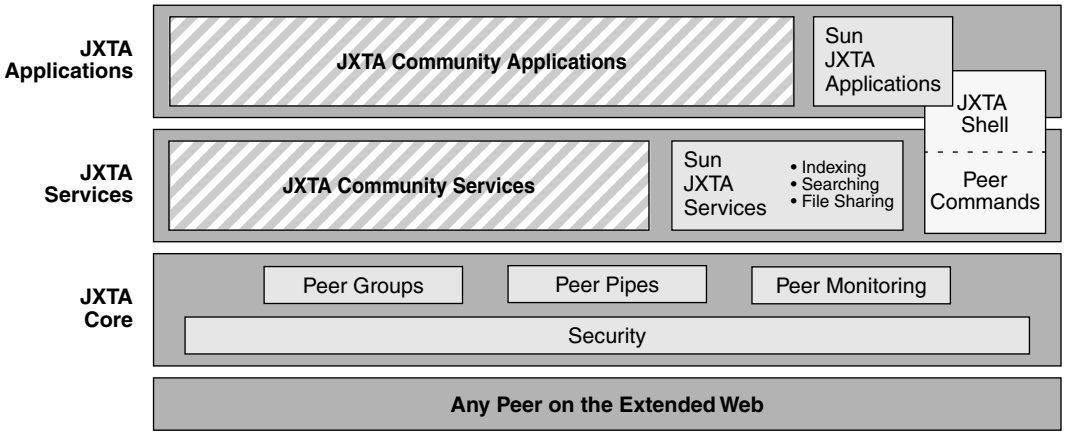
Project JXTA:
Technical Shell Overview

The JXTA Shell

The JXTA Shell permits interactive access to the JXTA platform via a simple command line interface. In the UNIX® operating system, interactive access is given by a Shell command line interpreter that enables users to manipulate and manage files and processes. Similarly, the JXTA Shell enables through a command line interpreter, interactive access to JXTA core building blocks: *peers, peer groups, pipes* and *codats*. Codats are units of contents that can hold both code and data. Codats are the smallest units of contents manipulated by the JXTA platform. Using the Shell command interpreter, a user can interact with the JXTA platform to publish, search, and execute codats, discover new peers or peer groups, create pipes to connect two peers, and send and receive messages.

The JXTA Shell is a command interpreter application that parses user's commands and interacts with the JXTA platform core services (Discovery, Membership, Authentication and Codat Management services). The interpreter operates in a simple loop: it accepts a command, interprets the command, executes the command, and then waits for another command. The shell displays a "`JXTA>`" prompt, to notify users that it is ready to accept a new command.

The JXTA Shell is an application identical to any other application built on the JXTA platform, except it is configured to run by the JXTA platform boot code. The JXTA Shell is not part of the JXTA platform (see figure), but sits on top of the core platform and services. The Shell functionality is tightly integrated with the JXTA core and provides commands to access key features of the platform. The JXTA Shell is not required to run on every JXTA peer. For instance, the Shell is not required if no interactive access to the peer is authorized. An alternative starting application may also be configured when the platform boots.



P2P Software Architecture

The JXTA Shell is an example of a Shell for the JXTA platform. It is provided as part of the JXTA Developer Kit to demonstrate one possible way to interact with the JXTA platform. We expect that multiple Shell applications will be developed to match different user needs and requirements just like many versions of the Shell have been developed for UNIX .

The JXTA Shell as delivered with the JXTA platform Developer Kit recognizes a limited set of commands. Each Shell command consists of a command name, followed by command options and arguments. The command names, options, and arguments are separated by blank spaces. The names of these commands have been deliberately chosen to be similar to UNIX Shell commands (e.g., *ls, cat, etc*.) to make the JXTA shell familiar to UNIX Shell users.

Most Shell commands are not built into the Shell, but are dynamically loaded and started by the Shell framework when they are invoked. Separating the Shell framework from the commands enables developers to dynamically add new commands to the Shell.

Shell Input and Output Pipe Re-Directions

The standardization of UNIX commands is fundamental to UNIX Shell programming and is a key reason to the success of UNIX. UNIX commands can be easily connected to form complex functions (e.g., *cat myfile | grep JXTA*). A key feature of the UNIX Shell is the *pipe* operator ("|") to construct a pipeline command sequence. The standard output of each command in the pipeline sequence, except for the last, is connected by a pipe to the standard input of the next command, with each command run as a separate process. The UNIX Shell waits for the last command to complete after a sequence of commands is issued. The exit status of the pipeline sequence is the exit status of the last command. The pipe construct enables users to build complex commands from simple ones in an intuitive way.

Like the UNIX Shell, the JXTA Shell provides a similar capability to redirect a command output pipe into another command input pipe. The JXTA Shell pipe, however, extends this capability by taking advantage of the asynchronous and unidirectional nature of JXTA pipes. In the JXTA Shell, an output pipe connection can be dynamically disconnected and reconnected to a different input pipe. This disconnection operation can be performed multiple times over the lifetime of the pipe. The ability to transparently redirect the output of pipes is an essential feature to build highly-available services in a loosely-coupled and unreliable environment such as peer-to-peer networks.

Every JXTA Shell command is given standard input, output, and error pipes that a user can connect, disconnect, and reconnect to any other Shell commands. Commands can support as many pipes as they require.

Every JXTA Shell command has the following syntax:

```
command [< pipe] [ >pipe] options arguments ;
```

where:

```
'>': redirects the output of the command
'<': redirects the input of the command
';' is a command separator.
```

In UNIX, the C Shell command `cat myfile | grep me` has to complete or be killed (*Cntrl-C*) before a user can modify the pipe re-direction. In JXTA, a user can dynamically disconnect and reconnect pipes between commands:

```
cat >p1 myfile /* cat myfile into the output pipe p1 */
grep <p1 me      /* connect the "grep" input pipe from the pipe p1*/
grep <p1 you     /* disconnect the output pipe p1, redirect the output of
                    p1 to the new grep "you" command input pipe*/
```

A short cut notation of the above pipe connection can be used via the '|' pipe operator to perform the same redirection:

```
cat myfile | grep me
```

is equivalent to

```
cat >p1 myfile ; grep <p1 me
```

The JXTA Shell also supports piping in both directions, not just in one direction as in the UNIX Shell. The JXTA Shell allows crossing pipe connections to be setup between two commands, e.g. the output pipe of the first command goes to the standard input pipe of the second command. The output pipe of the second command goes to the standard input pipe of the first command. A special crossing operator "<>" is used for creating crossing pipes between two commands

For example:

```
cmd1 <> cmd2
```

This command is equivalent to the following commands:

```
cmd1 >p1 <p2 ; cmd2 <p1 >p2
```

Importing and Exporting Data in the JXTA Shell

The JXTA Shell provides a generic framework for importing and exporting data in and out of the JXTA platform. A *share* and *unshare* command (see below) is provided to import data into a codat container, or to export a codat into an external object (i.e. a file under UNIX). The *share* command also permits users to associate a symbolic name and object type with a codat (e.g. the codat symbolic name is *myfile* and the type is *PostScript*). The symbolic name associated with a codat can be used afterward to reference codats within the Shell. The type of the codat can be used by the codat Management System to activate an associated service defined for that codat type. For example, when accessing a PostScript codat, a PostScript viewer can be launched to display the codat content.

An URI address scheme is used by the *share* and *unshare* commands to specify the address destination of external objects. The JXTA Shell only implements local file:/ URIs. It is expected that initially a flat symbolic name space will be used for codats. No directory or hierarchical structure is maintained when file objects are imported. Each peer group provides its own name space for codats. A hierarchical directory structure can be added later if necessary.

Batch Files

The Shell provides a *-f* option to load and execute batch command files. The *load* command can execute a set of Shell commands previously stored in a codat or in an external object (file). The codat type is marked as a batch codat (i.e., can be loaded and interpreted). Each command in the batch file is interpreted as if it was entered by the user. The JXTA Shell only provides the ability to create simple batch scripts, no conditional or looping operations are supported.

Basic JXTA Shell Commands

Here are the basic commands implemented by the JXTA Shell.

Shell [-f *filename*] [-s]

The Shell creates an input pipe (*stdin*) for reading input from the keyboard, and an output pipe (*stdout*) to display information on the Shell console. All commands executed by the Shell have their initial *stdin* and *stdout* set up to the Shell's *stdin* and *stdout* pipes. The Shell also creates the environment variable *stdgroup* that contains the current JXTA peer group in which the Shell and commands are executed.

The following Shell environment variables are defined by default:

consin = Default Console InputPipe
consout = Default Console OutputPipe
stdout = Default OutputPipe
stdin = Default InputPipe
Shell = Root Shell
stdgroup = Default peer group
rootgroup = Default NetPeerGroup

A new Shell can be forked within a Shell. The command *Shell -s* starts a new Shell with a new Shell window. The Shell can also read a command script file via the command *Shell -f myfile*.

Multiple commands can be entered in one line. Pipelines can be created by combining the pipe *stdout* of a command into the pipe *stdin* of another command using the pipe ('|') operator. For example the following command:

```
JXTA> cat env1 | more
```

Pipes the output of the command cat into the *stdin* of the command *more*. Arbitrary numbers of commands can be pipelined together within a single Shell command. Currently '|' is the only pipe operator supported.

The '=' operator can be used to assign the value of a command output to an environment variable. For example :

```
JXTA> myadv = mkadv -p
```

Shell commands stored in a file can be run in a batch mode using the *-f* option.

Example:

```
JXTA> Shell -f /home/tra/batch
```

This command executes the commands stored in the Shell script file */home/tra/myfile* in the current Shell environment.

A default startup batch file *$HOME/.jshrccan* be setup that is executed when the Shell is invoked.

whoami [-g] [-l]

Returns the local peer or the default peer group advertisement. The command displays an XML document that represents the Peer or PeerGroup advertisement. The long option *-l* shows the entire advertisement. The default (short) version only shows the Peer UUID and peer endpoints. The *-g* option displays information about the default peer group.

Example:

```
JXTA> whoami -l
<Name>Neptune</Name>
<Id>JXTA:/
                    000000000000000000000000000000A8476E187C424CA99763CE
                    F538FE34F1000000000000000000000000000000000000000000
                    0000000000000000001</Id>
<NetworkPeerGroup>Sun</NetworkPeerGroup
<TransportAddress>TCP:129.144.94.155:6001</TransportAddress>
```

env

Displays the current environment variables (*stdin* and *stdout* pipes, *peer* and current *PeerGroup*). Each environment variable is listed with its value.

The following environment variables are defined by default:

consin = Default Console InputPipe
consout = Default Console OutputPipe
stdout = Default OutputPipe
stdin = Default InputPipe
Shell = Root Shell
stdgroup = Default peer group
rootgroup = Default NetPeerGroup

Shell environment variables are defined as a result of executing Shell commands. The '=' operator can be used to assign value to a particular variable. For example *myenv = mkmsg* will assign a new message object to the *myenv* environment variable.

Example:

```
JXTA> env
stdgroup = Astronomy
consin = Default Console InputPipe
consout = Default Console OutputPipe
stdout = Default OutputPipe
stdin = Default InputPipe
Shell = Root Shell
```

peers [-r] [-p *peerName*] [-n *limit*] [-a *TagName*] [-v *Tagvalue*] [-f]

Executes the JXTA discovery protocols to find other peers in the scope of the default peer group. With no options, the command lists only the peers already known by the peer (cached). The *-r* option is used to send a propagate request to find new peers. The peers command stores results in the local cache, and inserts advertisement(s) into the environment, using the default naming: *peerX* where *X* is an increasing integer number.

A specific *peerId* (i.e. a rendezvous peer) can be specified using the *-p* option to expand the discovery search. The *-v* and *-a* options allow a (*tag, value*) string to be passed as search criteria. The *-f* option flushes the local list of known peers. The *-n* option is used to limit the number of responses to a remote discovery request.

Example:

```
JXTA> peers -r
peer discovery message sent
JXTA> peers
peer0: name = Demo JXTA Peer
peer1: name = Dioxine.net
peer2: name = chrisatwork
peer3: name = ecksteinpeer
peer4: name = emily5
peer6: name = john@home
JXTA>
```

groups [-r] [-p *peerName*] [-n *limit*] [-a *TagName*] [-v *Tagvalue*] [-f]

Executes the JXTA discovery protocols to find peer groups in the scope of the default peer group. With no options, the command lists only the peer groups already known by the peer (cached). The *-r* option is used to send a propagate request to find new peer groups. The *groups* command stores results in the local cache, and inserts advertisement(s) into the environment, using the default naming: *groupX* where X is an increasing integer.

A specific *peerId* (i.e. a rendezvous peer) can be specified using the *-p* option to expand the discovery search. The *-v* and *-a* options allow a (*tag, value*) string to be passed for providing search criteria for peer groups. The *-f* option flushes the local list of known peer groups. The *-n* option is used to limit the number of response to a remote discovery request.

Example:

```
JXTA> groups -r
group discovery message sent
JXTA> groups
group0: name = NetPeerGroup
group1: name = kaja
group2: name = dailupGrp
group3: name = frog101
group4: name = stevesgroup
group5: name = ice
group6: name = steve
group7: name = raelity
JXTA>
```

mkadv [-g|p] [-t type] [-d doc] name

Create a new peer group or pipe advertisement from the given configuration URI. The URI points to a file that contain a valid Peer Group or Pipe advertisement. The *-p* option is used to create a pipe advertisement. The *-t* option is only valid for pipes and defines the type of pipes. The *-g* option is used to create a new peer group advertisement. If no document is given, a clone of the current peer group is created. The new peer group will have the same policies and endpoints as the current group. If a *-d* argument is supplied, the document must be a peer group advertisement and is used to create the new peer group. The *-d* option specifies a Shell environment variable that contains a structure that holds an advertisement document that, in turn, contains as advertisement.

Example

```
JXTA> importfile -f saveadv groupadv
JXTA> mygroupadv = mkadv -g -d groupadv
JXTA> mkpgrp -d mygroupadv mygroup
JXTA> mkadv -p -t secure lightpipe file:/home/zeus/mypipe.xml

<?xml version="1.0" encoding="UTF-8"?>
<JXTA:PipeAdv>
<name>lightpipe</name>
<type>secure</type>
<PipeId>JXTA://
                  C263C0890CEA4721BAAAE6A564905603E3CC3D1732234D9E93AFA9
                  BBFFC54CDE0000000000000000000000000000000000000000000000
                  0000000000000000001</PipeId>
</JXTA:PipeAdv>
```

mkpgrp [-d *doc*] [-m *policy*] *groupname*

mkpgrp creates a new peer group using the supplied peer group advertisement. If no advertisement is provided, the command creates a clone of the *NetPeerGroup* peer group with the name specified. The command mkadv *-g* is used to create a peer group advertisement. The environment variable *PG#<group name>* is created to store the new peer group. The new peer group is advertised in the *NetPeerGroup*. All peer groups are created in the NetPeerGroup which acts like the JXTA world peer group. Every peer group can be found in this group.

Example:

```
JXTA> mygroupadv = mkadv -g
JXTA> mypgrp = mkpgrp -d mygroupadv mygroup
```

This creates a new peer group which clones the policies of the parent peer group. You can find the policies of the current peer group via the command *whoami -g*. The new group is given the name *mygroup*. Before you can do anything with the group you need to join the group via the *join* command.

chpgrp *group*

The *chpgrp* command is used to switch the default Shell peer group *stdgroup* variable to another group that was previously joined via a *join* command. The *join* command is used to join a peer group. After changing group, the Shell *stdgroup* variable is set to the value of the new peer group joined.

Example:

```
JXTA> mygroupadv = mkadv -g mygroup
JXTA> mkpgrp -d mygroupadv mygroup
JXTA> join mygroup
JXTA> chpgrp moi
```

This creates a new peer group which clones the policies of the parent peer group. You can find the policies of the current peer group via the command *whoami -g*. The new group is given the name *mygroup*. Before you can do anything with the group you need to join the group via the *join* command. The *chpgrp* command is used to change the default group to the new group *moi*.

join [-d *doc*] [-c *credential*] [*groupName*]

Join the peer group identified by the *PeerGroup* name parameter. The command can pass an optional authentication credential *-c* to be used by the peer group membership authenticator. The current *PeerGroup* environment variable is set to the new peer group the user is joining. If no argument is given, *join* lists all the existing groups and their status (*join, unjoined*) and the current group on the local peer. After a group is joined successfully, the *PG@<group name>* environment variable is created. This variable holds the group information. When joining a new peer group, the new peer group is advertised in the *NetPeerGroup*. Hierarchies of peer groups are not supported.

Example:

```
JXTA> join -c mypasswd Astronomy
Joined sucessfully PeerGroup Astronomy
JXTA> mygroupadv = mkadv -g mygroup
JXTA> mkpgrp -d mygroupadv mygroup
JXTA> join -c credential mygroup
```

This creates a new peer group which clones the policies of the parent peer group (*NetPeerGroup*). You can find the policies of the current peer group via the command *whoami -g*. The new group is given the name *mygroup*. Before you can do anything with the group you need to join the group via the *join* command.

leave *groupName*

Leave a peer group. The *leave* command is used to leave a group that was previously joined via a *join* command. The *join* command is used to join a peer group. After leaving the group, the Shell *stdgroup* variable is reset to the value of the default *rootgroup* variable (*NetPeerGroup*). Before a user can use the group again, the user will have to rejoin the group using the *join* command.

Example:

```
JXTA> mygroupadv = mkadv -g mygroup
JXTA> mkpgrp -d mygroupadv mygroup
JXTA> join mygroup
JXTA> leave
```

This creates a new peer group which clones the policies of the parent peer group (*NetPeerGroup*). You can find the policies of the current peer group via the command *whoami -g*. The new group is given the name *mygroup*. Before you

can do anything with the group you need to join the group using the *join* command. The *leave* command is used to leave the group.

ls [-l*PG*] [-p *peerName*]

List codats cached on the local peer for the current peer group. The *-l* option generates a long format with statistics information about each codat: symbolic name, size, type, etc. The *-P* option lists all the member peers discovered in the current peer group. The[*-G* option lists all the peer groups that have been discovered. The *-p* option lists codats stored on a specified remote peer member of the current peer group. The peer must be a member of the current peer group.

Example:

```
JXTA> ls
Name       Index                Type       Size
sunpic     "Planet sun Picture" Postscript 50 MB
earthpic   "Earth picture"      GIF        40 KB
moonletter "letter to the moon" Txt        10KB
JXTA> ls -P
Moon TCP:129.144.94.156:6001
Earth TCP:129.144.94.134:6001
```

share [-m *codatName*] [-t *type*] [-*s size*] [-e *encoding*] [-i *index*] *codatName URI*

Share a codat created from a URI external object. A name for the codat must be supplied. The name is used to create a symbolic name to reference the codat in future Shell commands. If the codat is metadata, the *codatName* (this codat is about) is specified in the *-m* option. A type, size and encoding description can be assigned to the codat. Default values are assigned if no values are provided. The URI specifies the location of the external object to be stored in the codat. The *-I* option creates an association of an indexing string to the codat, allowing a codat management service to insert the codat in the local store and then reference the codat via the specified index string. This enables indexing to be performed at the time the codat is shared.

Example:

```
JXTA> share -type Txt -i "compute star weight" startweight
                  file:/home/zeus/file.txt
```

search [-m *codatName*] [-p *peerName*] [-*s search*] [*codatName*]

Search a codat using its name or a search string. The *-m* option gets the list of metadata codats associated with this codat. A specific peer can be specified using *-p* to search codats. By default, the search is performed on the local peer. The *-s* option is used to search codats via a search string. The codat management service tries to match this string with the index strings stored when codats are shared (see the share command).

Example: search for a codat which was indexed with the keyword weight.

```
JXTA> search -i "planet"
Name       Index                Type       Size
sunpic     "Planet sun Picture" Postscript 50 MB
```

cat [-p] *env*

Display on *stdout* the content of objects stored in environment variables. *cat* knows how to display a limited (but growing) set of JXTA objects: *Advertisement*, *Message,* and *StructuredDocument*. If you are not sure, try to *cat* the object. The Shell will let you know if it cannot display that object.

Example:

```
JXTA> cat pipeAdv
<?xml version="1.0" encoding="UTF-8"?>
<JXTA:PipeAdv><PipeId>JXTA://
                  C263C0890CEA4721BAAAE6A564905603E3CC3D1732234D9E93AFA9
```

```
BBFFC54CDE00000000000000000000000000000000000000000000000000
000000000000000000001</PipeId></JXTA:PipeAdv>
```

unshare [-p *peerName*] *codatName* [URI]

Delete a codat from the local peer. The *-p* option allows the deletion of a codat from a remote peer. If *peerName* is not given, the codat is removed from the member peers that have been discovered by this peer. A URI can be given to export the content of the codat into an external object before the codat is deleted.

Example:

```
JXTA> unshare sunpic1
```

mkpipe -i|o *pipeAdv*

Create a new input pipe or output pipe for the given pipe advertisement.(see the *mkadv* command to create pipe advertisements). *mkpipe* creates an input pipe or an output pipe from a given pipe advertisement document. In order for pipes to communicate, an input and output pipe needs to be created with the same pipe advertisement. Pipe advertisements are structured documents that contains at least a unique pipe ID. The pipe ID uniquely identifies a pipe. Pipes are not localized or bound to a physical peer. Pipe connections are established by searching for pipe advertisements and resolving dynamically the location of an input pipe object bound to that advertisement. An input pipe can be bound to the same pipe advertisement on multiple peers, transparently to the output pipe. The output pipe does not need to known on which physical peer the input pipe is located. To communicate with the pipe, the output pipe needs to search for the input pipe that binds this advertisement.

Example:

```
JXTA> pipeadv = mkadv -p
JXTA> inpipe = mkpipe -i pipeadv
JXTA> msg = recv inpipe
JXTA> data = get msg mytag
```

This example creates a pipe advertisement *pipeadv*, creates an input pipe *inpipe* and receive a message *msg*. The body of the message associated with the tag *mytag* is retrieved from the message with the *get* command.

mkmsg *msg*

Create a new pipe message (*msg*). *mkmsg* creates a new message to send or receive data from a pipe. The message object is stored in a Shell environment variable. If no name is assigned via the '=' operator, a default environment variable *env#* is created for holding the message object (# is an increasing integer).

JXTA messages are composed of multiple tag body parts. Each tag body is uniquely identified via a unique tag name. The tag name is used to insert (*put* command) a new tag body in a message, or to retrieve (*get* command) a tag body from a message.

Example:

```
JXTA> mkmsg
```

This creates a message object and puts it in the environment variable *env#* where *#* is an integer number (e.g. *Env4*). You can assign a specific name to the message variable by assigning it a name with the '=' Shell operator.

```
JXTA> mymsg = mkmsg
JXAT> put mkmsg mytag data
JXTA> send outpipe mymsg
```

This create a new message *mymsg*, stored data in the message body tag *mytag*. The message is then sent via the output pipe *outpipe*

put *msg tag document*

Push the user data into the message with the specified *tag*. Pipe messages can hold many tags. *put* stores a document into the body of message. JXTA messages are composed of a set of tag bodies, each identified with a unique tag name A message tag name is supplied to specify which tag name is used to store the document. On the receiving system the document can be retrieved via the *get* command.

Example:

```
JXTA> importfile -f /home/tra/myfile mydata
JXTA> msg = mkmsg
JXTA> put msg mytag mydata
```

This example creates a document *mydata* by importing data from the file */home/tra/myfile*. Then, we create a pipe message *msg* and store the document *mydata* into the message *msg* with the associated tag name *mytag*.

get *msg tag*

Get the data associated with the given *tag* from an incoming message into a Shell environment variable. *get* retrieves the tag body of a message. JXTA messages are composed of a set of tag bodies, each identified with a unique tag name. A message tag name is supplied to specify which tag body to extract.

Example:

```
JXTA> pipeadv = mkadv -p
JXTA> inpipe = mkpipe -i pipeadv
JXTA> msg = recv inpipe
JXTA> data = get msg mytag
```

This example creates a pipe advertisement *pipeadv*, creates an input pipe *inpipe*, and receives a message *msg*. The tag body of the message associated with the tag *mytag* is retrieved from the message via the *get* command.

send *outputpipe msg*

Send a message to an output pipe

Example:

```
JXTA> pipeadv = mkadv -p
JXTA> outpipe = mkpipe -o pipeadv
JXTA> send outpipe msg
```

This example creates a pipe advertisement *pipeadv*, creates an output pipe *outpipe* and sends the message *msg* through the pipe.

recv [-t *timeout*] *inputpipe*

Receive a message from an input pipe. A timeout parameter *-t* can be given to timeout the receive operation. By default, the receive will block until a message is received on the pipe.

Example:

```
JXTA> pipeadv = mkadv -p
JXTA> inpipe = mkpipe -i pipeadv
JXTA> msg = recv inpipe
JXTA> data = get msg mytag
```

This example creates a pipe advertisement *pipeadv*, create an input pipe *inpipe* and receive a message *msg*. The body of the message associated with the tag *mytag* is retrieved from the message via the *get* command.

man [*commandName*]

Return list of shell commands or help information about a specified command.

```
JXTA> man discover
```

importfile -f *filename* [*env*]

importfile imports an external file into a *StructuredDocument* object stored in a Shell environment variable. The name of the environment variable is specified as an argument. *importfile* is the reverse operation of *exportfile*.

Example:

```
JXTA> importfile -f /home/tra/myfile myfile
JXTA> cat myfile
```

This command imports the file */home/tra/myfile* into the *myfile* environment variable.

exportfile -f *filename* [*env*]

exportfile exports the contents of a Shell environment variable into an external file. The exported object is stored in the supplied *filename* argument. If no variable name is given, the *stdin* pipe is used to read data and save it into the file. *exportfile* is the reverse operation of *importfile* The Shell environment variable is not deleted after the file is created.

Example:

```
JXTA> exportfile -f /home/tra/myfile myfile
```

This command saves into the file */home/tra/myfile* the contents of the *myfile* environment variable

version

List the release version of the JXTA Shell.

Example:

```
JXTA> version
jxta version 0.9 (build 11f, 04-04-2001)
```

peerinfo [-p *peerid*] [-r] [-l] [-f]

peerinfo can be used use to get information about other peers within a peer group or at a specified peer location. The default *peerinfo* options list only the information already known by the peer. The *-r* option is used to send a propagate request to find information about new peers. *peerinfo* stores results in the local cache, and inserts advertisement(s) into the environment, using the default naming: *peerinfoX* where is an increasing integer.

Example:

```
JXTA> peerinfo -r
peer info message sent
JXTA> peerinfo -l
stored peerinfo0
JXTA> cat -p peerinfo0
<?xml version="1.0"?>
<!DOCTYPE jxta:PeerInfoAdvertisement>
<jxta:PeerInfoAdvertisement>
   <sourcePid>

jxta://
                  5961626164626614A78746150325033C5D7371A5F374A6899429C
                  21D077EDE1000000000000000000000000000000000000000000
                  0000000000000000301
   </sourcePid>
   <targetPid>

jxta://
                  5961626164626614A78746150325033C5D7371A5F374A6899429C
                  21D077EDE1000000000000000000000000000000000000000000
                  0000000000000000301
   </targetPid>
   <uptime>
      411
   </uptime>
   ......
</jxta:PeerInfoAdvertisement>
JXTA>
```

In the truncated example above, peer uptime and peer ID information is shown. Updated information may be viewed by repeating the command sequence.

exit

Exit the Shell window.

```
JXTA> exit
```

An Open Framework for Adding New Shell Commands

The JXTA Shell is an open framework that allows new Shell commands to be added dynamically. Every Shell command is a separate application that is loaded by the Shell framework when the command is invoked. The Shell framework does not need to be recompiled when adding new commands. The JXTA Developer Kit provides base Shell command classes that can be extended to implement new commands. As long as these commands follow the guidelines of the JXTA Shell framework invocation, they will be able to be loaded by the Shell framework and communicate through pipes with other existing commands. New commands can be implemented for a variety of purposes, e.g. search engines, graphical interfaces, webproxies, etc. New commands might also be implemented for administrative or accounting purposes.

How to Write a New Command

All Shell commands need to extend the *net.JXTA.imp.shell.ShellApp* Class. This class provides the framework to interact with the Shell console, print and read from the console, and setup environment variables for the command (*stdgroup*). The new command class needs to implement the two method *startApp* and *stopApp*. The *startApp* method is called after the command is loaded. The *stopApp* is called when the Shell exits.

Here is an example of a simple "*Hello World*" command:

```
Package net.jxta.impl.shell.bin.myHelloWorld

public class myHelloWorld extends ShellApp {

   private ShellEnv myEnv;

   public int startApp (String[] args) { //args has arguments to command
      myEnv = getEnv(); // retrieve the command environment

      // extract the current peerGroup from the environment

      ShellObject obj = myEnv.get ("stdgroup");
      PeerGroup group = (PeerGroup) obj.getObject();
      if (args == null) { //println print to the console
         println("Hello my peergroup is" + group.toString());
      } else { // no arguments are authorized
         println ("Sorry no argument supported!");
      }
      return ShellApp.appNoError; // Everything went OK!
   }

   public void stopApp () {
      // not much to be done here
   }
}
```

All Shell commands are located in the package *net.jxta.impl.shell.bin*. For a new command to be accessible from the Shell, the new command class file needs to be copied into the Shell *bin class* directory. There is no need to recompile the Shell. By copying the new class file into the Shell *bin class* directory, the command will be dynamically loaded when invoked. The Shell does not have to be restarted.

SUN MICROSYSTEMS, INC., 901 SAN ANTONIO ROAD, PALO ALTO, CA 94303-4900 USA
PHONE: 650-960-1300 FAX: 650-969-9131 INTERNET: *www.sun.com*

Draft 1.0 April 25, 2001