_[SHORT,LONG]_TIMEOUT)
  -u [users]    XPA points can be from specified users (override XPA_NSUSERS)
  --version     display version and exit
```

Data will be retrieved from access points matching the template or host:port. A set of qualifying parameters can be appended.

**Examples:**

```
  csh> xpaget ds9 images
  csh> xpaget myhost.harvard.edu:12345
```

# xpainfo: send short message to one or more XPA servers

```
xpainfo [-h] [-i nsinet] [-m method] [-n] [-s] [-t sval,lval] [-u users] <template|host:port> [paramlist]

  -h            print help message
  -i            access XPA point on different machine (override XPA_NSINET)
  -m            override XPA_METHOD environment variable
  -n            don't wait for the status message after server completes
  -s            enter server mode
  -t [s,l]      set short and long timeouts (override XPA_[SHORT,LONG]_TIMEOUT)
  -u [users]    XPA points can be from specified users (override XPA_NSUSERS)
  --version     display version and exit
```

Info will be sent to access points matching the template or host:port. A set of qualifying parameters can be appended.

**Examples:**

```
csh> xpainfo IMAGE ds9 image
```

# xpaaccess: see if template matches registered XPA access points

```
xpaaccess [-c] [-h] [-i nsinet] [-m method] [-n] [-t sval,lval] [-u users] -v <template> [type]

  -c            contact each access point individually
  -h            print help message
  -i            access XPA point on different machine (override XPA_NSINET)
  -m            override XPA_METHOD environment variable
  -n            return number of matches instead of "yes" or "no"
  -t [s,l]      set short and long timeouts (override XPA_[SHORT,LONG]_TIMEOUT)
  -u [users]    XPA points can be from specified users (override XPA_NSUSERS)
  -v            print info about each successful access point
  -V            print info or error about each access point
  --version     display version and exit
```

xpaaccess returns "yes" to stdout (with a return error code if 1) if there are existing XPA access points that match the template (and optional access type: g,i,s). Otherwise, it returns "no" (with a return error code of 0). If -n is specified, the number of matches is returned instead (both to stdout and in the returned error code). If -v is specified, each access point is displayed to stdout instead of the number of matches.

By default, xpaaccess simply contacts the xpans name server to find the list of registered access points that match the specified template. It also checks to make sure the specified types are supported by that access point. This is the fastest way to determine available access points. However, an access point might registered but not yet available, if, for example, the server program has not entered its event loop to process XPA requests. To find access points that are guaranteed to be available for processing, use the -c (contact) switch. With this switch, xpaaccess contacts each matching XPA server (rather than the name server) to make sure the registered access point really is ready for processing. In this mode, if an access point is registered but not available, xpaaccess will pause for a period of time equal to the XPA_LONG_TIMEOUT, in order to give the server a chance to ready itself. By default, this timeout is 30 seconds. You can shorten the time of delay using the -t "short,long" switch. For example, to shorten the delay time to 2 seconds, use:

```
xpaaccess -c -t "2,2" ds9
```

The first argument is the short delay value, and is ignored in this operation. The second is the long delay timeout.

Note also that the default xpaaccess method (no -c switch) does not check access control (acls) but rather only checks whether the access point is both registered with the xpans name server and provides the specified type of access. In other words, the default xpaaccess could return ’yes’ when you might not actually have access. This mode also always returns ’yes’ for the xpans name server itself, regardless of whether the name server is active. The -c (contact) switch, which contacts the access point directly, can and does check the access control (only for servers using version 2.1 and above) and also returns the real

status of xpans.

Go to XPA Help Index

**Last updated: September 10, 2003**

# xpamb: the XPA Message Bus

## Summary

The xpamb program can act as a "classical" message bus interface between clients and servers. A client can send a data request to the message bus, which then interfaces with multiple servers and returns the data back to the client.

## Description

A "classical" message bus (such as ToolTalk) consists of servers and clients, along with a mediating program that transfers data between different processes. XPA takes a slightly different approach in that communication between clients and servers is direct. This generally is the correct technique when there is only one connection (or even a small number of connections), but can become inefficient for the serving program if a large amount of data is being transferred to many clients. For example, if a real-time data acquisition program is broadcasting a FITS image to several clients, it would need to transmit that image to each client individually. This might interfere with its own processing cycles. The preferable mechanism would be to pass the image off to an intermediate program that can then broadcast the data to the several clients.

The **xpamb** program can alleviate such problems by functioning as a message bus in cases where such an intermediary process is wanted. It pre-defines a single access point named **XPAMB:xpamb** to which data can be sent for re-broadcast. You also can tell **xpamb** to save the data, and associate with that data a new access point, so that it can be retrieved later on.

All interaction with **xpamb** is performed through **xpaset** and **xpaget** (or the corresponding API routines, **XPASet()** and **XPAGet()**) to the **XPAMB:xpamb** access point. That is, **xpamb** is just another XPA-enabled program that responds to requests from clients. The paramlist is used to specify the targets to which the data will be for re-broadcast, as well as the re-broadcast paramlist:

```
data | xpaset xpamb [switches] broadcast-target broadcast-paramlist
```

Optional switches are used to store data, and manipulate stored data, and are described below.

In its simplest form, you can, for example, send a FITS image to xpamb for broadcasting to all ds9 image simply by executing:

```
cat foo.fits | xpaset xpamb "DS9:*" fits foo.fits
```

Since **DS9** is the class name for the ds9 image display program, this will result in the FITS image being re-sent to all fits access points for all active image display programs.

You can send stored data and new data to the same set of access points at the same time. The stored data always is send first, followed by the new data:

```
cat foo2.fits | xpaset xpamb -send foo "DS9:*" fits foo.fits
```

will first send the foo.fits file, and then the foo2.fits file to all access points of class **DS9**. Notice that in this example, the foo2.fits file is not stored, but it could be stored by using the **-store [name]** switch on the command line.

The **xpaget** command can be used to retrieve a data from XPA access points or from a stored data buffer, or retrieve information about a stored data buffer. If no arguments are given:

```
xpaget xpamb
```

then information about all currently stored data buffers is returned. This information includes the data and time at which the data was stored, the size in bytes of the data, and the supplied info string.

If arguments are specified, they will be in the form:

```
xpaget xpamb [-info] [-data] [name [paramlist]]
```

If the optional **-info** and/or **-data** switches are specified, then information and/or data will be returned for the named data buffer following the switches. You can use either or both of these switches in a single command. For example, if the -info switch is used:

```
xpaget xpamb -info foo
```

then the info about that stored data buffer will be returned. If the -data is used with a specific name:

```
xpaget xpamb -data foo
```

then the stored data itself will be returned. If both are used:

```
xpaget xpamb -info -data foo
```

then the info will be returned, followed by the data. Note that it is an error to specify one of these switches without a data buffer name and that the paramlist will be ignored.

If neither the **-info** or **-data** switch is specified, then the name refers to an XPA access point (with an optional paramlist following). For example:

```
xpaget xpamb ds9 file
```

is equivalent to:

```
xpaget ds9 file
```

# Options

For xpaset, several optional switches are used to save data and manipulate the stored data:

**-data [name]**
: Add the supplied data buffer to a pool of stored data buffers, using the specified name as a unique identifier for later retrieval. An error occurs if the name already exists (use either **replace** or **del** to rectify this). The **-add** switch is supported for backwards compatibility with xpa 2.0.

**-replace [name]**
Replace previously existing stored data having the same unique name with new data. This essentially is a combination of the **del** and **data** commands.

**-info ["'info string'"]**
When adding a data buffer, you can specify an informational string to be stored with that data. This string will be returned by xpaget:

```
xpaget xpamb foo -info
```

(along with other information such as the date/time of storage and the size of the data buffer) if the -info switch is specified. If the info string contains spaces, you must enclose it in **two** sets of quotes:

```
cat foo | xpaset xpamb -store foo -info "'this is info on foo'"
```

The first set of quotes is removed by the shell while the second is used to delineate the info string.

**-send [name]**
Broadcast the stored data buffer to the named template.

**-del [name]**
Delete the named data buffer and free all allocated space.

Switches can be used in any combination that makes sense. For example:

```
cat foo.fits | xpaset xpamb -store foo -info "FITS" "DS9:*" fits foo.fits
```

will broadcast the foo.fits image to all access points of class **DS9**. In addition, the foo.fits file will be stored under the name of **foo** for later manipulation such as:

```
xpaset -p xpamb -send foo "DS9:*" fits foo.fits
```

will re-broadcast the foo.fits image to all access points of class "DS9".

Go to XPA Help Index

## Last updated: September 10, 2003

# xpans: the XPA Name Server

# Summary

```
xpans [-h] [-e] [-k sec] [-p port] [-l log] [-s security log] [-P n]

-h            print help message
-e            exit when there are no more XPA connections
-k            send keepalive messages every n sec
-l            log data base entries to specified file
-p            listen for connections on specified port
-s            log security info for each connection to specified file
-P            accept proxy requests (P=1) using separate thread (P=2)
--version     display version and exit
```

The xpans name server is an XPA-enabled program that is used to manage the names and ports of XPA access points. It is started automatically when an XPA access point is registered. You can access the name server using xpaget to get a list of registered access points.

The *xpans* name server provides a crucial link between XPA clients and servers. When an XPA server defines an access point using XPANew(), XPACmdNew(), or XPAInfoNew(), the name of the access point is registered in the name service, along with connection information. The name server then matches class:name templates passed to it by XPA clients with these registered entries, so that the clients can communicate with the appropriate servers.

The socket connection between an XPA-enabled program and *xpans* is kept open until the former exits (or explicitly closes the connection). Apparently, some Internet equipment (e.g. DSL modems) can cause such a connection to time-out after a period of inactivity. To prevent this from happening, you can use the *-k [sec]* switch to send a short keep-alive message to each open connection after the specified time delay. (Note that this application level use of keep-alive is necessary only if you are serving XPA-enabled clients over the Internet and have to deal with long-term connections involving DSL or similar equipment. XPA uses the ordinary socket-level keep-alive, which works for all other cases.) **NB (12/2/2009): Out-of-band (URG) TCP data, used by xpans keep-alive, is changed by some Cisco routers into in-band data. Encountering such a router will break the keep-alive function and may break your XPA server as well. Proceed with caution!**

The *xpans* program will be started automatically (assuming it can be found in the user's path) when the first XPA access point is registered. It therefore need not be started explicitly. However, when started automatically, the *-e* switch is used, so that the name server will exit when there are no more XPA access points registered. If you wish to keep the name server running continually, simply start it manually without the *-e* switch.

The name server will keep a log of registered access points if the *-l [log]* switch is used on the command line (this is the case for automatic start-up). The log contains enough name and connection information to allow you to re-register all XPA access points in case the name server process is terminated prematurely. For example, after the ds9 access point is registered,the log will contain the entry:

```
add 838e2f67:1863 ds9 ds9 gs eric
```

If *xpans* is terminated but ds9 still is running, you can re-register both access points for the ds9 process by running:

```
xpaset -p 838e2f67:1863 -nsconnect
```

Notice that the ip:port specifier is used with *xpaset* to bypass the need for contacting the name server (which does not have the name registered yet!)

The name server will keep a log of security information if the *-s [security log]* switch is used on the command line. For each accepted connection, (including connections via the *xpaget* command), information will be logged about the host issuing the command and the parameters passed into the program. This is most useful when *xpans* is accepting connections from untrusted machines.

When an XPA access point is removed by a server using *XPAFree()*, the access information is removed from the name server. If an XPA-enabled process is terminated, all names registered by that process will be removed automatically. The log file is always updated to reflect the currently registered access points.

The name server itself has an XPA access point names *xpans* registered through which you can find out information about currently registered access points (assuming you have access to the name server; see XPA Access Control for more information). For each registered access point, the following information is returned:

```
class         # class of the access point
name          # name of the access point
access        # allowed access (g=xpaget,s=xpaset,i=xpainfo)
id            # socket access method (host:port for inet, file for local/unix)
user          # user name of access point owner
```

For example, to display all currently registered access points, simply execute:

```
xpaget xpans
```

Continuing the example of ds9 above, this will return:

```
DS9 ds9 gs 838e2f67:1863 eric
```

If the same program has been started with different XPA access names, you can look up only names matching a specified template. For example, assume that ds9 has been started up using:

```
ds9 &
ds9 -title ds9-1-eric &
ds9 -title ds9-2-eric &
```

To lookup all ds9 access points which end in ".eric" and which can be accessed using *xpaset*, use:

```
xpaget xpans "DS9:*.eric" "s" "*"
```

This will return:

```
DS9 ds9-2-eric gs 838e29d3:42102 eric
DS9 ds9-1-eric gs 838e29d3:42105 eric
```

The third argument "*" requests all access points from all users. You also can specify a specific user name and only access points registered by that user will be returned.

The name server uses the *XPA_METHOD* environment variable to determine whether it should listen for requests on INET or LOCAL sockets. Since XPA access points also use this environment variable, the choice of socket method will be consistent. Note that, when INET sockets are used, a local server can be accessed from remote machines if the *XPA_NSINET* environment variable is set to point to the local machine. See XPA Environment Variables for more information.

An experimental feature of xpans is its ability to act as a proxy to XPA servers behind firewalls that want to communicate with external processes. The basic idea is the following: an XPA server (call it "foo") on host1, possibly behind a firewall, makes a remote connection to a proxy-enabled xpans program on host2 (specifying host2's XPA method). For example:

```
xpaset -p foo -remote 'host2:28571' + -proxy   # on host1
```

When this is done, host2 can use xpaset, xpaget, and xpainfo calls to communicate with the XPA server foo. All command communication is performed via the xpans socket connection between foo on host1 and xpans on host2 (which was initiated by foo from inside the firewall). Data communication is similarly performed using a socket connection initiated on host1 (usually with a port value two greater than the port value of the main xpans socket connection). An xpaset or xpaget call on host2 contacts xpans, which performs an XPASet() or XPAGet() call to foo, passing commands and data back and forth between the two programs.

By default, proxy connections are not allowed by xpans. If the -P switch is specified with a value of 1, proxy connection are allowed, but all proxy communication is performed in the same thread as xpans processing. If a value of 2 is specified, the proxy processing is performed in a separate thread (assuming pthreads are supported on your system). Because xpa callback processing of any type can take a long time and therefore can interfere with normal xpans processing, threaded proxy connections (-P 2) are recommended. When using proxy connections, it might also be useful to set the XPA_IOCALLSXPA environment variable, so that multiple proxy requests can be handled at the same time, instead of serially.

Note that this proxy interface to xpans is experimental. It is used to provide remote data analysis capabilities on the Chandra-Ed system using ds9. (See http://chandra-ed.cfa.harvard.edu and http://hea-www.harvard.edu/saord/ds9 for more details). As always, please contact us if you have problems or questions.

Go to XPA Help Index

**Last updated: January 24, 2005**

# XPAServer: The XPA Server-side Programming Interface

## Summary

A description of the XPA server-side programming interface.

## Introduction to XPA Server Programming

Creating an XPA server is easy: you generally only need to call the XPANew() subroutine to define a named XPA access point and set up the send and receive callback routines. You then enter an event loop such as XPAMainLoop() to field XPA requests.

```
#include <xpa.h>

XPA XPANew(char *class, char *name, char *help,
    int (*send_callback)(), void *send_data, char *send_mode,
    int (*rec_callback)(),  void *rec_data,  char *rec_mode);

XPA XPACmdNew(char *class, char *name);

XPACmd XPACmdAdd(XPA xpa,
       char *name, char *help,
       int (*send_callback)(), void *send_data, char *send_mode,
       int (*rec_callback)(),  void *rec_data,  char *rec_mode);

void XPACmdDel(XPA xpa, XPACmd cmd);

XPA XPAInfoNew(char *class, char *name,
    int (*info_callback)(), void *info_data, char *info_mode);

int XPAFree(XPA xpa);

void XPAMainLoop(void);

int XPAPoll(int msec, int maxreq);

void XPAAtExit(void);

void XPACleanup(void);
```

## Introduction

To use the XPA application programming interface, a software developer generally will include the xpa.h definitions file:

```
#include <xpa.h>
```

in the software module that defines or accesses an XPA access point, and then will link against the libxpa.a library:

```
gcc -o foo foo.c libxpa.a
```

XPA has been compiled using both C and C++ compilers.

A server program generally defines an XPA access point by calling the XPANew() routine and specifies "send" and/or "receive" callback procedures to be executed by the program when an external process either sends data or commands to this access point or requests data or information from this access point. A program also can define several sub-commands for a single access point by calling XPACmdNew() and XPACmdAdd() instead. Having defined one or more public access points in this way, an XPA server program enters its usual event loop (or uses the standard XPA event loop).

# XPANew: create a new XPA access point

```
#include <xpa.h>

XPA XPANew(char *class, char *name, char *help,
           int (*send_callback)(),
           void *send_data, char *send_mode,
           int (*rec_callback)(),
           void *rec_data,  char *rec_mode);
```

Create a new XPA public access point with the class:name identifier template and enter this access point into the XPA name server, so that it can be accessed by external processes. XPANew() returns an XPA struct. Note that the length of the class and name designations must be less than or equal to 1024 characters each.

The XPA name server daemon, xpans, will be started automatically if it is not running already (assuming it can be found in the path). The program's ip address and listening port are specified by the environment variable XPA_NSINET, which takes the form :. If no such environment variable exists, then xpans is started on the current machine listening on port 14285. It also uses 14286 as a known port for its public access point (so that routines do not have to go to the name server to find the name server ip and port!) As of XPA 2.1.1, version information is exchanged between the xpans process and the new access point. If the access point uses an XPA major/minor version newer than xpans, a warning is issued by both processes, since mixing of new servers and old xpa programs (xpaset, xpaget, xpans, etc.) is not likely to work. You can turn off the warning message by setting the XPA_VERSIONCHECK environment variable to "false".

The help string is meant to be returned by a request from xpaget:

```
xpaget class:name -help
```

A send_callback and/or a receive_callback can be specified; at least one of them must be specified.

A send_callback can be specified that will be executed in response to an external request from the xpaget program, the XPAGet() routine, or XPAGetFd() routine. This callback is used to send data to the requesting client.

The calling sequence for send_callback() is:

```
int send_callback(void *send_data, void *call_data,
  char *paramlist, char **buf, size_t *len)
{
  XPA xpa = (XPA)call_data;
  ...
  return(stat);
}
```

The send_mode string is of the form: "key1=value1,key2=value2,..." The following keywords are recognized:

| key | value | default | explanation |
|---|---|---|---|
| acl | true/false | true | enable access control |
| freebuf | true/false | true | free buf after callback completes |

The call_data should be recast to the XPA struct as shown. In addition, client-specific data can be passed to the callback in send_data.

The paramlist will be supplied by the client as qualifying parameters for the callback. There are two ways in which the send_callback() routine can send data back to the client:

1. The send_callback() routine can fill in a buffer and pass back a pointer to this buffer. An integer len also is returned to specify the number of bytes of data in buf. XPA will send this buffer to the client after the callback is complete.

2. The send_callback can send data directly to the client by writing to the fd pointed by the macro:

```
xpa_datafd(xpa)
```

Note that this fd is of the kind returned by socket() or open().

If a buf has been allocated by a standard malloc routine, filled, and returned to XPA, then freebuf generally is set so that the buffer will be freed automatically when the callback is completed and data has been sent to the client. If a static buf is returned, freebuf should be set to false to avoid a system error when freeing static storage. Note that default value for freebuf implies that the callback will allocate a buffer rather than use static storage.

On the other hand, if buf is dynamically allocated using a method other than a standard malloc/calloc/realloc routine (e.g. using Perl's memory allocation and garbage collection scheme), then it is necessary to tell XPA how to free the allocated buffer. To do this, use the XPASetFree() routine within your callback:

```
void XPASetFree(XPA xpa, void (*myfree)(void *), void *myfree_ptr);
```

The first argument is the usual XPA handle. The second argument is the special routine to call to free your allocated memory. The third argument is an optional pointer. If not NULL, the specified free routine is called with that pointer as its sole argument. If NULL, the free routine is called with the standard buf pointer as its sole argument. This is useful in cases where there is a mapping between the buffer pointer

and the actual allocated memory location, and the special routine is expecting to be passed the former.

If, while the callback performs its processing, an error occurs that should be communicated to the client, then the routine XPAError should be called:

```
XPAError(XPA xpa, char *s);
```

where s is an arbitrary error message. The returned error message string will be of the form:

```
XPA$ERROR   [error] (class:name ip:port)
```

If the callback wants to send a specific acknowledgment message back to the client, the routine XPAMessage can be called:

```
XPAMessage(XPA xpa, char *s);
```

where s is an arbitrary error message. The returned error message string will be of the form:

```
XPA$MESSAGE [message] (class:name ip:port)
```

Otherwise, a standard acknowledgment is sent back to the client after the callback is completed.

The callback routine should return 0 if no error occurs, or -1 to signal an error.

A receive_callback can be specified that will be executed in response to an external request from the xpaset program, or the XPASet (or XPASetFd()) routine. This callback is used to process data received from an external process.

The calling sequence for receive_callback is:

```
int receive_callback(void *receive_data, void *call_data,
  char *paramlist, char *buf, size_t len)
{
  XPA xpa = (XPA)call_data;
  ...
  return(stat);
}
```

The mode string is of the form: "key1=value1,key2=value2,..." The following keywords are recognized:

```
key           value          default         explanation
------        --------       --------        -----------
acl           true/false     true            enable access control
buf           true/false     true            server expects data bytes from client
fillbuf       true/false     true            read data into buf before executing callback
freebuf       true/false     true            free buf after callback completes
```

The call_data should be recast to the XPA struct as shown. In addition, client-specific data can be passed to the callback in receive_data.

The paramlist will be supplied by the client. In addition, if the receive_mode keywords buf and fillbuf are true, then on entry into the receive_callback() routine, buf will contain the data sent by the client. If buf is true but fillbuf is false, it becomes the callback's responsibility to retrieve the data from the client, using the data fd pointed to by the macro xpa_datafd(xpa). If freebuf is true, then buf will be freed when the callback is complete.

If, while the callback is performing its processing, an error occurs that should be communicated to the client, then the routine XPAError can be called:

```
XPAError(XPA xpa, char *s);
```

where s is an arbitrary error message.

The callback routine should return 0 if no error occurs, or -1 to signal an error.

# XPACmdNew: create a new XPA public access point for commands

```
#include <xpa.h>

XPA XPACmdNew(char *class, char *name);
```

Create a new XPA public access point for commands that will share a common identifier class:name. Enter this access point into the XPA name server, so that it can be accessed by external processes. XPACmdNew() returns an XPA struct.

It often is more convenient to have one public access point that can manage a number of commands, rather than having individual access points for each command. For example, it is easier to command the ds9 image display using:

```
echo "colormap I8"   | xpaset ds9
echo "scale log"     | xpaset ds9
echo "file foo.fits" | xpaset ds9
```

then to use:

```
echo "I8"       | xpaset ds9_colormap
echo "log"      | xpaset ds9_scale
echo "foo.fits" | xpaset ds9_file
```

In the first case, the commands remain the same regardless of the target XPA name. In the second case, the command names must change for each instance of ds9. That is, if a second instance of ds9 called DS9 were running, it would be commanded either as:

```
echo "colormap I8"   | xpaset DS9
echo "scale log"     | xpaset DS9
echo "file foo.fits" | xpaset DS9
```

or as:

```
echo "I8"       | xpaset DS9_colormap
echo "log"      | xpaset DS9_scale
echo "foo.fits" | xpaset DS9_file
```

Thus, in cases where a program is going to manage many commands, it generally is easier to define them as commands associated with the XPACmdNew() routine, rather than as separate access points using XPANew().

When XPACmdNew() is called, only the class:name identifier is specified. Each sub-command is subsequently defined using the XPACmdAdd() routine.

# XPACmdAdd: add a command to an XPA command public access point

```
#include <xpa.h>

XPACmd XPACmdAdd(XPA xpa, char *name, char *help,
                 int (*send_callback)(),
                 void *send_data, char *send_mode,
                 int (*rec_callback)(),
                 void *rec_data,  char *rec_mode);
```

Add a command to an XPA command access point. The XPA argument specifies the XPA struct returned by a call to XPANewCmd(). The name argument is the name of the command. The other arguments function identically to the arguments in the XPANew() command, i.e., the send_callback and rec_callback routines have identical calling sequences to their XPANew() counterparts, with the exceptions noted below.

When help is requested for a command access point using:

```
xpaget -h class:name
```

all of the command help strings are listed. To get help for a given command, use:

```
xpaget -h class:name cmd
```

Also, the acl keyword in the send_mode and receive_mode strings is global to the access point, not local to the command. Thus, the value for the acl mode should be the same in all send_mode (or receive_mode) strings for each command in a command access point. (The acl for send_mode need not be the same as the acl for receive_mode, though).

# XPACmdDel: remove a command from an XPA command public access point

```
#include <xpa.h>

void XPACmdDel(XPA xpa, XPACmd cmd);
```

This routine removes a command from the list of available commands in a given XPA. That command will no longer be available for processing.

# XPAInfoNew: define an XPA info public access point

```
#include <xpa.h>

XPA XPAInfoNew(char *class, char *name,
               int (*info_callback)(),
               void *info_data, char *info_mode);
```

[NB: this is an experimental interface, new to XPA 2.0, whose value and best use is evolving.]

A program can register interest in receiving a short message about a particular topic from any other process that cares to send such a message. Neither has to be an XPA server. For example, if a user starts to work with a new image file called new.fits, she might wish to alert interested programs about this new file by sending a short message using xpainfo:

```
xpainfo IMAGEFILE /data/new.fits
```

In this example, each process that has used the XPAInfoNew() call to register interest in messages associated with the identifier IMAGEFILE will have its info_callback() executed with the following calling sequence:

```
int info_cb(void *info_data, void *call_data, char *paramlist)
{
  XPA xpa = (XPA)call_data;
}
```

The arguments passed to this routine are equivalent to those sent in the send_callback() routine. The main difference is that there is no buf sent to the info callback: this mechanism is meant for short announcement of messages of interest to many clients.

The mode string is of the form: "key1=value1,key2=value2,..." The following keywords are recognized:

```
key           value           default         explanation
------        --------        --------        -----------
acl           true/false      true            enable access control
```

Because no buf is passed to this callback, the usual buf-related keywords are not applicable here.

The information sent in the parameter list is arbitrary. However, we envision sending information such as file names or XPA access points from which to collect more data. Note that the xpainfo program and the XPAInfo() routine that cause the info_callback to execute do not wait for the callback to complete before returning.

# XPAFree: remove an XPA public access point

```
#include <xpa.h>

int XPAFree(XPA xpa);
```

Remove the specified XPA public access point from the name server and free all associated storage. Note that removal from the name server happens automatically when the process terminates, so this call is not generally needed. It is used when public access points are being defined temporarily and then destroyed when no longer needed. For example, ds9 temporarily creates a public access point when it loads a new image for display and destroys it when the image is unloaded.

# XPAMainLoop: optional main loop for XPA

```
#include <xpa.h>

void XPAMainLoop();
```

Once XPA access points have been defined, a program must enter an event loop to watch for requests from external programs. This can be done in a variety of ways, depending on whether the event loop is processing events other than XPA events. In cases where there are no non-XPA events to be processed, the program can simply call the XPAMainLoop() event loop. This loop is implemented essentially as follows (error checking is simplified in this example):

```
FD_ZERO(&readfds);
while( XPAAddSelect(NULL, &readfds) ){
  if( sgot = select(swidth, &readfds, NULL, NULL, NULL) >0 )
    XPAProcessSelect(&readfds, 0);
  else
    break;
  FD_ZERO(&readfds);
}
```

The XPAAddSelect() routine sets up the select() readfds variable so that select() will wait for I/O on all the active XPA channels. It returns the number of XPAs that are active; the loop will end when there are no active XPAs. The standard select() routine is called to wait for an external I/O request. Since no timeout struct is passed in argument 5, the select() call hangs until there is an external request. When an external I/O request is made, the XPAProcessSelect() routine is executed to process the pending requests. In this routine, the maxreq value determines how many requests will be processed: if maxreq <=0, then all currently pending requests will be processed. Otherwise, up to maxreq requests will be processed. (The most usual values for maxreq is 0 to process all requests.)

If a program has its own Unix select() loop, then XPA access points can be added to it by using a variation of the standard XPAMainLoop:

```
XPAAddSelect(xpa, &readfds);
[app-specific ...]
if( select(width, &readfds, ...) ){
  XPAProcessSelect(&readfds, maxreq);
  [app-specific ...]
  FD_ZERO(&readfds);
}
```

XPAAddSelect() is called before select() to add the access points. If the first argument is NULL, then all active XPA access points are added. Otherwise only the specified access point is added. After select() is called, the XPAProcessSelect() routine can be called to process XPA requests. Once again, the maxreq value determines how many requests will be processed: if maxreq <=0, then all currently pending requests will be processed. Otherwise, up to maxreq requests will be processed.

XPA access points can be added to Xt event loops (using XtAppMainLoop()) and Tcl/Tk event loops (using vwait and the Tk loop). When using XPA with these event loops, you only need to call:

```
int XPAXtAddInput(XtAppContext app, XPA xpa)
```

or

```
int XPATclAddInput(XPA xpa)
```

respectively before entering the loop.

# XPAPoll: execute existing XPA requests

```
#include <xpa.h>

int XPAPoll(int msec, int maxreq);
```

It is sometimes desirable to implement a polling loop, i.e., where one checks for and processes XPA requests without blocking. For this situation, use the XPAPoll() routine:

```
XPAPoll(int msec, int maxreq);
```

The XPAPoll() routine will perform XPAAddSelect() and select(), but with a timeout specified in millisecs by the msec argument. If one or more XPA requests are made before the timeout expires, the XPAProcessSelect() routine is called to process those requests. The maxreq value determines how many requests will be processed: if maxreq < 0, then no events are processed, but instead, the return value indicates the number of events that are pending. If maxreq == 0, then all currently pending requests will be processed. Otherwise, up to maxreq requests will be processed. (The most usual values for maxreq are 0 to process all requests and 1 to process one request).

# XPAAtExit: install exit handler

```
#include <xpa.h>

void XPAAtExit(void);
```

XPAAtExit() will install an exit handler using atexit() to run XPAFree on all XPA access points. This might be useful in cases where Unix sockets are being used: if an explicit call to XPAFree() is not made by the program, the Unix socket file will not be deleted immediately without an atexit handler. (NB: this call should not be made in a Tcl/Tk application. Accessing the Tcl native file system after Tcl has shut down all file systems causes the Tcl/Tl program to crash).

# XPACleanup: release reserved XPA memory

```
#include <xpa.h>

void XPACleanup(void);
```

When XPA is initialized, it allocates a small amount of memory for the access control list, temp directory path, and reserved commands. This memory is found by valgrind to be "still reachable", meaning that "your program didn't free some memory it could have". Calling the XPACleanup() routine before exiting the program will free this memory and make valgrind happy.

# XPA Server Callback Macros

```
#include <xpa.h>

xpa_class, xpa_name, xpa_method, xpa_cmdfd, xpa_datafd,
xpa_sendian, xpa_cendian
```

Server routines have access to information about the XPA being called via the following macros (each of which takes the xpa handle as an argument):

```
macro                 explanation
------                -----------
xpa_class             class of this xpa
xpa_name              name of this xpa
xpa_method            method string (inet or local connect info)
xpa_cmdfd             fd of command socket
xpa_datafd            fd of data socket
xpa_sendian           endian-ness of server ("little" or "big")
xpa_cendian           endian-ness of client ("little" or "big"
```

The argument to these macros is the call_data pointer that is passed to the server procedure. This pointer should be type case to XPA in the server routine:

```
XPA xpa = (XPA)call_data;
```

The most important of these macros is xpa_datafd(). A server routine that sets "fillbuf=false" in receive_mode or send_mode can use this macro to perform I/O directly to/from the client, rather than using buf.

The xpa_cendian and xpa_sendian macros can be used together to determine if the data transferred from the client is byte swapped with respect to the server. Values for these macros are: "little", "big", or "?". In order to do a proper conversion, you still need to know the format of the data (i.e., byte swapping is dependent on the size of the data element being converted).

# XPA Race Conditions

Potential XPA race conditions and how to avoid them.

Currently, there is only one known circumstance in which XPA can get (temporarily) deadlocked in a race condition: if two or more XPA servers send messages to one another using an XPA client routine such as XPASet(), they can deadlock while each waits for the other server to respond. (This can happen if the servers call XPAPoll() with a time limit, and send messages in between the polling call.) The reason this happens is that both client routines send a string to the other server to establish the handshake and then wait for the server response. Since each client is waiting for a response, neither is able to enter its event-handling loop and respond to the other's request. This deadlock will continue until one of the timeout periods expire, at which point an error condition will be triggered and the timed-out server will return to its event loop.

Starting with version 2.1.6, this rare race condition can be avoided by setting the XPA_IOCALLSXPA environment variable for servers that will make client calls. Setting this variable causes all XPA socket IO calls to process outstanding XPA requests whenever the primary socket is not ready for IO. This means that a server making a client call will (recursively) process incoming server requests while waiting for client completion. It also means that a server callback routine can handle incoming XPA messages if it makes its own XPA call. The semi-public routine oldvalue=XPAIOCallsXPA(newvalue) can be used to turn this behavior off and on temporarily. Passing a 0 will turn off IO processing, 1 will turn it back on. The old value is returned by the call.

By default, the XPA_IOCALLSXPA option is turned off, because we judge that the added code complication and overhead involved will not be justified by the amount of its use. Moreover, processing XPA requests within socket IO can lead to non-intuitive results, since incoming server requests will not necessarily be processed to completion in the order in which they are received.

Aside from setting XPA_IOCALLSXPA, the simplest way to avoid this race condition is to multi-process: when you want to send a client message, simply start a separate process to call the client routine, so that the server is not stopped. It probably is fastest and easiest to use fork() and then have the child call the client routine and exit. But you also can use either the system() or popen() routine to start one of the command line programs and do the same thing. Alternatively, you can use XPA's internal launch() routine instead of system(). Based on fork() and exec(), this routine is more secure than system() because it does not call /bin/sh.

Starting with version 2.1.5, you also can send an XPAInfo() message with the mode string "ack=false". This will cause the client to send a message to the server and then exit without waiting for any return message from the server. This UDP-like behavior will avoid the server deadlock when sending short XPAInfo messages.

Go to XPA Help Index

**Last updated: September 10, 2003**

# Xpaoom: What happens when XPA runs out of memory?

## Summary

When XPA can't allocate memory, it exits. You can arrange to have it call longjmp() instead.

## Description

When an XPA server or client cannot allocate memory, it will attempt to output an error message and then exit. If this is not satisfactory (e.g., perhaps your program is interactive and can recover from OOM errors), you can tell XPA to call longjmp() to go to a recovery branch. To pass the requisite jmp_buf variable to XPA, make the following call:

```
XPASaveJmp(void *env);
```

The value of env is the address of a jmp_buf variable that was previously passed to setjmp(). For example:

```
jmp_buf env;
...
if( setjmp(jmp_buf) != 0 ){
  /* out of memory -- take corrective action, if possible */
} else {
  /* save env for XPA */
  XPASaveJmp((void *)&jmp_buf);
}
// enter main loop ...
```

Go to XPA Help Index

**Last updated: April 7, 2009**

# XPAClient: The XPA Client-side Programming Interface

## Summary

A description of the XPA client-side programming interface.

## Introduction to XPA Client Programming

Sending/receiving data to/from an XPA access point is easy: you generally only need to call the XPAGet() or XPASet() subroutines.

```
#include <xpa.h>

int XPAGet(XPA xpa,
    char *template, char *paramlist, char *mode,
    char **bufs, size_t *lens, char **names, char **messages, int n);

int XPASet(XPA xpa,
    char *template, char *paramlist, char *mode,
    char *buf, size_t len, char **names, char **messages, int n);

int XPAInfo(XPA xpa,
    char *template, char *paramlist, char *mode,
    char **names, char **messages, int n);

int XPAAccess(XPA xpa,
    char *template, char *paramlist, char *mode,
    char **names, char **messages, int n);

int XPAGetFd(XPA xpa,
    char *template, char *paramlist, char *mode,
    int *fds, char **names, char **messages, int n);

int XPASetFd(XPA xpa,
    char *template, char *paramlist, char *mode,
    int fd, char **names, char **messages, int n);

XPA XPAOpen(char *mode);

void XPAClose(XPA xpa);

int XPANSLookup(XPA xpa,
    char *template, char *type,
    char ***classes, char ***names, char ***methods, char ***infos);
```

## Introduction

To use the XPA application programming interface, a software developer generally will include the xpa.h definitions file:

```
#include <xpa.h>
```

in the software module that defines or accesses an XPA access point and then will link against the libxpa.a library:

```
gcc -o foo foo.c libxpa.a
```

XPA has been compiled using both C and C++ compilers.

Client communication with XPA public access points generally is accomplished using XPAGet() or XPASet() within a program (or xpaget and xpaset at the command line). Both routines require specification of the name of the access point. If a template is used to specify the access point name (e.g., "ds9*"), then communication will take place with all servers matching that template.

# XPAGet: retrieve data from one or more XPA servers

```
#include <xpa.h>

int XPAGet(XPA xpa,
           char *template, char *paramlist, char *mode,
           char **bufs, size_t *lens, char **names, char **messages,
           int n);
```

Retrieve data from one or more XPA servers whose class:name identifier matches the specified template.

A template of the form "class1:name1" is sent to the XPA name server, which returns a list of at most n matching XPA servers. A connection is established with each of these servers and the paramlist string is passed to the server as the data transfer request is initiated. If an XPA struct is passed to the call, then the persistent connections are updated as described above. Otherwise, temporary connections are made to the servers (which will be closed when the call completes).

The XPAGet() routine then retrieves data from at most n XPA servers, places these data into n allocated buffers and places the buffer pointers in the bufs array. The length of each buffer is stored in the lens array. A string containing the class:name and ip:port is stored in the name array. If a given server returned an error or the server callback sends a message back to the client, then the message will be stored in the associated element of the messages array. NB: if specified, the name and messages arrays must be of size n or greater.

The returned message string will be of the form:

```
XPA$ERROR error-message (class:name ip:port)
```

or

```
XPA$MESSAGE message (class:name ip:port)
```

Note that when there is an error stored in an messages entry, the corresponding bufs and lens entry may or may not be NULL and 0 (respectively), depending on the particularities of the server.

The return value will contain the actual number of servers that were processed. This value thus will hold the number of valid entries in the bufs, lens, names, and messages arrays, and can be used to loop through these arrays. In names and/or messages is NULL, no information is passed back in that array.

The bufs, names, and messages arrays should be freed upon completion (if they are not NULL);

The mode string is of the form: "key1=value1,key2=value2,..." The following keywords are recognized:

| key | value | default | explanation |
|---|---|---|---|
| ack | true/false | true | if false, don't wait for ack from server (after callback completes) |
| doxpa | true/false | true | client processes xpa requests |

The ack keyword is not very useful, since the server completes the callback in order to return the data anyway. It is here for completion (and perhaps for future usefulness).

Normally, an XPA client will process incoming XPA server requests while awaiting the completion of the client request. Setting this variable to "false" will prevent XPA server requests from being processed by the client.

**Example:**

```
#include <xpa.h>

#define NXPA 10
int  i, got;
size_t  lens[NXPA];
char *bufs[NXPA];
char *names[NXPA];
char *messages[NXPA];
got = XPAGet(NULL, "ds9", "file", NULL, bufs, lens, names, messages,
NXPA);
for(i=0; i<got; i++){
  if( messages[i] == NULL ){
    /* process buf contents */
    ProcessImage(bufs[i], ...);
    free(bufs[i]);
  }
  else{
    /* error processing */
    fprintf(stderr, "ERROR: %s (%s)\n", messages[i], names[i]);
  }
  if( names[i] )
    free(names[i]);
  if( messages[i] )
    free(messages[i]);
}
```

# XPASet: send data to one or more XPA servers

```
#include <xpa.h>
```

```
int XPASet(XPA xpa,
           char *template, char *paramlist, char *mode,
           char *buf, size_t len, char **names, char **messages,
           int n);
```

Send data to one or more XPA servers whose class:name identifier matches the specified template.

A template of the form "class1:name1" is sent to the XPA name server, which returns a list of at most n matching XPA servers. A connection is established with each of these servers and the paramlist string is passed to the server as the data transfer request is initiated. If an XPA struct is passed to the call, the persistent connections are updated as described above. Otherwise, temporary connections are made to the servers (which will be closed when the call completes).

The XPASet() routine transfers data from buf to the XPA servers. The length of buf (in bytes) should be placed in the len variable.

A string containing the class:name and ip:port of each of these server is returned in the name array. If a given server returned an error or the server callback sends a message back to the client, then the message will be stored in the associated element of the messages array. NB: if specified, the name and messages arrays must be of size n or greater.

The returned message string will be of the form:

```
XPA$ERROR   [error] (class:name ip:port)
```

or

```
XPA$MESSAGE [message] (class:name ip:port)
```

The return value will contain the actual number of servers that were processed. This value thus will hold the number of valid entries in the names and messages arrays, and can be used to loop through these arrays. In names and/or messages is NULL, no information is passed back in that particular array.

The mode string is of the form: "key1=value1,key2=value2,..." The following keywords are recognized:

```
key          value          default        explanation
------       --------       --------       -----------
ack          true/false     true           if false, don't wait for ack from server (after callback completes)
verify       true/false     false          send buf from XPASet[Fd] to stdout
doxpa        true/false     true           client processes xpa requests
```

The ack keyword is useful in cases where one does not want to wait for the server to complete, e.g. if a lot of processing needs to be done by the server on the passed data or when the success of the server operation is not relevant to the client.

Normally, an XPA client will process incoming XPA server requests while awaiting the completion of the client request. Setting this variable to "false" will prevent XPA server requests from being processed by the client.

**Example:**

```
#include <xpa.h>

#define NXPA 10
int  i, got;
size_t  len;
char *buf;
char *names[NXPA];
char *messages[NXPA];
...
[fill buf with data and set len to the length, in bytes, of the data]
...
/* send data to all access points */
got = XPASet(NULL, "ds9", "fits", NULL, buf, len, names, messages, NXPA);
/* error processing */
for(i=0; i<got; i++){
  if( messages[i] ){
    fprintf(stderr, "ERROR: %s (%s)\n", messages[i], names[i]);
  }
  if( names[i] )    free(names[i]);
  if( messages[i] ) free(messages[i]);
}
```

# XPAInfo: send short message to one or more XPA servers

```
#include <xpa.h>

int XPAInfo(XPA xpa,
            char *template, char *paramlist, char *mode,
            char **names, char **messages, int n);
```

Send a short paramlist message to one or more XPA servers whose class:name identifier matches the specified template.

A template of the form "class1:name1" is sent to the XPA name server, which returns a list of at most n matching XPA servers. A connection is established with each of these servers and the paramlist string is passed to the server as the data transfer request is initiated. If an XPA struct is passed to the call, then the persistent connections are updated as described above. Otherwise, temporary connections are made to the servers (which will be closed when the call completes).

The XPAInfo() routine does not send data from a buf to the XPA servers. Only the paramlist is sent. The semantics of the paramlist is not formalized, but at a minimum is should tell the server how to get more information. For example, it might contain the class:name of the XPA access point from which the server (acting as a client) can obtain more info using XPAGet.

A string containing the class:name and ip:port of each server is returned in the name array. If a given server returned an error or the server callback sends a message back to the client, then the message will be stored in the associated element of the messages array. The returned message string will be of the form:

```
XPA$ERROR   error-message (class:name ip:port)
```

or

```
XPA$MESSAGE message     (class:name ip:port)
```

The return value will contain the actual number of servers that were processed. This value thus will hold the number of valid entries in the names and messages arrays, and can be used to loop through these arrays. In names and/or messages is NULL, no information is passed back in that array.

The following keywords are recognized:

```
key           value           default         explanation
------        --------        --------        -----------
ack           true/false      true            if false, don't wait for ack from server
```

When ack is false, XPAInfo() will not wait for an error return from the XPA server. This means, in effect, that XPAInfo will send its paramlist string to the XPA server and then exit: no information will be sent from the server to the client. This UDP-like behavior is essential to avoid race conditions in cases where XPA servers are sending info messages to other servers. If two servers try to send each other an info message at the same time and then wait for an ack, a race condition will result and one or both will time out.

**Example:**

```
(void)XPAInfo(NULL, "IMAGE", "ds9 image", NULL, NULL, NULL, 0);
```

# XPAGetFd: retrieve data from one or more XPA servers and write to files

```
#include <xpa.h>

int XPAGetFd(XPA xpa,
             char *template, char *paramlist, char *mode,
             int *fds, char **names, char **messages, int n);
```

Retrieve data from one or more XPA servers whose class:name identifier matches the specified template and write it to files associated with one or more standard I/O fds (i.e, handles returned by open()).

A template of the form "class1:name1" is sent to the XPA name server, which returns a list of at most ABS(n) matching XPA servers. A connection is established with each of these servers and the paramlist string is passed to the server as the data transfer request is initiated. If an XPA struct is passed to the call, then the persistent connections are updated as described above. Otherwise, temporary connections are made to the servers (which will be closed when the call completes).

The XPAGetFd() routine then retrieves data from the XPA servers, and write these data to the fds associated with one or more fds (i.e., results from open). Is n is positive, then there will be n fds and the data from each server will be sent to a separate fd. If n is negative, then there is only 1 fd and all data is sent to this single fd. (The latter is how xpaget is implemented.)

A string containing the class:name and ip:port is stored in the name array. If a given server returned an error or the server callback sends a message back to the client, then the message will be stored in the associated element of the messages array. NB: if specified, the name and messages arrays must be of size

n or greater.

The returned message string will be of the form:

```
XPA$ERROR   error-message (class:name ip:port)
```

or

```
XPA$MESSAGE message     (class:name ip:port)
```

Note that when there is an error stored in an messages entry, the corresponding bufs and lens entry may or may not be NULL and 0 (respectively), depending on the particularities of the server.

The return value will contain the actual number of servers that were processed. This value thus will hold the number of valid entries in the bufs, lens, names, and messages arrays, and can be used to loop through these arrays. In names and/or messages is NULL, no information is passed back in that array.

The mode string is of the form: "key1=value1,key2=value2,..." The following keywords are recognized:

```
key          value          default        explanation
------       --------       --------       -----------
ack          true/false     true           if false, don't wait for ack from server (after callback completes)
```

The ack keyword is not very useful, since the server completes the callback in order to return the data anyway. It is here for completion (and perhaps for future usefulness).

**Example:**

```
#include <xpa.h>
#define NXPA 10
int  i, got;
int fds[NXPA];
char *names[NXPA];
char *messages[NXPA];
for(i=0; i<NXPA; i++)
  fds[i] = open(...);
got = XPAGetFd(NULL, "ds9", "file", NULL, fds, names, messages, NXPA);
for(i=0; i<got; i++){
  if( messages[i] != NULL ){
    /* error processing */
    fprintf(stderr, "ERROR: %s (%s)\n", messages[i], names[i]);
  }
  if( names[i] )
    free(names[i]);
  if( messages[i] )
    free(messages[i]);
}
```

# XPASetFd: send data from stdin to one or more XPA servers

```
#include <xpa.h>
```

```
int XPASetFd(XPA xpa,
             char *template, char *paramlist, char *mode,
             int fd, char **names, char **messages, int n)
```

Read data from a standard I/O fd and send it to one or more XPA servers whose class:name identifier matches the specified template.

A template of the form "class1:name1" is sent to the XPA name server, which returns a list of at most n matching XPA servers. A connection is established with each of these servers and the paramlist string is passed to the server as the data transfer request is initiated. If an XPA struct is passed to the call, then the persistent connections are updated as described above. Otherwise, temporary connections are made to the servers (which will be closed when the call completes).

The XPASetFd() routine then reads bytes from the specified fd until EOF and sends these bytes to the XPA servers. The final parameter n specifies the maximum number of servers to contact. A string containing the class:name and ip:port of each server is returned in the name array. If a given server returned an error, then the error message will be stored in the associated element of the messages array. NB: if specified, the name and messages arrays must be of size n or greater.

The return value will contain the actual number of servers that were processed. This value thus will hold the number of valid entries in the names and messages arrays, and can be used to loop through these arrays. In names and/or messages is NULL, no information is passed back in that array.

The mode string is of the form: "key1=value1,key2=value2,..." The following keywords are recognized:

| key | value | default | explanation |
|---|---|---|---|
| ack | true/false | true | if false, don't wait for ack from server (after callback completes) |
| verify | true/false | false | send buf from XPASet[Fd] to stdout |

The ack keyword is useful in cases where one does not want to wait for the server to complete, e.g. is a lot of processing needs to be done on the passed data or when the success of the server operation is not relevant to the client.

**Example:**

```
#include <xpa.h>

#define NXPA 10
int  i, got;
int fd;
char *names[NXPA];
char *messages[NXPA];
fd = open(...);
got = XPASetFd(NULL, "ds9", "fits", NULL, fd, names, messages, NXPA);
for(i=0; i<got; i++){
  if( messages[i] != NULL ){
    /* error processing */
    fprintf(stderr, "ERROR: %s (%s)\n", messages[i], names[i]);
  }
  if( names[i] )
```

```
    free(names[i]);
  if( messages[i] )
    free(messages[i]);
}
```

# XPAOpen: allocate a persistent client handle

```
#include <xpa.h>

XPA XPAOpen(char *mode);
```

XPAOpen() allocates a persistent XPA struct that can be used with calls to XPAGet(), XPASet(), XPAInfo(), XPAGetFd(), and XPASetFd(). Persistence means that a connection to an XPA server is not closed when one of the above calls is completed but will be re-used on successive calls. Using XPAOpen() therefore saves the time it takes to connect to a server, which could be significant with slow connections or if there will be a large number of exchanges with a given access point. The mode argument currently is ignored ("reserved for future use").

An XPA struct is returned if XPAOpen() was successful; otherwise NULL is returned. This returned struct can be passed as the first argument to XPAGet(), etc. Those calls will update the list of active XPA connections. Already connected servers (from a previous call) are left connected and new servers also will be connected. Old servers (from a previous call) that are no longer needed are disconnected. The connected servers will remain connected when the next call to XPAGet() is made and connections are once again updated.

**Example:**

```
#include <xpa.h>

XPA xpa;
xpa = XPAOpen(NULL);
```

# XPAClose: close a persistent XPA client handle

```
#include <xpa.h>

void XPAClose(XPA xpa);
```

XPAClose closes the persistent connections associated with this XPA struct and frees all allocated space. It also closes the open sockets connections to all XPA servers that were opened using this handle.

**Example:**

```
#include <xpa.h>

XPA xpa;
XPAClose(xpa);
```

# XPANSLookup: lookup registered XPA access points

```
#include <xpa.h>

int XPANSLookup(XPA xpa,
                char *template, char type,
                char ***classes, char ***names,
                char ***methods, char ***infos)
```

XPA routines act on a class:name identifier in such a way that all access points that match the identifier are processed. It is sometimes desirable to choose specific access points from the candidates that match the template. In order to do this, the XPANSLookup routine can be called to return a list of matches, so that specific class:name instances can then be fed to XPAGet(), XPASet(), etc.

The first argument is an optional XPA struct. If non-NULL, the existing name server connection associated with the specified xpa is used to query the xpans name server for matching templates. Otherwise, a new (temporary) connection is established with the name server.

The second argument to XPANSLookup is the class:name template to match.

The third argument for XPANSLookup() is the type of access and can be any combination of:

```
type          explanation
------        -----------
g             xpaget calls can be made on this access point
s             xpaset calls can be made on this access point
i             xpainfo calls can be made on this access point
```

The call typically specifies only one of these at a time.

The final arguments are pointers to arrays that will be filled in and returned by the name server. The name server will allocate and return arrays filled with the classes, names, and methods of all XPA access points that match the template and have the specified type. Also returned are info strings, which generally are used internally by the client routines. These can be ignored (but the strings must be freed). The function returns the number of matches. The returned value can be used to loop through the matches: **Example:**

```
#include <xpa.h>

char **classes;
char **names;
char **methods;
char **infos;
int i, n;
n = XPANSLookup(NULL, "foo*", "g", &classes, &names, &methods, &infos);
for(i=0; i<n; i++){
  [more specific checks on possibilities ...]
  [perhaps a call to XPAGet for those that pass, etc. ...]
  /* don't forget to free alloc'ed strings when done */
  free(classes[i]);
  free(names[i]);
  free(methods[i]);
  free(infos[i]);
}
```

```
/* free up arrays alloc'ed by names server */
if( n > 0 ){
  free(classes);
  free(names);
  free(methods);
  free(infos);
}
```

The specified template also can be a host:port specification, for example:

```
myhost:12345
```

In this case, no connection is made to the name server. Instead, the call will return one entry such that the ip array contains the ip for the specified host and the port array contains the port. The class and name entries are set to the character "?", since the class and name of the access point are not known.

# XPAAccess: return XPA access points matching template (XPA 2.1 and above)

```
#include <xpa.h>

int XPAAccess(XPA xpa,
              char *template, char *paramlist, char *mode,
              char **names, char **messages, int n);
```

The XPAAccess routine returns the public access points that match the specified second argument template and have the specified access type.

A template of the form "class1:name1" is sent to the XPA name server, which returns a list of at most n matching XPA servers. A connection is established with each of these servers and the paramlist string is passed to the server as the data transfer request is initiated. If an XPA struct is passed to the call, then the persistent connections are updated as described above. Otherwise, temporary connections are made to the servers (which will be closed when the call completes).

The XPAAccess() routine retrieves names from at most n XPA servers that match the specified template and that were checked for access using the specified mode. The return string contains both the class:name and ip:port. If a given server returned an error or the server callback sends a message back to the client, then the message will be stored in the associated element of the messages array. NB: if specified, the name and messages arrays must be of size n or greater.

The returned message string will be of the form:

```
XPA$ERROR error-message (class:name ip:port)
```

Note that names of matching registered access points are always returned but may not be valid; it is not sufficient to assume that the returned number of access points is the number of valid access points. Rather, it is essential to check the messages array for error messages. Any string in the messages array is an error message and indicated that the associated access point is not available.

For example, assume that a server registers a number of access points but delays entering its event loop. If a call to XPAAccess() is made before the event loop is entered, the call will timeout (after waiting for the long timeout period) and return an error of the form:

```
XPA$ERROR: timeout waiting for server authentication (XPA:xpa1)
```

The error means that the XPA access point has been registered but is not yet available (because events are not being processed). When the server finally enters its event loop, subsequent calls to XPAAccess() will return successfully.

NB: This routine only works with XPA servers built with XPA 2.1.x and later. Servers with older versions of XPA will return the error message: XPA$ERROR invalid xpa command in initialization string If you get this error message, then the old server actually is ready for access, since it got to the point of fielding the query! The xpaaccess program, for example, ignores this message in order to work properly with older servers.

The third argument for XPAAccess() is the type of access and can be any combination of:

| type | explanation |
|---|---|
| g | xpaget calls can be made on this access point |
| s | xpaset calls can be made on this access point |
| i | xpainfo calls can be made on this access point |

The mode string argument is of the form: "key1=value1,key2=value2,..." The following keywords are recognized:

| key | value | default | explanation |
|---|---|---|---|
| ack | true/false | true | if false, don't wait for ack from server (after callback completes) |

The ack keyword is not very useful, since the server completes the callback in order to return the data anyway. It is here for completion (and perhaps for future usefulness).

Go to XPA Help Index

## Last updated: March 10, 2007

# XPAXt: the XPA Interface to Xt (X Windows)

## Summary

Describes how XPA access points can be added to X Toolkit (Xt) programs.

## Description

XPA supports Xt programs: you can call XPANew(), XPACmdNew(), or XPAInfoNew() within any C routine to add XPA server callbacks to an Xt program. Since an Xt program has its own event loop call (i.e., XtAppMainLoop()), it therefore does not need or want to use the XPA even loop. Thus, in order to add XPA access points to the standard Xt event loop, the following routine should be called before entering the loop:

```
int XPAXtAddInput(XtAppContext app, XPA xpa)
```

The XPAAddAddInput() routine will add XPA access points to the Xt event loop by making calls to the standard XtAppAddInput() routine. (If the XtAppContext argument is NULL, then the alternate XtAddInput() routine is used instead.) If the xpa argument is NULL, then all active XPA access points are added to the loop. If xpa is not NULL, then only the specified access point is added. The latter type of call is used to add new access points from within a callback, after the program has entered the XtAppMainLoop() even loop.

Go to XPA Help Index

**Last updated: September 10, 2003**

# XPATcl: the XPA Interface to the Tcl/Tk Environment

## Summary

Tcl/Tk programs can act as XPA clients and/or servers using the Tcl interface to XPA that is contained in the libtclxpa.so shared object.

## Server Routines

```
set xpa [xpanew class name help sproc sdata smode rproc rdata rmode]
xpafree xpa
set xpa [xpanew class name help iproc idata imode]
set xpa [xpacmdnew class name]
xpacmdadd xpa name help sproc sdata smode rproc rdata rmode
xpacmddel xpa cmd
set val [xparec xpa option]
  options: name, class, method, cmdfd, datafd, cmdchan, datachan
xpasetbuf xpa buf len
xpaerror xpa message
xpamessage xpa message
```

## Client Routines

```
set xpa [xpaopen mode]
xpaclose xpa
set got [xpaget xpa template paramlist mode bufs lens names errs n]
set got [xpaget xpa template paramlist mode chans names errs n]
set got [xpaset xpa template paramlist mode buf len names errs n]
set got [xpasetfd xpa template paramlist mode chan names errs n]
set got [xpainfo xpa template paramlist mode names errs n]
# NB: 2.1 calling sequence change
# set got [xpaaccess template type] (2.0.5)
set got [xpaaccess xpa template paramlist mode names errs n]
set got [xpanslookup template type classes names methods]
```

## Description

You can call XPANew(), XPACmdNew(), or XPAInfoNew() within a C routine to add C-based XPA server callbacks to a TCL/Tk program that uses a Tcl/Tk event loop (either vwait() or the Tk event loop); Such a program does not need or want to use the XPA event loop. Therefore, in order to add XPA access points to the Tcl/Tk loop, the following routine should be called beforehand:

```
int XPATclAddInput(XPA xpa);
```

Normally, the xpa argument is NULL, meaning that all current XPA access points are registered with the event loop. However, if a single XPA access point is to be added (i.e., after the event loop is started) then the handle of that XPA access point can be passed to this routine.

The significance of the XPA/TCL interface goes beyond the support for using XPA inside C code. The interface allows you to write XPA servers and to make calls to the XPA client interface within the Tcl environment using the Tcl language directly. The XPA/Tcl interface can be loaded using the following package command:

```
package require tclxpa 2.0
```

Alternatively, you can load the shared object (called libtclxpa.so ) directly:

```
load .../libtclxpa.so tclxpa
```

Once the tclxpa package is loaded, you can use Tcl versions of XPA routines to define XPA servers or make client XPA calls. The interface for these routines is designed to match the Unix XPA interface as nearly as possible. Please refer to XPA Servers and XPA Clients for general information about these routines.

The file test.tcl in the XPA source directory gives examples for using the XPA/Tcl interface.

The following notes describe the minor differences between the interfaces.

# XPANew

```
set xpa [xpanew class name help sproc sdata smode rproc rdata rmode]
```

rproc and sproc routines are routines. The calling sequence of the rproc routine is identical to its C counterpart:

```
proc rec_cb { xpa client_data paramlist buf len } { ... }
```

The sproc routine, however is slightly different from its C counterpart because of the difficulty of passing data back from the callback to C:

```
proc sendcb { xpa client_data paramlist } { ... }
```

Note that the C-based server's char **buf and int *len arguments are missing from the Tcl callback. This is because we did not know how to fill buf with data and pass it back to the C routines for communication with the client. Instead, the Tcl server callback uses the following routine to set buf and len:

```
xpasetbuf xpa buf len
```

where:

```
arg           explanation
------        -----------
xpa           the first argument of the server callback
buf           the data to be returned to the client
len           data length in bytes, (if absent, use length of the buf object)
```

When this routine is called, a copy of buf is saved for transmission to the client.

The fact that buf is duplicated means that TCL server writers might wish to perform the I/O directly within the callback, rather than have XPA do it automatically at the end of the routine. To do this, set:

```
fillbuf=false
```

in the xpanew smode and then perform I/O through the Tcl channel obtained from:

```
set dchan [xparec $xpa datachan]
```

where:

```
arg           explanation
------        -----------
xpa           the first argument of the server callback
datachan      literal string "datachan" that returns the data channel
len           data length in bytes, (if absent, use length of the buf object)
```

**NB: datachan and cmdchan are not available under Windows. It is necessary to use the "raw" equivalents: datafd and cmdfd.**

The same considerations apply to the rproc for receive servers: a copy of the incoming data is generated to pass to the receive callback. This copy again can be avoided by using "fillbuf=false" in the rmode and then reading the incoming data from datachan.

The send and receive callback routines can use the xpaerror and xpamessage routines to send errors and messages back to the client. If you also want tcl itself to field an error condition, use the standard return call:

```
return ?-code c? ?-errorinfo i? ?-errorcode ec? string
```

See the Tcl man page for more info.

# XPARec

The Tcl xparec procedure supplies server routines with access to information that is available via macros in the C interface:

```
set val [xparec xpa <option>]
```

where option is: name, class, method, cmdfd, datafd, cmdchan, datachan. Note that two additional identifiers, cmdchan and datachan, have been added to to provide Tcl channels corresponding to datafd and cmdfd. (These latter might still be retrieved in Tcl and passed back to a C routines.) An additional option called "version" can be used to determine the XPA version used to build the Tcl interface. Note that the standard options require a valid XPA handle, but "version" does not (since it simply reports the value of the XPA_VERSION definition in the XPA source include file).

**NB: datachan and cmdchan are not available under Windows. It is necessary to use the "raw" equivalents: datafd and cmdfd.**

```
macro         explanation
------        -----------
class         class of this xpa
name          name of this xpa
method        method string (inet or local connect info)
cmdchan       Tcl channel of command socket
datachan      Tcl channel of data socket
cmdfd         fd of command socket
datafd        fd of data socket
sendian       endian-ness of server ("little" or "big")
cendian       endian-ness of client ("little" or "big"
version       XPA version used to build this code
```

Under Windows, the Tcl event handler cannot automatically sense when an XPA socket is ready for IO (i.e. Tcl_CreateFileHandler() is not available under Windows). The Windows Tcl event handler therefore must be awakened occasionally for check for XPA events. This is done using the standard Tcl_SetMaxBlockTime() call. The time parameter is defined in tclloop.c and is currently set to 1000 microseconds (1/1000 of a second).

The version option can be used to differentiate between source code versions. It was created to support legacy Tcl code that needs to maintain the 2.0.5 calling sequence for xpaaccess. You can use a version test such as:

```
if [catch { xparec "" version } version] {
  puts "pre-2.1.0e"
} else {
  puts [split $version .]
}
```

Go to XPA Help Index

## Last updated: September 10, 2003

# XPAEnv: Environment Variables for XPA Messaging

## Summary

Describes the environment variables which can be used to tailor the overall XPA environment.

## Description

The following environment variables are supported by XPA:

**XPA_ACL**
If *XPA_ACL* is *true*, then host-based XPA Access Control is turned on and only specified machines can access specified access points. If *false*, then access control is turned off and any machine can access point. The default is turn turn access control on.

**XPA_ACLFILE**
If XPA Access Control is turned on, this variable specifies the name of the file containing access control information for all access points started by this user. The default file name is: *$HOME/acls.xpa*.

**XPA_CONNECT_TIMEOUT**
When an XPA server first starts up, it immediately tries to connect to the XPA name server program (xpans) on the host specified by the *XPA_NSINET* variable. (If this connection fails on the local host, and if xpans can be found in the path, then the name server is started automatically.) Unfortunately, a mis-configured network can cause this connect attempt to hang for many seconds while the connect() system call times out. Therefore, an alarm is started to interrupt the connect() call and prevent a long hang. The initial value of the alarm timeout is 10 seconds, but can be changed by setting this environment variable. If you want to disable the alarm and allow the initial connect() to time out, set the value of this variable to 0. Normally, users would not change this variable at all.

**XPA_CLIENT_DOXPA**
Normally, an XPA client (xpaget, xpaset, etc.) will process incoming XPA server requests while awaiting the completion of the client request. Setting this variable to "false" will prevent XPA server requests from being processed by the client.

**XPA_DEFACL**
If XPA Access Control is turned on, this variable specifies the default access control condition for all access points, if the *XPA_ACLFILE* file does not exist. The default acl is: *$host:* $host* +, meaning that all processes on the host machine have full access to all access points.

**XPA_HOST**
For the INET socket method, XPA utilizes the canonical hostname (as returned by the gethostname() routine) to construct the IP part of the method id. Under some circumstances, this might not be a correct choice of name and IP. For example, if an XPA server is started on a machine running VPN, you might want to use the VPN name and IP instead of the canonical host name, so that other machines in the VPN network can access the server. In this case, you can set the XPA_HOST to be

the VPN name (if resolvable) or, more easily, the VPN IP.

**XPA_IOCALLSXPA**

Setting this variable causes all XPA socket IO calls to process outstanding XPA requests whenever the primary socket is not ready for IO. This means that a server making a client call will (recursively) process incoming server requests while waiting for client completion. This inter-IO XPA processing avoids a rare XPA Race Condition: two or more XPA servers sending messages to one another using an XPA client routine such as XPASet() can deadlock while each waits for the other server to respond. This can happen, for example, if the servers call XPAPoll() with a time limit, and send messages in between the polling call.

By default, this option is turned off, because we judge that the added code complication and overhead involved will not be justified by the amount of its use. Moreover, processing XPA requests within socket IO can lead to non-intuitive results, since incoming server requests will not necessarily be processed to completion in the order in which they are received.

**XPA_LOGNAME**

XPA preferentially uses the de facto standard environment variable LOGNAME to determine the username when registering an access point in the name server. If this environment variable has been used for something other than the actual user name (such as a log file name), unexpected results can ensue. In such cases, use the XPA_LOGNAME variable to set the user name. (If neither exists, then getpwuid(geteuid()) is used as a last resort).

**XPA_LONG_TIMEOUT**

XPA is designed to allow data to be sent from one process to another over a long period of time (i.e., a program that generates image data sends that data to an image display, but slowly) but it also seeks to prevent hangs. This is done by supporting 2 timeout periods: a *short* timeout for protocol communication and a *long* for data communication.

The *XPA_LONG_TIMEOUT* variable controls the *long* timeout and is used to prevent hangs in cases where communication between the client and server that is *not* controlled by the XPA interface itself. Transfer of data between client and server, or a client's wait for a status message after completion of the server callback, are two examples of this sort of communication. By default, the *long* timeout is set to 180 seconds. Setting the value to -1 will disable *long* timeouts and allow an infinite amount of time.

**XPA_MAXHOSTS**

The maximum number of access points that the programs *xpaset*, *xpaget*, and *xpainfo* will communicate with at one time. The default is 64, meaning, for example, that the *xpaset* program will not send a message to more than 100 access points at one time and *xpaget* will not retrieve from more than 100 access points at one time.

**XPA_METHOD**

Determines the socket connection method used by this session of XPA. The choices are: *inet* (to use INET or Internet-based sockets), *localhost* (to use the machines localhost inet socket), or *local (unix)* (to use UNIX sockets). The default is *INET*. Using the *inet* method will allow access from other machines (subject to access controls) but using *localhost* or *local* will not. Localhost is most useful for private access and when the machine in question is not connected to the Internet. The unix

method also can be used for private access and non-Internet connections (Unix platforms only).

Once defined, the first registration of an XPA access point will ensure that an instance of the XPA Name Server (xpans) is running that handles that connection method. All new access points will use the new connection method but existing access points will use the original method.

**XPA_NSINET**

For the *inet* method of socket connection, this variable specifies the host and port on which the XPA Name Server (xpans) is listens for new access points. The default is *$host:$port*, meaning that the default XPA port (14285) on the current machine (as returned by gethostname()) is used. If several machines were all accessing the same XPA access points, you would use this variable to specify that they all use the same name server to find out about these access points. For example, a value of *myhost:$port* would mean that the xpans name server is running on myhost and uses the default port 12345. All machines would then get the XPA access points registered with that name server, subject to access controls.

The port used by xpans to register its XPA access point normally is taken to be one greater than the port on which it receives new access points from XPA servers. You can specify a specific access point port using the syntax machine:port1,port2, i.e., the access point port is specified after the comma. For example, $host:12345,23456 will listen for new access ports on 12345 and will accept XPA commands on 23456.

**XPA_NSREGISTER**

This boolean variable specifies whether a server registers its XPA access point with the specified xpans name server. The default is *true*. If set to *false*, the access point still is set up but it is not registered with xpans and therefore cannot be accessed by name. (It can be accessed by method, if the latter is known.) Note that an access point can be registered later on (using -remote or -proxy, for example). This variable mainly is useful in cases where the Internet configuration is broken (so that registration causes a DNS hang) but you still wish to and can use the server with a remote xpans (e.g., ds9's Virtual Observatory capability).

**XPA_NSUNIX**

For the *local* method of socket connection, this variable specifies the name of the Unix file that will be used to access the XPA Name Server (xpans). The default is *xpans_unix*. This variable is not usually needed. Note that is the *local* socket method is used, then remote machines will not be able to access the xpans name server or the registered XPA access points.

**XPA_NSUSERS**

This variable specifies whether other users' access points will be returned by the XPA Name Server (xpans) for use by *xpaget*, *xpaset*, etc. Generally speaking, it is sufficient to run one xpans name server per machine and register the access points for all users with that xpans. This means, for example, that if you request information from ds9 by running:

```
xpaget ds9 colormap
```

you might get information from your own ds9 as well as from another user running ds9 on the same machine. The *XPA_NSUSERS* variable controls whether you want such access to the access points of other users. By default, only your own access points are returned, so that, in the example above, you

would only get the colormap information from the ds9 you registered. If, however, you had set the value of the *XPA_NSUSERS* variable to *eric,fred*, then you would be able to communicate with both eric and fred's access points. Note that this variable can be overridden using the *-u* switch on the *xpaget*, *xpaset*, and *xpainfo* programs.

**XPA_PORT**

A semi-colon delimited list of user specified ports to use for specific XPA access points. The format is each specification is:

```
class:template port1[ port2]
```

where **port1** is the main (command) port for the access point and **port2** is the (secondary) data port. If port2 is not specified, it defaults to a value of 0 (meaning the system assigns the port).

Specification of specific ports is useful, for example, when a machine outside a firewall needs to communicate with a machine inside a firewall. In such a case, the firewall should be configured to allow socket connections to both the command and data port from the outside machine, and the inside XPA program should be started up with the outside machine in its ACL list. Then, when the inside program is started with specified ports, outside XPA programs can use "machine:port" to contact the inside access points, instead of the access point names. That is, the machine outside the firewall does not need access to the XPA name server:

```
export XPA_PORT="DS9:ds9 12345 12346"   # on machine "inside"
cat foo.fits | xpaset inside:12345 fits # on machine "outside"
```

Note that 2 ports are required for full XPA communication and therefore 2 ports should be specified to go through a firewall. The second port assignment is not important if you simply are assigning the command port in order to communicate commands with a known port (e.g., to bypass the xpans name server). If only one (command) port is specified, the system will negotiate a random data port and everything will work properly.

This support is somewhat experimental. If you run into problems, please let us know.

**XPA_PORTFILE**

A list of user-specified port to use for specific xpa access points. The format of the file is:

```
class:template port1 [port2]
```

where **port1** is the main port for the access point and **port2** is the data port. If port2 is not specified, it defaults to a value of 0 (meaning the system assigns the port). See **XPA_PORT** above for an explanation of user-specified ports.

**XPA_SHORT_TIMEOUT**

XPA is designed to allow data to be sent from one process to another over a long period of time (i.e., a program that generates image data sends that data to an image display, but slowly) but it also seeks to prevent hangs. This is done by supporting 2 timeout periods: a *short* timeout for protocol communication and a *long* for data communication.

The *XPA_SHORT_TIMEOUT* variable controls the *short* timeout and is used to prevent hangs in cases where the XPA protocol requires internal communication between the client and server that is controlled by the XPA interface itself. Authentication is an example of this sort of communication, as is the establishment of a data channel between the two processes. The default value for the *short* is 30 seconds (which is a pretty long time, actually). Setting the value to -1 will disable *short* timeouts and allow an infinite amount of time.

**XPA_SIGUSR1**

If the value of this variable is *true*, then XPA will catch SIGUSR1 signals when performing an I/O operation in order to curtail that operation. This facility allows users to send a SIGUSR1 signal to an XPA server if a client is hanging up the server by sending or receiving data too slowly (timeouts also can be used -- see above). When enabled in this way, the SIGUSR1 signal is ignored at all other times, so that its safe to send the signal at any time. If the variable is set to *false*, then SIGUSR1 is not used at all. Turning off SIGUSR1 would be desired in cases there the program uses SIGUSR1 for some other reason and does not want XPA interfering. The default is to use the signal.

**XPA_TIMESTAMP_ERRORS**

If *XPA_TIMESTAMP_ERRORS* is *true*, then error messages will include a date/time string. This can be useful when XPA errors are being saved in an error log (e.g. Web/CGI use). The default is false.

**XPA_TMPDIR**

This variable specifies the directory into which XPA logs, Unix socket files (when *XPA_METHOD* is *local*), etc. are stored. The default is */tmp/.xpa*.

**XPA_VERBOSITY**

Specify the verbosity level of error messages. If the value is set to *0*, *false*, or *off*, then no error messages are printed to stderr. If the value is *1*, then important XPA error messages will be output. If the value is set to *2*, XPA warnings about out-of-sync messages will also be output. These latter almost always can be ignored.

**XPA_VERSIONCHECK**

Specify whether a new access point should check its major and minor XPA version number against the version used by the xpans name server at registration time. The default is *true*. When checking is performed, a warning is issued if the server major version is found to be greater than the xpans version. Note that the check is performed both by the XPA server and by the xpans process and warnings will be issued by each. Also, instead of the values of *true* or *false*, you can give this variable an integer value n. In this case, each version checking process (i.e., the XPA-enabled server or xpans) will print out a maximum of n warning messages (after which version warnings are silently swallowed).

In general, it is a bad idea to run an XPA-enabled server program using a version of XPA newer than the basic xpaset, xpaget, xpaaccess, xpans programs. This sort of mismatch usually will not work due to protocol changes.

Go to XPA Help Index

**Last updated: December 23, 2009**

# XPAAcl: Access Control for XPA Messaging

## Summary

XPA supports host-based access control for each XPA access point. You can enable/disable access control using the XPA_ACL environment variable. You can specify access to specific XPA access points for specific machines using the XPA_DEFACL and XPA_ACLFILE environment variables. By default, an XPA access point is accessible only to processes running on the same machine (same as X Windows).

## Description

When INET sockets are in use (the default, as specified by the *XPA_METHOD* environment variable), XPA supports a host-based access control mechanism for individual access points. This mean that access can be specified for get, set, or info operations for each access point on a machine by machine basis. For LOCAL sockets, access is restricted (by definition) to the host machine.

XPA access control is enabled by default, but can be turned off by setting the *XPA_ACL* environment variable to *false*. In this case, any process can access any XPA server.

Assuming that access control is turned on, the ACL for an individual XPA access point is set up when that access point is registered (although it can be changed later on; see below). This can be done in one of two ways: Firstly, the *XPA_ACLFILE* environment variable can defined to point to a file of access controls for individual access points. The format of this file is:

```
class:name ip acl
```

The first argument is a template that specifies the class:name of the access point covered by this ACL. See XPA Access Points and Templates for more information about xpa templates.

The second argument is the IP address (in human-readable format) of the machine which is being given access. This argument can be * to match all IP addresses. It also can be *$host* to match the IP address of the current host.

The third argument is a string combination of *s*, *g*, or *i* to allow *xpaset*, *xpaget*, or *xpainfo* access respectively. The ACL argument can be + to give *sgi* access or it can be - to turn off all access.

For example,

```
*:xpa1  somehost sg
*:xpa1  myhost +
* * g
```

will allow processes on the machine somehost to make xpaget and xpaset calls, allow processes on myhost to make any call, and allow all other hosts to make xpaget (but not xpaset) calls. Secondly, if the *XPA_ACLFILE* does not exist, then a single default value for all access points can be specified using the *XPA_DEFACL* environment variable. The default value for this variable is:

```
#define XPA_DEFACL "*:* $host +"
```

meaning that all access points are fully accessible to all processes on the current host. Thus, in the absence of any ACL environment variables, processes on the current host have full access to all access points created on that host. This parallels the X11 xhost mechanism.

Access to an individual XPA access point can be changed using the -acl parameter for that access point. For example:

```
xpaset -p xpa1 -acl "somehost -"
```

will turn off all access control for somehost to the xpa1 access point, while:

```
xpaset -p XPA:xpa1 -acl "beberly gs"
```

will give beberly xpaget and xpaset access to the access point whose class is XPA and whose name is xpa1.

Similarly, the current ACL for a given access point can be retrieved using:

```
xpaget xpa1 -acl
```

Of course, you must have xpaget access to this XPA access point to retrieve its ACL.

Note that the XPA access points registered in the *xpans* program also behave according to the ACL rules. That is, you cannot use xpaget to view the access points registered with xpans unless you have the proper ACL.

Note also when a client request is made to an XPA server, the access control is checked when the initial connection is established. This access in effect at this time remains in effect so long as the client connection is maintained, regardless of whether the access fro that XPA is changed later on.

We recognize that host-based access control is only relatively secure and will consider more stringent security (e.g., private key) in the future if the community requires such support.

Go to XPA Help Index

## Last updated: September 10, 2003

# XPA ChangeLog

This ChangeLog covers the XPA 2 implementation. It will be updated as we continue to develop and improve XPA. The up-to-date version can be found here.

# Public Release 2.1.15 (July 23, 2013)

- Added support for large data transfers
- XPAGet and XPASet now pass size_t instead of int for lengths
- Send and receive callbacks now pass size_t instead of int for lengths
- Port to mingw (Windows)

# Public Release 2.1.14 (June 7, 2012)

- Fixed several memory leaks in the Tcl wrappers (tcl.c).
- Use Tcl stubs library for linking shared Tcl, if available.

# Public Release 2.1.13 (April 14, 2011)

- An atexit handler is no longer installed automatically (it crashes Tcl 8.5.8 applications). Call XPAAtExit(void) to install the handler.
- Removed permission checking from Find() on cygwin systems. This was broken by Windows 7.
- Removed addition of -no-cpp-precomp flag from gcc 4.2 and beyond (Mac).

# Public Release 2.1.12 (January 26, 2010)

- Added XPA_HOST environment variable to allow users to specify the hostname (and hence, ip) component of the INET method id. This is useful, for example, if you want to register an access point using a VPN-generated IP instead of the canonical IP.
- Fix typo in Tcl binding to xpainfo causing a crash after 2 invocations.

# Public Release 2.1.11 (December 7, 2009)

- Generalized XPANSKeepAlive() to send messages to xpans, proxy xpans, or both. The default is to send just to proxies (e.g. chandra-ed).

- Changed XPANSKeepAlive() to send an in-band new-line char to xpans, changed xpans to handle an in-band new-line as a keep-alive message. Necessitated by Cisco routers that clear the URG flag in a TCP packet, breaking OOB data transfer for the whole Internet, as well as the OOB-based keep-alive implemented in xpans.
- In xpans, print warning when the keep-alive option switch is used.
- Port to mingw (thanks to B.Schoenhammer)
- Change OOB character sent by xpans keepalive to a space, trying to working around cisco routers that force OOB data into the inbound stream.
- fix gcc fprintf warning in xpans.c

# Public Release 2.1.10 (September 1, 2009)

- Update mklib and configure.ac to support 64-bit builds on Macs.
- Fixed bug in XPAAccess() in which the returned names could have an extra (bogus) character when the target is an explicit ip:port or local socket file.
- Add setjmp/longjmp support to xalloc.
- Add XPASaveJmp(void *env) as a high-level interface to xalloc_savejmp();

# Internal Release 2.1.9

- Fixed a bug that prevented an access point starting with a number from being recognized peoperly. NB: a pure number still signifies a port on the current machine. Also num:num signifies ip:port, where ip can be a pure hex value or the canonical form vvv.xxx.yyy.zzz.
- Modified internal Launch() routine to use posix_spawn(), if necessary. This is required for OS X 10.5 (leopard), which frowns upon use of fork() and exec(). Also modified zprocess routines to use Launch().
- Added XPASetFree(xpa, void (*myfree)(void *)) routine to allow callbacks to specify a free routine other than malloc free (e.g. Perl garbage collection).
- XPACmdAdd() now checks to ensure that it was passed an XPA struct created by XPACmdNew().
- Change launch.h to xlaunch.h to avoid conflict with OS X.

# Public Release 2.1.8 (1 November 2007)

- A public release to complete current XPA development work.

# Patch Release 2.1.7b[1,2] (Feb 22, 2006; March 8, 2007)

- Added a convenience null to the end of the buffers returned by XPAGet.
- Added code to avoid calling atexit routine if a fork'ed child calls exit() instead of _exit().
- Added XPA_CLIENT_DOXPA environment variable to turn off client processing of xpa server requests.
- Added --version to xpaset, xpaget, xpainfo, xpaaccess, xpans to display XPA version and exit.
- Added support for integrating XPA into a Gtk loop.
- xpaaccess now returns its answer in the error code as well as to stdout (without the -n switch, it returns 1 for a match, with the -n switch, the number of matches is returned).
- Fixed bug which prevented xpans from being started up automatically by an xpa server if its pathname contained a space character.
- Fixed bug in MINGW port of xpans in which an XPA server that terminated via an interrupt was not being properly removed from the list of registered access points.
- Added XPA_LOGNAME to override LOGNAME when registering username
- Upgraded swish-e indexing code to 2.4.5.

# Patch Release 2.1.6 (4 May 2005)

- Added -P switch to xpans to enable experimental proxy support (default is disabled). An argument of 1 processes proxy requests in the same thread as xpans requests, while an argument of 2 processes proxy requests in a separate thread. (The latter is recommended to avoid xpans timeouts, since xpa callback processing can take a long time.)
- Added ability to build shared libraries (done automatically with configure --enable-shared) with compilers other than gcc.
- Made yet another attempt to build shared libraries under OS X.
- Fixed a server bug in Tcl support under Windows (introduced early in 2.1.6) which caused an occasional SEGV.
- Fixed race condition in cases where 2 or more servers makes client calls to one another.
- Fixed bug in the XPA handler routine in which an access point was turned off if an error occurred in that routine (as opposed to the user-defined callback routine).

- Fixed race condition when "ack=false" flag (or -n) is used with XPASet() (or xpaset).
- Added defensive code to XPA handler to ensure that the passed XPA record is valid.
- Tcl/XPA servers such as ds9 were not turning off select() on the xpa channels inside an xpa callback, as required. This is now fixed.
- Added timestamps to most server and client error messages if the XPA_TIMESTAMP_ERRORS variable is set. This is useful when XPA errors are being logged in an error log (e.g. Web/CGI use).
- Generated PostScript and PDF versions of the help pages.
- Moved OPTIONS section before (often-lengthy) DESCRIPTION section in man pages.
- All memory allocation now performs error checking on the result.
- Removed some compiler warnings that surfaced when using gcc -O2.
- Updated configure.ac to better support Tcl in Panther with Apple Frameworks.

# Patch Release 2.1.5 (12 January 2004)

- Fixed bug in XPAPoll(). Erroneously, no requests were being processed when maxreq==0. Now, all pending events are processed, as per the documentation.
- Added ack=false to XPAInfo() (and corresponding -n to xpainfo) so that client does not wait for a response from the server. This is essential in cases where XPA servers wish to send info messages to one another without causing a race condition.
- Generated man pages from the html pages. These are installed automatically at build time.
- The xpans program with Unix sockets now uses a lock file to signal that it is running, in order to avoid a potential (but rare) race condition at startup.
- Code that calls Unix-type bind() now manipulate umask() to ensure that all users have write permissions to the socket file (OS X apparently uses these permissions while previous platforms ignore them).
- Configure now checks for socklen_t type (OS X does not define it).
- Added an atexit function to run XPAFree. The aim here is to delete Unix socket files on exiting.
- Under Windows, the Tcl event-handling code now blocks for 1/1000 of a second instead of not blocking at all (which inadvertently used 100% of cpu).
- Upgraded Tcl/Tk support to 8.4.

- Made another round of checks was made through all instances of strcat, strcpy, etc. to look for potential buffer overflows. Changed all instances of sprintf() to snprintf().
- Class and name designators are now limited to 1024 characters, for no particular reason.
- The obsolete $SAORD_BIN variable was being added to the path when searching for xpans. This is no longer the case.
- Fixed non-ANSI compiler errors in both xpa.c and xpans.c.
- Fixed minor problems to support compilation with g++.
- Ported to Intel icc and gcc 3.3 compilers.
- Upgraded autoconf to 2.57. Included in this upgrade is a change that makes gcc the default compiler (use "configure CC=cc" to change this). Also, by default, the Tcl shared object is no longer automatically built if the Tcl libraries are used. Use the --enable-tclshlib switch in configure to enable this feature.
- Changed license from public domain to GNU GPL.

# Patch Release 2.1.4 (24 March 2003)

- Made inet method unique, even when hosts are behind a firewall using the same ports (on different local machines).
- The initial connection from an xpa server to a local xpans now is controlled by a timeout (default 5 sec, controlled by XPA_CONNECT_TIMEOUT variable). This should prevent a hang on connect() if the network is not set up correctly.
- Fixed rare race condition when an XPA server callback performed its own XPAGet or XPASet call to another XPA server.
- Use POSIX O_NONBLOCK for non-blocking I/O in fcntl call if it exists, instead of O_NDELAY.

# Patch Release 2.1.3 (26 September 2002)

- Added -k [sec] switch to xpans to support sending one-byte keepalive messages from xpans to registered xpa servers.
- Added XPANSKeepAlive routine (and Tcl equivalent) to allow xpa servers to send a one-byte keepalive message to xpans.

# Patch Release 2.1.2 (18 July 2002)

- The "-help" reserved command now also displays the XPA version, if no explicit sub-commands are specified.

- Change internal state of xpans proxy to save ip:port value of a server behind a NAT firewall. This is required to give some hope of distinguishing multiple instances of ds9 running behind NAT.

# Patch Release 2.1.1 (20 June 2002)

- Added a version check between xpans and an access point, performed when it gets registered by an XPA server. If the server has a version greater than the xpans version, a warning is issued by both programs.
- Added a boolean XPA_NSREGISTER environment variable to allow an XPA server to skip xpans registration. The default is to register with the name server. If set to "false", the access point still is set up but it is not registered with an xpans. It can be registered later on (using -remote or -proxy, for example).
- Fixed bug in which xpans was still listening on any interface when XPA_METHOD was localhost (instead of just listening on localhost).

# Public Release 2.1.0 (22 April 2002)

New features include:

- Support for proxy access to XPA servers (e.g. ds9) situated behind a firewall (useful for NVO-type applications).
- Improved support for allowing remote machines access rights to the XPA access points (useful for NVO-type applications).
- Ability for XPAAccess() routine and xpaaccess program to contact XPA directly.
- Support for a clipboard access point that allows clients to store ASCII state information in an XPA-enabled server.
- Improved support for Windows platform, as well as new support for Mac OSX.

# Pre-Release 2.1.0e (2 April 2002)

- Removed the environment variable generated by each XPA access point (of the form XPA_name=method). The putenv() call was causing ds9 to crash under both Linux and LinuxPPC during a socket operation. We suspect a bug in putenv but cannot prove it and this feature is not essential, so ...

# Pre-Release 2.1.0e (1 April 2002)

- Fixed an uninitialized variable in xpamb which prevented it from working at all on some systems.

- Changed xpamb switch from "-add" to "-data" (to store named data).
- Changed how xpamb works with xpaget so that xpamb can return data from XPA access points as well as from stored data. (Previous versions only returned stored data.) Now, you can retrieve stored data explicitly using the -info and/or -data switches. For example:

  ```
  xpaget xpamb -info foo
  ```

  will return info about the previously stored data named foo. If neither switch is present, then the name is assumed to be an XPA access point.

# Pre-Release 2.1.0e (25 March 2002)

- Changed symbol for default port from "*" to "$port" to avoid a syntactical conflict between class:* and machine:* when processing an XPA access point class:name specification. Thus, the default inet method now is '$host:$port' instead of '$host:*'.

# Pre-Release 2.1.0e (19 March 2002)

- Removed timeout check when reading data (in clients using xpaget and servers filling the data buffer). We have more and more cases where we need to wait a long time to retrieve data (e.g., slow networks or receiving data being compressed on the fly).
- Moved call to sigaction(SIGCHLD,...) out of XPAOpen(), so that it is only executed when needed by XPAGet()/XPASet() routines called from within xpans/proxy. But then changed logic to use a double fork() instead of sigaction() to prevent zombies (Stevens Adv. Programming p 202).
- Each XPA access point now generates an environment variable of the form XPA_name=method so that children can communicate with the parent access point more easily.
- Added version option to Tcl xparec:

  ```
  if [catch { xparec "" version } version] {
    puts "pre-2.1.0e"
  } else {
    puts [split $version .]
  }
  ```

  to help differentiate between XPA versions within Tcl code.

# Pre-Release 2.1.0e (14 February 2002)

- Fixed client handling of out-of-sync messages.

## Pre-Release 2.1.0e (11 February 2002)

- Fixed client.c/xopen() so that it does not open an extra socket.
- Fixed xpainfo/xopen() to prevent client from hanging waiting for ack.
- Modified stest to generate xpaaccess points xpa, xpa1, c_xpa, and i_xpa (or more generally, , 1, c_, i) to allow more flexible testing of templates. Also added -a for testing XPAAccess().

## Beta Release 2.1.0b10 (31 January 2002)

- Added support for Mac OSX/Darwin to configure file.

## Beta Release 2.1.0b9 (26 January 2002)

- Fixed bug in client library that caused XPAAccess() call to hang.

## Beta Release 2.1.0b8 (4 January 2002)

- Made modifications to Makefile.in to make releases easier.
- Added instructions to Makefile.in so that xpa.h will always have correct #defines for XPA_VERSION, XPA_MAJOR_VERSION, XPA_MINOR_VERSION, and XPA_PATCH_LEVEL.

## Beta Release 2.1.0b7 (21 December 2001)

- Added -proxy switch to -remote sub-command to allow remote access through a firewall, using xpans as a proxy server. The support for proxy processing required a change to the client/server protocol. This means that new xpa servers will not work with old xpa clients (although new xpa clients will work with old xpa servers). For details about proxy firewall support, see http://hea-www.harvard.edu/RD/xpa/inet.html.
- Fixed Tcl support for XPA under Windows/Cygwin by re-writing the code used to add XPA to the Tcl event loop. This fix makes ds9 support for XPA much more stable under Windows.
- Added the shutdown() call to XPA under Cygwin/Windows before closing send() sockets. It appears that a Cygwin recv() socket call does not always sense when the other end closes the socket using close(). This measure must be considered a hack, since the actual problem was never resolved.
- Added code to protect accept() and select() calls from interrupts.
- Extended syntax of the environment variable XPA_NSINET to:

```
setenv XPA_NSINET host:port[,port[,port]]
```

to allow specification of the XPA access point port for xpans, as well as the proxy data port.

- Modified xpans log file so that it contains the xpaset commands required to reconnect with xpa servers.
- xpans now deletes its Unix socket files.

# Beta Release 2.1.0b6 (29 October 2001)

- Implemented a reserve public access point named -clipboard so that clients can store ASCII state information on any number of named clipboards. Clipboards of the same name created by clients on different machines are kept separate. The syntax for creating a clipboard is:

```
[data] | xpaset [server] -clipboard add|append [clipboard_name]
xpaset -p [server] -clipboard delete [clipboard_name]
xpaget [server] -clipboard [clipboard_name]
```

Use "add" to create a new clipboard or replace the contents of an existing one. Use "append" to append to an existing clipboard.

# Beta Release 2.1.0b5 (22 October 2001)

- Use FD_SETSIZE instead of getdtablesize() to determine how many files to check during select();
- Under Cygwin, the launch() routine now uses the Cygwin spawnvp() instead of fork()/exec() where possible (i.e., if no stdfiles are being redirected). This is recommended by Cygwin's (skimpy) on-line documentation and seems to fix the problems ds9 had when starting xpans automatically.
- Added error check to select() call in xpans.

# Beta Release 2.1.0b4 (24 September 2001)

- The launch() now can return an error code if the execv() system call fails (something system() does not do).
- INET socket calls between xpa clients and servers now will use localhost if they are on the same machine. This protects against Linux systems where the hostname is hardwired (wrongly) in a DHCP environment.

# Beta Release 2.1.0b3 (6 September 2001)

- Modified xpans so that, in the case of a firewall, it tries to correct the specified ip:port by matching against the ip found in the socket packet at accept() time.

- Replaced system() call used to start xpans automatically with a special launch() call, which performs execvp() directly without going through sh. (launch() works under DOS and has fewer security problems.)
- Fixed bug in xpans in which its xpa port was always being set to 14286.

# Beta Release 2.1.0b2 (17 August 2001)

- Added support for -remote command, which registers the access point in the XPA name server of the specified remote server, and gives the remote server access rights to the access point. This is used, for example, to give data servers xpa access to ds9 so that data can be sent to ds9 as a result of a CGI-based Web query.
- Reserved commands (except "-help" and "-version") now can only be executed on the machine on which the xpa service is running (not through -remote servers).
- Fixed bug in xpans in which a bad telnet command could hang the program.
- Added -s [security file] to xpans to allow logging of all external connections.

# Beta Release 2.1.0b1 (6 August 2001)

- The xpaaccess client program and XPAAccess() client subroutine were modified so that an access-type query can directly contact the xpa servers matching the requested xpa template, instead of just querying the name server for registered access points. This avoid the race condition in which an access point is registered but is not yet available, perhaps because the server has not yet entered its event loop. Note that the calling sequence of the XPAAccess() routine was changed to return all matching access points and their availability status (instead of just returning the number of registered access points). Because of this, we are calling this a minor release instead of a patch.
- Added support for XPA_PORT and XPA_PORTFILE environment variables to allow specification of the port to be used by the command channel (and data channel, if an optional second port is specified) for a given access point.
- Added -m switch to xpaget, xpaset, xpainfo, xpaaccess to allow override of the XPA_METHOD environment variable.
- Changed the default name of the ACL file from xpa.acl to acls.xpa.
- Fixed bug in which it was not possible to send a "set ACL" command to an XPA server which did not have a receive callback (i.e., did not allow xpaset). The xpans program is one such server. It now is possible to set the ACL on xpans.
- We have discovered that Tcl support for datachan and cmdchan is broken under Windows due to an unexplained incompatibility between Cygwin sockets and Win32 sockets. We therefore have removed datachan and cmdchan from the Windows/Tcl support until further notice.

- Extended the behavior of the XPA_DEFACL environment variable so that it can support more than one acl, using a list of semi-colon delimited controls such as: setenv XPA_DEFACL '*:* $host +; *:foo1 otherhost +'.
- Fixed bug in which the class:name specifier "*:*" was erroneously trying to access the xpans name server, instead of accessing all access points.
- Support TMPDIR and TMP environment variables as well as XPA_TMPDIR.

# Patch Release 2.0.5 (10 November 2000)

- Added support for Tcl on Windows where there is no select()-based event loop (i.e., where there is no Tcl_CreateFileHandler call in Tcl)
- Minor fixes in Makefile for installing on Windows
- Minor compiler fixes from gcc -Wall.

# Patch Release 2.0.4 (20 September 2000)

- Removed extraneous include of varargs.h from find.c.
- Ported to SGI C compiler, which caught lots of unused variables, etc.
- Ported to Cygwin/Windows, which required that we change socket read() and write() calls to recv() and send() respectively. Also had to ensure that we only did socket I/O on sockets (no fileio).

# Patch Release 2.0.3 (15 June 2000)

- Fixed the client XPASet() and XPASetFd() calls to handle the specified max number of connections (they were ignoring this argument, leading to memory overwrites).
- Fixed Makefile.in so that CFLAGS and LDFLAGS are not hard-wired values.
- Fixed word.h to load malloc.h and stdlib.h only if they exist.
- Documentation fixes to programs.html (in xpaaccess) and client.html (XPANSLookup).
- Added explicit typecast to strlen() argument to MAX #define in XPAClientStart (strlen() is unsigned in Linux, which can break MAX).
- Removed bogus Imakefile from directory.
- Changed directory name to include patch level (i.e., xpa-2.0.3).

# Patch Release 2.0.2 (9 September 1999)

- Fixed server mode (-s) in the xpaset program by properly cleaning up the input buffers (sending commands and data in server mode was broken).

# Patch Release 2.0.1 (6 August 1999)

- Fixed the Tcl binding code (tcl.c) for 64-bit machines (Dec Alpha) (erroneously used %x instead of %p when converting pointers to ASCII).

- Got rid of a few compiler warnings on 64-bit machines (a few are unavoidable since we must cast int to void * and back when passing around client data).

# Public Release 2.0 (27 May 1999)

- "a new day with no mistakes ... yet"

---

Index to the XPA Help Pages

---

**Last updated: 22 April 2002**

# XPACode: Where to Find Example/Test Code

## Summary

The XPA source code directory contains two test programs, *stest.c*, and *ctest.c* that can serve as examples for writing XPA servers and clients, respectively. They also can be used to test various features of XPA.

## Description

To build the XPA test programs, execute:

```
make All
```

in the XPA source directory to generate the *stest* and *ctest* programs. (NB: this should work on all platforms, although we have had problems with unresolved externals on one Sun/Solaris machine, for reasons still unknown.)

The stest program can be executed with no arguments to start an XPA server that contains the access points: xpa, xpa1, c_xpa (containing sub-commands cmd1 and cmd2), and i_xpa. You then can use xpaset and xpaget to interact with these access points:

```
cat xpa.c | xpaset xpa      # send to xpa
cat xpa.c | xpaset "xpa*"   # send to xpa and xpa1
xpaget xpa                  # receive from xpa
xpaget xpa*                 # receive from xpa and xpa1
```

etc. You also can use ctest to do the same thing, or to iterate:

```
ctest -s -l 100 xpa        # send to xpa 100 times
ctest -s -l 100 "xpa*"     # send to xpa and xpa1 100 times
ctest -g -l 100 xpa        # receive from xpa 100 times
ctest -g -l 100 "xpa*"     # receive from xpa and xpa1 100 times
```

More options are available: see the stest.c and ctest.c code itself, which were used extensively to debug XPA.

The file test.tcl in the XPA source directory gives examples for using the XPATclInterface.

Go to XPA Help Index

**Last updated: September 10, 2003**

# XPA Changes: Changes For Users from XPA 1.0 and 2.0

## Summary

This document describes changes that will affect users who migrate from XPA 1.0 to XPA 2.0.

## Description

There have been a few changes that affect users who upgrade XPA from version 1.0 to version 2.0. These changes are detailed below.

- XPA commands no longer have a resolver routine (this is open to negotiations, but we decided the idea was dumb). For the SAOtng program, this means that you must explicitly specify the access point, i.e.,:

  ```
  cat foo.fits | xpaset SAOtng fits
  ```

  instead of:

  ```
  cat foo.fits | xpaset SAOtng
  ```

- By default, xpaset, xpaget, etc. now wait for the server callback to complete; i.e., the old -W is implied (and the switch is ignored). This allows support for better error handling. If you want xpaset, etc. to return before the callback is complete, use -n switch:

  ```
  echo "file foo.fits" | xpaset -n SAOtng
  ```

- The old -w switch in xpaset and xpaget is no longer necessary (and is ignored), since you can have more than one process communicating with an xpa access point at one time.

- The new -p switch on xpaset means you need not read from stdout:

  ```
  xpaset -p SAOtng colormap I8
  ```

  will send the paramlist to the SAOtng callback without reading from stdin.

Go to XPA Help Index

**Last updated: September 10, 2003**

# XPAConvert: Converting the XPA API to 2.0

## Summary

This document describes tips for converting from xpa 1.0 (Xt-based xpa) to xpa 2.0 (socket-based xpa).

## Description

The following are tips for converting from xpa 1.0 (Xt-based xpa) to xpa 2.0 (socket-based xpa). The changes are straight-forward and almost can be done automatically (we used editor macros for most of the conversion).

- The existence of the cpp XPA_VERSION directive to distinguish between 1.0 (where it is not defined) and 2.0 (where it is defined).
- Remove the first widget argument from all send and receive server callbacks. Also change first 2 arguments from XtPointer to void *. For example:

```
#ifdef XPA_VERSION
static void XPAReceiveFile(client_data, call_data, paramlist, buf, len)
     void *client_data;
     void *call_data;
     char *paramlist;
     char *buf;
     int len;
#else
static void XPAReceiveFile(w, client_data, call_data, paramlist, buf, len)
     Widget w;
     XtPointer client_data;
     XtPointer call_data;
     char *paramlist;
     char *buf;
     int len;
#endif
```

- Server callbacks should be declared as returning int instead of void. They now should return 0 for no errors, -1 for error.
- The mode flags have changed when defining XPA server callbacks. The old *S* flag (save buffer) is replaced by *freebuf=false*. The old *E* flag (empty buffer is OK) is no longer used (it was an artifact of the X implementation).
- Change NewXPACommand() to XPAcmdNew(), with the new calling sequence:

```
xpa = NewXPACommand(toplevel, NULL, prefix, NULL);
```

  is changed to:

```
xpa = XPACmdNew(xclass, name);
```

- Change the AddXPACommand() subroutine name to XPACmdAdd (with the same calling sequence):

```
AddXPACommand(xpa, "file",
  "\tdisplay a new file\n\t\t  requires: filename",
  NULL, NULL, NULL, XPAReceiveFile, text, NULL);
```

  is changed to:

```
XPACmdAdd(xpa, "file",
  "\tdisplay a new file\n\t\t  requires: filename",
  NULL, NULL, NULL, XPAReceiveFile, text, NULL);
```

- The XPAXtAppInput() routine should be called just before XtAppMainLoop() to add xpa fds to the Xt event loop:

```
/* add the xpas to the Xt loop */
XPAXtAddInput(app, NULL);

/* process events */
XtAppMainLoop(app);
```

- Change NewXPA() to XPANew() and call XPAXtAddInput() if the XtAppMainLoop routine already has been entered:

```
xpa = NewXPA(saotng->xim->toplevel, prefix, xparoot,
             "FITS data or image filename\n\t\t  options: file type",
             XPASendData, new, NULL,
             XPAReceiveData, new, "SE");
```

  is changed to:

```
sprintf(tbuf, "%s.%s", prefix, xparoot);
xpa = XPANew("SAOTNG", tbuf,
             "FITS data or image filename\n\t\t  options: file type",
             XPASendData, new, NULL,
             XPAReceiveData, new, "SE");
XPAXtAddInput(XtWidgetToApplicationContext(saotng->xim->toplevel), xpa);
```

- Change XPAInternalReceiveCommand() to XPACmdInternalReceive() remove first argument in the calling sequence):

```
XPAInternalReceiveCommand(im->saotng->xim->toplevel,
                          im->saotng, im->saotng->commands,
                          "zoom reset", NULL, 0);
```

  is changed to:

```
XPACmdInternalReceive(im->saotng, im->saotng->commands,
                      "zoom reset", NULL, 0);
```

- Change DestroyXPA to XPAFree:

  ```
  DestroyXPA(im->dataxpa);
  ```

  is changed to:

  ```
  XPAFree(im->dataxpa);
  ```

Go to XPA Help Index

**Last updated: September 10, 2003**

# XPAName: What does XPA stand for?

## Summary

What does XPA stand for? Who knows anymore!

## Description

What does XPA stand for? Dunno! The XPA messaging system originally was built on top of the X Window System and XPA was the mnemonic for *X Public Access*, to emphasize that we were providing public access to previously private data and algorithms in Xt programs. Now that XPA no longer is tied to X, it can be argued that we ought to change the name (how about *SPAM: simple public access mechanism*), but XPA is in wide-spread use in the astronomical community of its birth, and the name has taken on a life of its own. If anyone can think of what XPA now means, please let us know.

If you think this is bad, consider the MMT Telescope on Mount Hopkins, Arizona. When first installed twenty years ago, it featured an array of six 72-inch diameter mirrors. from which came its name: the *Multiple Mirror Telescope*. In spring of 1999, these mirrors were replaced by a single 21 and 1/2 -foot diameter primary mirror, the largest single-piece glass reflector on the North American continent. And now MMT stands for ... MMT!

Go to XPA Help Index

**Last updated: September 10, 2003**