Table of Contents

Contents:

Panther

JetNet/Oracle Tuxedo Guide

Prolifics®

Release 5.51

Document 0404
March 2017

Copyright

This software manual is documentation for Panther® 5.51. It is as accurate as possible at this time; however, both this manual and Panther itself are subject to revision.

Send suggestions and comments regarding this document to:

Technical Publications Manager
Prolifics, Inc.
24025 Park Sorrento, Suite 405
Calabasas, CA 91302

http://prolifics.com
support@prolifics.com
(800) 458-3313

TABLE OF CONTENTS

Contents:

C. Enterprise Bank

About This Document

The *JetNet/Oracle Tuxedo Guide* is divided into two sections. The first section, about setting up and maintaining your application servers, contains the following information:

- An overview of three-tier system architecture and implementation of this model with Panther.
- Setup requirements for Panther development environments and deployed applications.
- How to use the JetNet manager to configure an application, its machines and servers.
- How to use the JetNet manager and other utilities to boot, monitor and manage a running Panther application.

Description of command-line utilities that can help you develop and manage a Panther application are located in the appendices.

The second section about writing services and calling them from your client application contains the following information:

- Description of a service.
- How to write services and make them available to your application.
- How to use features only available in the Panther Oracle Tuxedo Edition.

What You Need to Know

The first section of this guide is written for system administrators who are responsible for setting up a Panther development environment and deploying a Panther application. Knowledge of Panther can be helpful but is not essential.

The second section is written for application developers who are responsible for writing JetNet and TUXEDO services and calling them from the client application.

This guide assumes that you have already installed Panther on your system.

Documentation Website

The Panther documentation website includes manuals in HTML and PDF formats and the Java API documentation in Javadoc format. The website enables you to search the HTML files for both the manuals and the Java API.

Panther product documentation is available on the Prolifics corporate website at http://docs.prolifics.com/panther/.

How to Print the Document

You can print a copy of this document from a web browser, one file at a time, by using the File→Print option on your web browser.

A PDF version of this document is available from the Panther library page of the documentation website. You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe website at https://get.adobe.com/reader/otherversions/.

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
Ctrl+Tab	Indicates that you must press two or more keys simultaneously. Initial capitalization indicates a physical key.
italics	Indicates emphasis or book titles.
UPPERCASE TEXT	Indicates Panther logical keys. *Example*: XMIT
boldface text	Indicates terms defined in the glossary.

Convention	**Item**

`monospace text`	Indicates code samples, commands and their options, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <smdefs.h>`
	`chmod u+w *`
	`/usr/prolifics`
	`prolifics.ini`

`monospace italic text`	Identifies variables in code representing the information you supply. *Example*: `String` *`expr`*

`MONOSPACE UPPERCASE TEXT`	Indicates environment variables, logical operators, SQL keywords, mnemonics, or Panther constants.
	Examples:
	`CLASSPATH`
	`OR`

`{ }`	Indicates a set of choices in a syntax line. One of the items should be selected. The braces themselves should never be typed.
`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `formlib [-v]` *`library-name`* `[`*`file-list`*`]...`
`...`	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `formlib [-v]` *`library-name`* `[`*`file-list`*`]...`

Convention	Item
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Contact Us!

Your feedback on the Panther documentation is important to us. Send us e-mail at support@prolifics.com if you have questions or comments. In your e-mail message, please indicate that you are using the documentation for Panther 5.50.

If you have any questions about this version of Panther, or if you have problems installing and running Panther, contact Customer Support via:

- Email at support@prolifics.com
- Prolifics website at http://profapps.prolifics.com

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address and phone number
- Your company name and company address
- Your machine type
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

CHAPTER

1 Enterprise Model and Implementation

With Panther, you can build client/server applications that comply with either a two-tier or three-tier architecture on multiple operating systems, including Windows and UNIX; you can also build fully functional web applications that run on the Internet or intranet. Inside the Panther framework, a development environment for Panther components allows shared access to libraries, and includes utilities to configure and manage middleware and server components.

Three-Tier Processing

The two-tier client/server model typically separates data from the logic of an application. In the three-tier or enhanced client/server model, the *backend server* is known as the *resource manager*, and is usually a database. The layer between client and backend server is the *application server*. The client is responsible for user interactions, and the application server is responsible for providing business-level services and interacting with the resource manager as needed.

Three-tier applications offer the following benefits:

- Reduced overhead per user — Because the application server performs most processing, client processes require fewer resources. For example, to query a database, a client needs to contain only enough code to gather user input and send its request and data to the application server, which contains the code and SQL libraries required for database interaction.

- Reduced overhead per interaction — By off-loading most processing onto the application server, the client is free to perform its job independently of actual interaction with the database. Database requests can be efficiently serialized, thereby reducing the interaction time required for each request.
- Distributed processing — Because the client is removed from application server processing, clients and servers can reside on different machines.
- Modular design — By separating clients from database interaction, the database is protected from client errors. Further, you can modify client and server processes without disturbing other parts of the application.
- Extensibility — As the application grows, client and server processes can be added to handle increased traffic or provide new functionality. New client nodes and server machines can be added as needed while the application runs.

Three-Tier Application Components

The configuration for a three-tier JetNet or Oracle Tuxedo application typically accounts for these application components:

- Clients that host user interaction and issue service requests. Clients can be one of two types: a *non-workstation client* (or local client) resides on a machine which is running an application server; a *workstation client* attaches to an application through its connection to an application machine.

  Panther screens provide the mechanism for user interaction and client processing. Client screens are stored in client libraries.
- Services that are invoked at runtime to handle service requests. Services are defined in JPL or C or Java and are typically grouped with service components; refer to Chapter 5, “Defining Services in JetNet and Oracle Tuxedo Applications.” Service components and routines are stored in server libraries.
- The *JIF*, which specifies all the services and service groups that are available for a server to advertise. The JIF provides the mapping between a service and its service component and routine.

- Application servers that handle service requests from clients. Panther supports three types of application servers: *standard servers*, which advertise JIF-defined services; *conversion servers*, which handle service requests from client screens converted from a two-tier application; and *file access servers*, which provide developers shared access to libraries across the network.

  Application servers provide access to resource managers such as a database. Each server can be connected to one or more resource managers.

- A JetNet configuration file that defines properties for the application in general, and for participating machines and servers. When you start an application, Panther reads the configuration file to determine which servers to boot and on which machines. The configuration file is accessible through the JetNet manager. For more information, refer to Chapter 3, “Configuring the Enterprise.”

- A bulletin board liaison process (*BBL*) that resides on each machine and maintains the addresses of servers and the services that they provide. In a multi-machine configuration, the master machine also contains the distinguished bulletin board (*DBBL*), which maintains information about all machines and servers.

Figure 1-1 shows how components of a Panther enterprise application interact in a single-machine configuration:

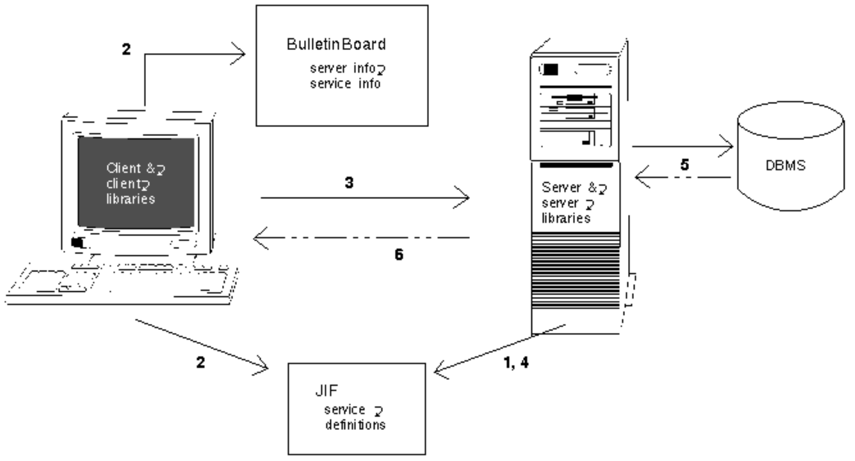


Figure 1-1 Interaction of components in a Panther enterprise application.

A Panther enterprise application typically works as follows:

1. During initialization, a server consults with the JIF to determine which services it needs to provide:
   - A server that is configured to provide all services reads all service definitions from the JIF.
   - A server that is initialized to provide services from a service group reads the services that are defined for that group.

   The initialized server advertises its services to the local bulletin board, which stores this information and the server's address.
2. When a client makes a service request, it consults the JIF to determine which data, if any, need to accompany the request and how to make the request. Panther then relays the request to the local bulletin board. Through the bulletin board data, the service request is matched to a specific server. This information is returned to the client.
3. The client sends its request to the server.
4. On receiving the service request, the server responds as follows:
   - Consults with the JIF to determine which routine and service component handles this service.
   - Finds the service component in the server library and uses it in order to process the request.
5. If the request requires database interaction, the server process provides the necessary SQL to query the database.
6. When the server finishes processing the request, it sends its reply to the client.

Workstation Connections

In the implementation shown earlier, all processing and client interaction takes place on the same machine. More typically, clients interact with an application from personal workstations. Each workstation is outside the JetNet configuration and communicates with the application over a network.

A server machine has two processes that enable workstation connections:

- A workstation listener (*WSL*) that intercepts requests to connect from workstation clients.
- A workstation handler (*WSH*) that manages all interaction between workstation clients and application servers.

The following figure shows an application that has two workstation client connections:

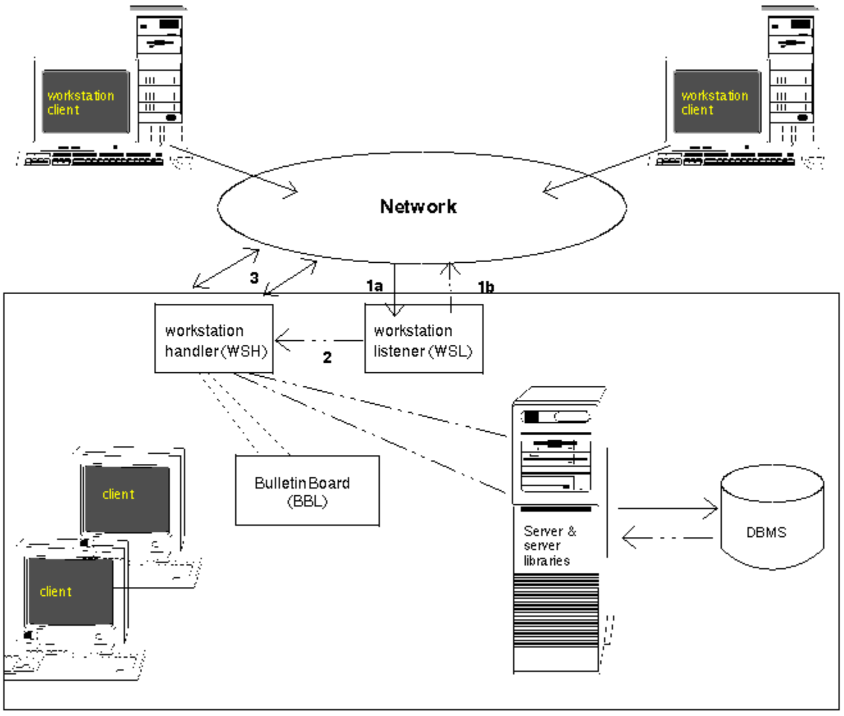


Figure 1-2 Three-tier configuration with workstation client connections

Workstation clients connect to the application in three steps:

1. The client connects to the server machine's workstation listener process through a known network address. The workstation listener returns the address of a workstation handler to the client.
2. The workstation listener process sends a message to the workstation handler, informing it of the connection request.

3. The workstation client connects to the workstation handler. All further communication between the client and the application takes place through the workstation handler.

Multi-Machine Configuration

The three-tier model can be expanded so that multiple server machines can participate in a single application. In this way, resources can be distributed to allow efficient throughput and avoid overburdening any one machine.

In a networked application, one host serves as the master machine; you can boot and shut down an application only from this machine. The master machine also has a DBBL, which coordinates among BBLs on all other machines including the master. All machines have a bridge process, which enables communication between them and the network.

Figure 1-3 diagrams a two-machine configuration that is enabled for workstation client connections. The data that users require on deposit data and loan data is split between the two sites; for example, a request for checking account data is directed to host 1, while a query on mortgages is directed to host 2. Host 1 is also the master machine:

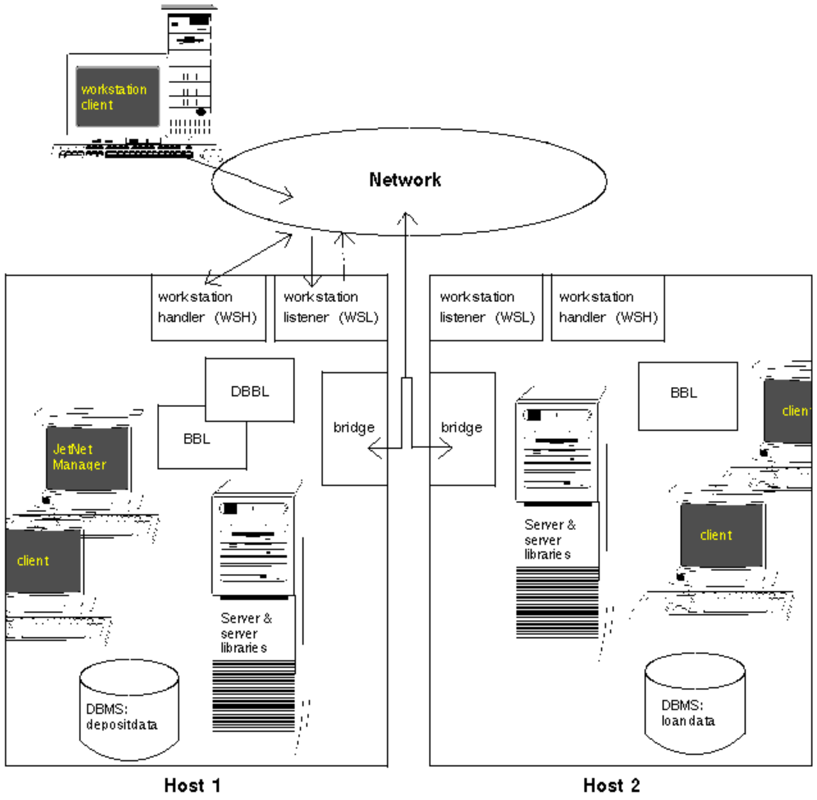

workstation client
Network
workstation handler (WSH)
workstation listener (WSL)
workstation listener (WSL)
workstation handler (WSH)
DBBL
BBL
bridge
bridge
BBL
JetNet Manager
client
client
client
Server & server libraries
Server & server libraries
DBMS: depositdata
DBMS: loan data
Host 1
Host 2

Figure 1-3 System with multiple server machines.

Web Application Server

Panther web applications can be integrated into a three-tier, distributed processing model. In this model, the Panther application server establishes and maintains connections to the database. When the web application requires data, the web application server, acting in the role of client, sends the service request to the Panther application server. The Panther application server processes this request and returns the desired data to the web application server.

Note that in a three-tier environment, the web application server remains responsible for opening screens, mapping browser and data cache values, and generating HTML from the screen.

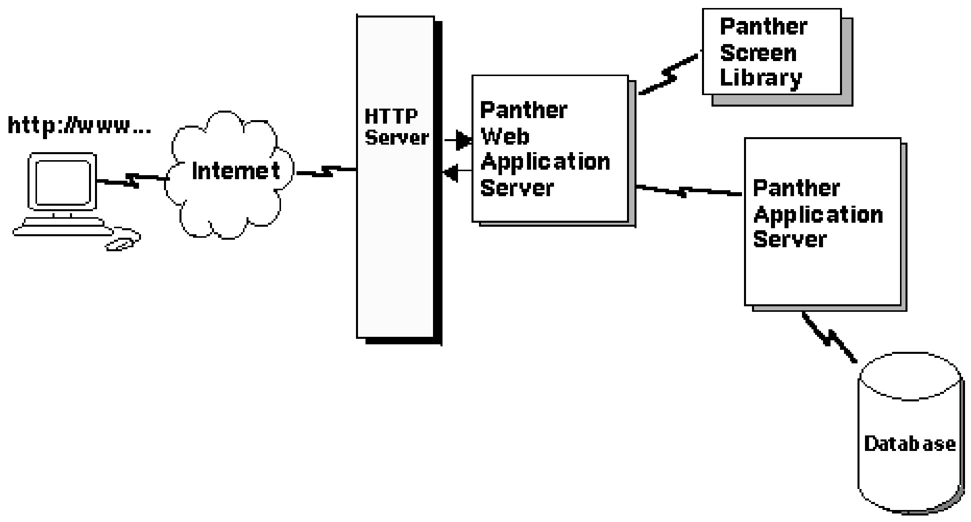


Figure 1-4 In three-tier processing, the web application server is a Panther client.

For more information about web application development, refer to the *Web Developer's Guide*.

Panther Development Environment

Whether the applications that you build use a two-tier or three-tier processing model, Panther provides a distributed development environment that affords concurrent access to server libraries and repositories across the network. This can help give all members of your development team access to the same information and sets of standards via the file access server (`devserv`) in your distribution, which you can configure and manage through the JetNet manager (refer to page 3-19).

You can take advantage of Panther's built-in controls for monitoring multi-user access of shared libraries and their contents. You can also use features of your own source control management system. Panther provides an interface to SCCS and PVCS source code management systems. If you do not use or have one of these systems, Panther provides a default warning system for controlling concurrent access to shared application objects during the development process.

For more information about implementing configuration management, refer to page 10-4 in the *Application Development Guide*.

Administration Tools

In Panther software, a number of interactive and command-line utilities let you configure and manage a distributed development environment as well as a deployed three-tier and web application environment.

JetMan, the graphical JetNet Manager, integrates all the facilities you need to configure and manage the middleware component of a Panther application. With it, you create and edit a binary JetNet configuration file; this file specifies how to set up an application's clients and servers and configure their interaction.

The following command-line utilities are also provided:

- `rb2asc` converts a binary JetNet configuration file to ASCII and vice versa.
- `rbboot` starts a Panther application.
- `rbconfig` creates a JetNet configuration file.
- `rblisten` starts the listener process.
- `rbshutdown` shuts down a Panther application.
- `monitor` administers a running web application.

CHAPTER

2 Setting the Enterprise Environment

Panther typically runs in two types of environment: one tailored for development, the other for production. Each environment has different requirements—for example, a development environment is typically configured to offer shared access to libraries and repositories. Despite these differences, setting up a Panther environment for remote processing and access consists of these tasks:

- Create the server application directory and populate it with the required software and files.
- Set up the machine and server environments.
- Set up the client environment.
- Create the middleware configuration file.
- Modify the UNIX kernel or Windows registry to ensure adequate resources.

This chapter describes each task in detail; differences between development and production environments are noted where applicable.

For information about setting up a Panther application that is accessible via the web, refer to Chapter 2, "Web Application Setup," in the *Web Development Guide*.

Note: This chapter assumes that the Panther software is installed (refer to the *Installation*).

Setting Up the Enterprise Directory

The application directory is the area in a server machine where Panther runs. You must create this directory on each server machine and populate it with the required server executables, application libraries, and environment files.

Server Executables

Server executables are required for shared library access during development and for deploying three-tier applications. Copy or create symbolic links to the required server executables; in the default distribution, these are located in the `util` directory. Or set `PATH` to `$SMBASE/util` in the machine environment files. Panther provides four server executables:

- `devserv`—Available only in development environments (and in applications running reports remotely), *file access servers* offer shared access to libraries and repositories across the network. You can use this server to facilitate development of both two-tier and three-tier applications.
- `proserv`—Required for three-tier applications, this server executable is called by *standard servers* and is used to advertise and execute JIF-defined services.
- `prodserv`—(UNIX only) Identical to `proserv` with the Panther debugger linked in, you can run a server in debug mode if the server definition specifies this executable and specifies Debug (refer to page 3-23).
- `progserv`—(UNIX only) Required for three-tier applications that are converted from two-tier applications, this server executable is used by *conversion servers*. Conversion servers advertise their services only to converted client screens.

Note: On UNIX, the distributed server executables have the JDB database linked in. To run a server with another database, you must recompile the executable. For information on building executables, refer to Chapter 42, “Building Application Executables,” in *Application Development Guide*.

Application Libraries

The distribution provides three application libraries in the `samples/newapp` directory which you should copy to your application directory:

- `common.lib` stores application-wide objects such as JIF and JPL modules.
- `client.lib` stores client screens, menus, JPL modules, and other resources that are used in the client interface such as styles and pixmaps. Copy this library to the server's application directory only if you want to share access to its contents with other developers. `client.lib` should be accessible to a server only during development; in a deployed application, it is available only to clients.
- `server.lib` stores service components and routines.

Specify to open these libraries by setting the application variable `SMFLIBS` in the environment file of the host machine or in the standard server's environment file `proserv.env`:

Environment Files

A machine environment file `machine.env` supplements the environment that is established on machine activation. In addition, two other server environment files are provided for some server types: `proserv.env` (used by `proserv` and `prodserv` servers), and `progserv.env` (used by `progserv`). These can be used to supplement the environment for a given server. All files are in the distribution's `config` directory; copy these to your application directory.

You specify the machine environment file in JetMan through its Machine Environment Variable File property (refer to page 3-14); you can specify an environment file for each server through its Server Environment Variable File property (refer to page 3-21).

For information about environment file settings and format, refer to page 2-4.

Setting the Environment

On activation, all servers on a UNIX master machine inherit the environment of `rbboot`; servers on a non-master machine inherit the environment of `rblisten`; a Windows server inherits the environment of the `tuxipc` service. Each host machine's environment is supplemented by the settings of its environment file `machine.env`. All servers inherit their host machine's environment. Each server's environment can be further supplemented with the appropriate server environment file: `devserv.env`, `proserv.env`, or `progserv.env`. You must make sure that all necessary environment variables are properly set, either in the machine environment before activation, or through the appropriate environment file.

UNIX Environment

A UNIX host environment must have these variables set before the server is activated:

```
SMBASE=<root of Panther installation>

PATH=$SMBASE/util:$PATH

LM_LICENSE_FILE=$SMBASE/licenses/license.dat:$LM_LICENSE_FILE

LD_LIBRARY_PATH=$SMBASE/lib:$LD_LIBRARY_PATH
```

A `prodserv` server (one that has the debugger linked in) must also set the terminal display with this variable:

```
SMTERM=<terminal-type mnemonic>
```

Windows Environment

A Windows server must have these variables set:

```
SMBASE=<root of Panther installation>

ULOGDIR=<root of Panther installation>
```

```
PATH=$SMBASE\util;%PATH%
```

```
LM_LICENSE_FILE=%SMBASE%\licenses\license.dat;%LM_LICENSE_FILE%
```

On the Control Panel under System, check the Environment settings and update them if necessary.

Environment File Settings

On activation, each machine and server reads the environment files that are specified for it in the middleware's configuration file; in the JetNet manager, these files are set in the Machine Environment Variable File (page 3-14) and Server Environment Variable File (page 3-21) properties. Environment file settings supplement the environment that exists when the host and its servers activate.

Note: You cannot use environment files to override the machine settings for the Panther Install Directory, Application Directory, and Local JetNet Configuration File.

Each line in the environment file contains a variable assignment with the format *variable* =*value*, where *variable* starts with an underscore or alphabetic character and contains only underscores or alphanumeric characters. Within the value, strings of the form `${env}` are expanded using variables already in the environment. Forward referencing is not supported, and if a value is not set, the variable is replaced with the empty string. Use backslash (\) to escape the dollar sign or another backslash. All other shell quoting and escape mechanisms are ignored.

Machine Environment File

A machine environment file should contain these settings:

- Each file must set `SMBASE` to the Panther installation directory.
- On UNIX, unless you copied or linked to the server executables in the application directory, the machine environment file must set `PATH` to `${SMBASE}/util`. On Windows, `PATH` must be set to `${SMBASE}\util` in order to support DLLs. On both platforms, this setting is added to the previously established setting to yield the following `PATH`:

  UNIX: `${APPDIR}:${SMBASE}/bin:/bin:${SMBASE}/util`

  Windows: `${APPDIR};${SMBASE}\bin;${SMBASE}\util`

PATH is set to ${APPDIR}:${SMBASE}/bin:/bin:*path*, where *path* is the value of the environment file's last PATH= entry.

- If servers on a UNIX host have the debugger linked in, LD_LIBRARY_PATH must be set to Motif shared libraries and the DISPLAY environment to the display on which debugging takes place.

  Setting LD_LIBRARY_PATH in the environment file adds to the previously established setting:

  APPDIR:${SMBASE}/lib:/lib:/usr/lib:*lib*

  where *lib* is the value of the environment file's last LD_LIBRARY_PATH= entry.

  Note: On HPUX, use SHLIB_PATH in place of LD_LIBRARY_PATH; on AIX, use LIBPATH.

- Optionally, set SMFLIBS to any libraries that clients and servers need to access. If SMFLIBS is not set in the machine environment file, it must be set for each server individually in its server environment file, and for native clients through the client's setup file (in UNIX) or initialization file (Windows).

- Optionally, set SMPATH to include any directories that servers need to access besides the application directory (set through the JetNet Manager—refer to page 3-14). If the environment file contains an SMPATH entry, make sure that the path includes the config directory.

  For example, the following entry in a machine environment file ensures that servers on this host have access to files in the myapps and config directories as well as in the application directory:

  UNIX: SMPATH=/u/myapps|${SMBASE}/config

  Windows: SMPATH=c:\myapps|${SMBASE}\config

  Note: The path to the application directory is set in the machine's configuration through its Application Directory property (page 3-14). SMPATH must not include the same directory again.

Server Environment File

A server environment file should contain these settings:

- In proserv.env, set SMFLIBS to server.lib and common.lib so these libraries open at application startup:

```
SMFLIBS=server.lib|common.lib
```

- If the application has been upgraded from two-tier to three-tier architecture, set `SMFLIBS` in `progserv.env` to the service component library created by the upgrade utility `clnt2svr`:

  ```
  SMFLIBS=sv.lib
  ```

- Optionally, set `SMPATH` to include any directories that the server needs to access besides the application directory (set through the JetNet Manager—refer to page 3-14). If the environment file contains an `SMPATH` entry, make sure that the path includes the `config` directory.

  Note: Because a server's application directory is already set through its machine's configuration (page 3-14), `SMPATH` must not include the same directory again.

Interfacing with SCCS/PVCS

When using remote libraries that are under SCCS/PVCS control, files are checked out of the library under the name of the user who started the application rather than the client who made the request. This makes it appear as if all work is performed by a single user, and is probably not desirable.

To workaround for this on UNIX, the check-in/check-out daemon must be able to pretend to be any user. You may achieve this using our `devserv` by changing it to be owned by `root` and setting the suid bit.

If this is not possible in your configuration, or presents too much risk, there is an alternate solution. The Panther software contains an alternate check-in/check-out daemon, named `cfgserv`. It, too, must be owned by `root` and have the suid bit set, but you can eliminate the security risks by running `devserv` under a dedicated group id and making `cfgserv` executable only by that group.

`cfgserv` is located in `$SMBASE/util`; the source code is located in `$SMBASE/link`.

SMVARS Settings

You can test an application with the Panther debugger by running it directly on the server. To do so, the server's `SMVARS` file must set configuration variables `SMTERM` and `SMKEY` so that Panther's interacts properly with the terminal's display and keyboard.

Setting Up the Client Environment

Two types of clients can connect to a Panther application: native and workstation clients.

- A *workstation client* (or remote client) resides on a machine that is not defined in the application configuration.
- A *native client* (or local client) resides on a server machine—that is, a machine that is defined in the application configuration.

When you create an application using JetMan, the machine name is specified automatically and is then stored as part of the application configuration in `broker.bin`.

Each type of client has its environment set differently, as described in the following sections.

Workstation Clients

Each workstation client's environment must be set up in two ways:

- The configuration variable `SMFLIBS` must be set to enable shared access to libraries across the network.
- Configuration variables `SMRBHOST` and `SMRBPORT` must be set so that clients can connect to a Panther application.

Setting Access to Libraries

Access to application libraries—`client.lib`, `server.lib`, and `common.lib`—is set through the configuration variable `SMFLIBS`. You can set `SMFLIBS` in one of two ways, depending on whether you want to provide shared access to these libraries:

- To allow shared access to application libraries, set `SMFLIBS` on each workstation as follows:

```
SMFLIBS=host!client.lib|host!common.lib|host!server.lib
```

host is the server machine on which these libraries reside. For example, the following `SMFLIBS` setting specifies that all application libraries are located on host `aspen`:

```
SMFLIBS=aspen!client.lib|aspen!common.lib|aspen!server.lib
```

- If you want developers to have their own libraries, copy the application libraries to each workstation. The workstation's setting for `SMFLIBS` can remain as set in the default distribution. For example:

  ```
  SMFLIBS=client.lib|common.lib|server.lib
  ```

- If you installed a web application server, it cannot access remote libraries on startup using `SMFLIBS=` *server_id!lib_name*. If a client library cannot be placed on a shared file system, connect to the middleware on application startup using `client_init` and execute the Panther library function `sm_l_open` to open a remote library.

For Windows workstations, `SMFLIBS` must be set in the initialization file: `prol5w32.ini` for Panther, `mbedit32.ini` or `mbedit.ini` for the menu bar editor, and `jifedit32.ini` or `jifedit.ini` for the JIF editor. For UNIX clients, you can set `SMFLIBS` in the environment, or in the `SMVARS` or `SMSETUP` file.

Note: Make sure that, during development, `SMFLIBS` specifies client libraries before server libraries; if screens in client and server libraries use the same name, this ensures that the client version opens when invoked.

Enabling Client Connections

To establish a workstation client connection, two conditions must be true:

- One or more active server machines must be enabled for workstation connections (refer to page 3-15).
- The workstation must know the network address of a server machine that is enabled for workstation connections.

Two Panther configuration variables, `SMRBHOST` and `SMRBPORT`, let you specify the network addresses of one or more server machines. For example, you might set the two variables as follows:

```
SMRBHOST=aspen,marks,sparks
```

```
SMRBPORT=37000,37100
```

Given these settings, the workstation client initially tries to connect to `aspen` at port 37000; if unable to connect to `aspen`, it next tries to connect to `marks` at port 37100. Failing both connections, it tries `sparks` at port 37100.

Set `SMRBHOST` and `SMRBPORT` in one or more of the following initialization files:

- In the application initialization file.
- To enable client connections from Panther editors: in `prol5w32.ini` for the screen editor, `mbedit32.ini` for the menu bar editor, and `jifedt32.ini` for the JIF editor.
- To enable workstation connections to the JetNet manager, in `jetman32.ini`.

For more information about setting `SMRBHOST` and `SMRBCONFIG`, refer to the *Configuration*.

Native Clients

Native clients use the value in `SMRBCONFIG` to establish their connection to the middleware. Make sure that the client machine's `SMRBCONFIG` matches the value specified for the machine's local configuration file (refer to page 3-14).

If the client resides on a UNIX machine, its environment must have the `LD_LIBRARY_PATH` variable (or its equivalent) set to the required JetNet or Oracle Tuxedo libraries.

Middleware Configuration File

A middleware configuration file is required to develop and deploy three-tier applications; it is also required for shared library access in a development environment. This file contains all the information that the middleware requires to provide server access: which machines are configured for servers, the types of servers that are available, network settings that enable workstation client connections, and so on.

Panther has its own utilities for creating and editing a configuration file for the JetNet middleware. You can create a minimal JetNet configuration file through the utility `rbconfig`; you can also create and edit JetNet configuration files through the JetNet manager (page 3-1).

An application using the Oracle Tuxedo middleware adapter can be run with a JetNet configuration file. You can also use Oracle Tuxedo utilities to create and edit a Oracle Tuxedo configuration file.

Warning: A configuration file that is created with Oracle Tuxedo utilities must be used only with the Oracle Tuxedo middleware adapter.

You specify the location of the middleware configuration file through the configuration variable `SMRBCONFIG`.

Setting IPC Resources

After you install Panther and before you start development, you might need to adjust resource settings for the IPC (interprocess communication) subsystem in your UNIX kernel or Windows registry.

On UNIX, you set IPC resources in the system kernel. You can check the status of resources using the UNIX command `ipcs`.

On Windows, the Panther installation adds entries to the registry and adds an Windows service to the system. This allows JetNet features to work with the embedded Oracle Tuxedo software. Registry entries can be edited in order to facilitate system management and debugging.

Registry entries are added under this layout:

```
HKEY_LOCAL_MACHINES\System/CurrentControlSet
    Control\Session Manager\Environment
    Services\TUXEDO IPC HELPER
    Enum\Root\LEGACY_TUXEDO IPC HELPER
```

```
HKEY_LOCAL_MACHINE\Software\ORACLE\TUXEDO\6.3
    Environment
    Security
```

You can view the Oracle Tuxedo IPC HELPER service via from the Services applet, invoked from the Control Panel. This applet lets you perform tasks such as starting or stopping the service.

You must stop the Oracle Tuxedo IPC HELPER service before you uninstall the Panther application server; otherwise, the service and its DLLs are still available after server deinstallation.

The following sections describe IPC resources for message queues, semaphores, and shared memory. The descriptions include default values; these are usually adequate for most Panther applications unless otherwise indicated.

Messages

Panther uses messages and message queues to transfer data between clients and servers. All instances of a server share a single message request queue, while each client has its own response queue. Individual server instances each get their own reply queues if the server's Enable Service Calls property (in the JetNet configuration file) is set to Yes. If the limits set by message parameters are exceeded, additional server instances and clients cannot start or a blocking condition occurs. If all attempts to send messages by processes are blocked, application deadlock occurs.

UNIX

`MSGMNI`

Specifies the number of unique message queue identifiers. The default is 50, which may be too low. To calculate the correct value, count one queue per process, including one for each BBL and the DBBL, one for each client and server, and two for each bridge process. If a server's Enable Service Calls property (in the JetNet configuration file) is set to Yes, also count one queue per server instance.

`MSGMAP`

Specifies the size of the control map that manages message segments. Set this to twice the value of `MSGMNI`.

`MSGMAX`
: Specifies the maximum message size in bytes, typically between 16K and 32K. Any message that is larger than 75 percent of `MSGNB` is sent via file transfer.

`MSGMNB`
: Specifies the maximum message queue length in bytes. This parameter determines the amount of data that can be placed on a message queue at any time. It is typically set to the same value or higher as `MSGMAX`.

`MSGSSZ`
: Determines the size of a message segment. Each message contains contiguous segments large enough for the message text. A size of 32 is typical.

`MSGTQL`
: Specifies the number of outstanding messages. The default is 40 but should be set higher than 200 for a system with a high volume of traffic.

`MSGSEG`
: Specifies the number of message segments in the system. This value multiplied by `MSGSSZ` should be less than or equal to 128K for most platforms. Refer to your UNIX documentation for more information.

Windows

`TUXIPC_MSG_BYTES`
: Maximum allowed message size. The default is 65536.

`TUXIPC_MSG_HDRS`
: Maximum number of message headers. The default is 8128.

`TUXIPC_MSG_QUEUE_BYTES`
: Maximum message queue size. The default is 65536.

`TUXIPC_MSG_QUEUES`
: Maximum number of message queues. The default is 256.

`TUXIPC_MSG_SEG_BYTES`
: Size of the message segment. The default is 64.

`TUXIPC_MSG_SEGS`
: Number of message segments. The default is 32767.

Semaphores

Each process that attaches itself to a bulletin board acquires a semaphore, which are means of passing flags from process to process. Semaphores can be tuned by the following parameters:

UNIX

`SEMMNS`
: Specifies the maximum number of semaphores in use in the system. The default for most UNIX System V implementations is 60. Check the JetNet configuration file's Max Accessors settings for individual machines (refer to page 3-11).

`SEMMNI`
: Specifies the maximum number of active semaphore sets. The default is 10.

`SEMMSL`
: Specifies the maximum number of semaphores per semaphore set. The default is 25, which should be enough for 200 accessors. Panther will use up to eight sets, so `SEMMSL` should be set to the total number of accessors divided by eight. It is common for `SEMMNS` to equal `SEMMSL * SEMMNI`.

`SEMMAP`
: Specifies the size of the control map used to manage semaphore sets. Set this to the same value as `SEMMNS`.

`SEMMNU`
: Specifies the number of undo structures in the system. This controls the number of UNIX system processes that attempt to access the bulletin board at the same time. If more than this number try to connect to the bulletin board, a UNIX system error results.

`SEMUME`
: Specifies the maximum number of undo entries per process. Set this to the same value as `SEMMNS`.

Windows

`TUXIPC_SEM`
: Maximum number of semaphores. The default is 1024.

`TUXIPC_SEM_IDS`
: Maximum number of semaphores set. The default is 1024.

`TUXIPC_SEM_UNDO`
: Maximum number of semaphore undo structures. The default is 1024.

Shared Memory Requirements

UNIX

`SHMAX`
: Specifies the maximum size in bytes of an attached shared memory segment, up to 4194304 (4 megabytes). The default is 524288.

`SHMSEG`
: Specifies the maximum number of shared memory segments that a process can attach to, up to 10. The default is 6.

`SHMMNI`
: Specifies the maximum number of shared memory identifiers in the system. The default is 100.

Windows

`TUXIPC_SHM_PROCS`
: Number of processes per shared memory mappings. The default is 500.

`TUXIPC_SHM_SEGS`
: Number of shared memory segments. The default is 50.

CHAPTER

3 Configuring the Enterprise

The graphical JetNet Manager (JetMan) integrates all the facilities you need to configure and manage the middleware component of a Panther application. With it, you create and edit a binary JetNet configuration file; this file specifies how to set up an application's clients and servers and configure their interaction—for example, whether multiple workstations can attach to the application, the maximum number of machines, servers, and services that the application can support, how many servers to activate and on which machine.

JetNet configuration files can be used with the Tuxedo middleware adapter, and are accessible to all Tuxedo utilities, including `xtuxadm`. Thus, you can use the JetNet manager to create a configuration file, then edit and enhance it for use with Tuxedo later on.

Note: The JetNet manager is designed for editing JetNet configuration files only; you should edit and manipulate Tuxedo configuration files with the appropriate Tuxedo utilities.

This chapter shows how to use the JetNet manager to create and configure an application. It describes the interface and the properties that you can set for each object— application, machine, and server. For information about using the JetNet manager to manage a running application, refer to page 4-1.

Using the JetNet Manager

You invoke the JetNet manager through the executable `jetman`. When you start JetNet manager on UNIX, it automatically reads its `Jetman` resource file; on Windows, it reads the `jetman32.ini` initialization file. It then reads the configuration file specified by `SMRBCONFIG` or `TUXCONFIG`; if neither is set, the JetNet manager opens without a configuration file.

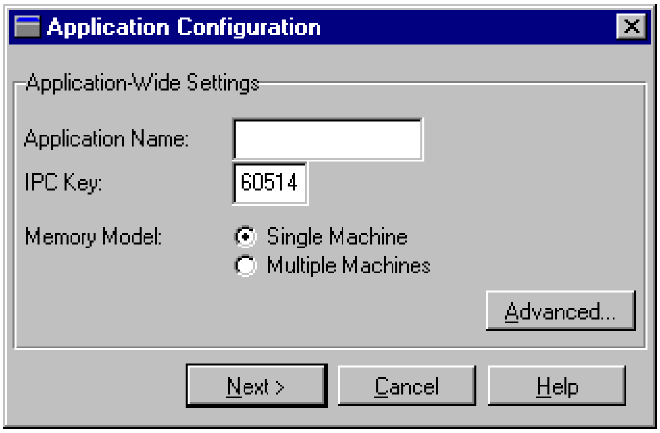
The following steps create a basic configuration file. For more information about each field, refer to page 3-7.

Creating a Configuration File

How to Create a Basic Configuration File

1. Choose File→New Application from the menu bar or the New Application button from the toolbar. If another configuration file is already selected, the JetNet manager asks whether to close it.

   The JetNet manager displays the Application Configuration dialog:


Application Configuration

Application-Wide Settings

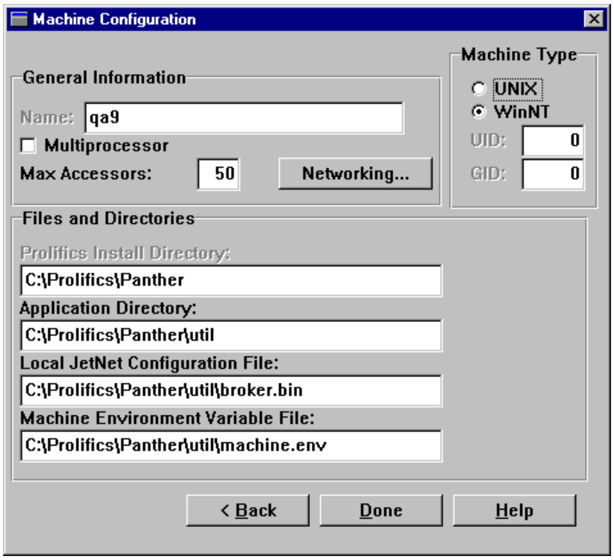
Application Name:

IPC Key: 60514

Memory Model: [✓ Single Machine] Multiple Machines

2. Assign a name to the application through the Application Name property. Other application properties are optional; set these as desired (refer to page 3-8).

3. When you finish editing application properties, choose Next. The JetNet manager displays the Machine Configuration dialog, where you configure the master machine:


Machine Configuration

General Information

Name: qa9

Multiprocessor

Max Accessors: 50

Machine Type: UNIX [✓ WinNT]

UID: 0

GID: 0

Files and Directories

Prolifics Install Directory: C:\Prolifics\Panther

Application Directory: C:\Prolifics\Panther\util

Local JetNet Configuration File: C:\Prolifics\Panther\util\broker.bin

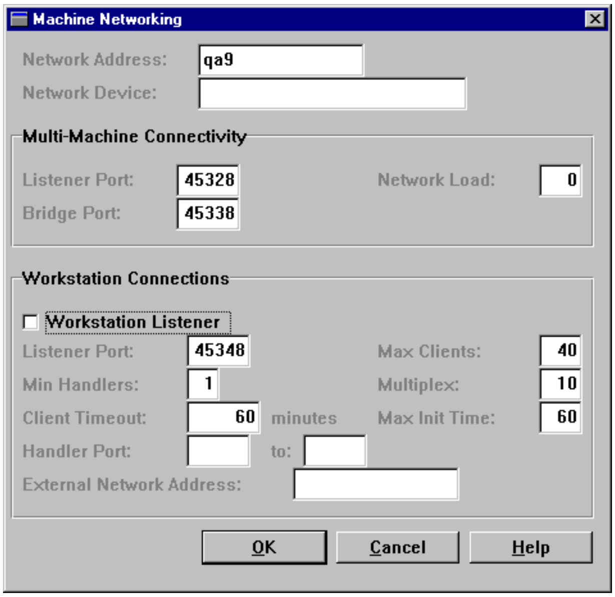
Machine Environment Variable File: C:\Prolifics\Panther\util\machine.env

4. Configure the master machine properties as desired (refer to page 3-12). The Name property is initially set to the machine on which you are running the JetNet manager.

   Note: In a multi-machine application, this machine is initially defined as the master machine. Later, you can reassign this machine to a non-master role after other machines are added (refer to page 3-8).

5. Check the path specified for the Panther installation, the application directory, and the configuration file. These files should be located on the same machine.

6. When you finish setting machine properties, choose Networking.



Network Address:	qa9
Network Device:	

Multi-Machine Connectivity

Listener Port:	45328	Network Load:	0
Bridge Port:	45338		

Workstation Connections

☐ Workstation Listener

Listener Port:	45348	Max Clients:	40
Min Handlers:	1	Multiplex:	10
Client Timeout:	60 minutes	Max Init Time:	60
Handler Port:		to:	
External Network Address:			

7. To enable remote access, select Workstation Listener, and choose OK.

8. When you finish setting machine properties on Machine Configuration, choose Done. If the file name specified in Local

JetNet Configuration File already exists, a message asks for confirmation to overwrite it.

Note: You can create a minimal configuration through the utility `rbconfig`.

Editing a Configuration File

The JetNet manager's opening Application Status dialog shows the hierarchy of components in an application. Initially, only the top-level component—for the application itself—is shown. Double-click on each component in the list to toggle in and out of view components that are one level below. Or you can expand and collapse the list of all components below the selected component by choosing View→Expand Subtree and View→Collapse Subtree. A component with subordinates is prefixed with a - or + symbol to indicate whether they are visible or not.

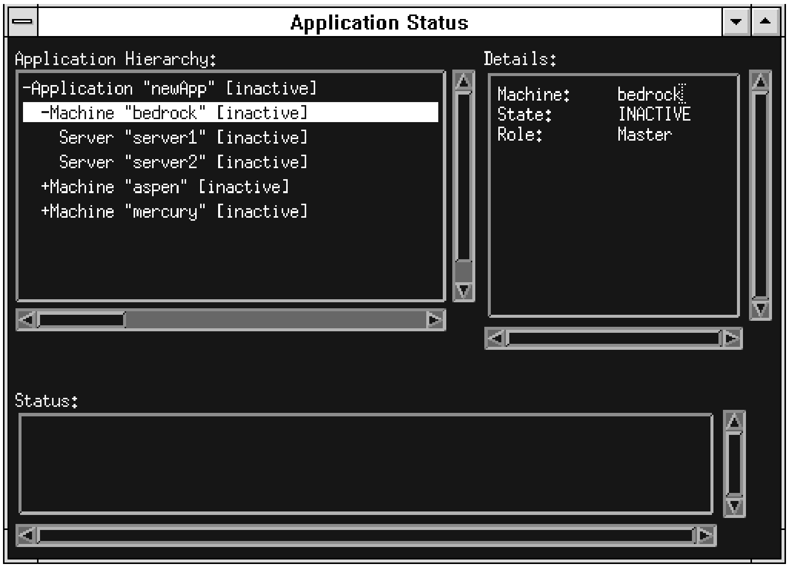
Note: In an inactive multi-machine application, you can edit the configuration file from the master machine only.

When you edit an inactive application's configuration, the hierarchy list can contain three types of components:

- Application—The topmost item in the hierarchy list; the JetNet manager can view only one application at a time.
- Machine—One or more machines that are configured for an application, listed below the application name. In a multi-machine application, machines are listed in the order in which they are added.
- Servers—Servers are listed under the machine for which they are defined. In an active application, a machine that is enabled for workstation connections also shows its workstation handler as a server.

You can also edit the configuration file of an active application. Some component properties are editable only when the component itself is inactive. For more information about managing an active application, refer to page 4-1.

To the right of the hierarchy list, the Details list displays information about the selected component, such as its name, type, and state (active or inactive). The dialog also contains a Status box that displays informational messages.



```
Application Hierarchy:
-Application "newApp" [inactive]
  -Machine "bedrock" [inactive]
    Server "server1" [inactive]
    Server "server2" [inactive]
  +Machine "aspen" [inactive]
  +Machine "mercury" [inactive]
```

```
Details:
Machine:    bedrock
State:      INACTIVE
Role:       Master
```

```
Status:
```

You can use the hierarchy window to add and delete components from an application's configuration; you can also use it to access those components' definitions and edit them:

Adding and Deleting Components

Add servers and machines to a configuration by selecting a higher-level object—for a new server, its host machine; for a new machine, the application—and choosing the appropriate File⟶New option or toolbar button. Servers and machines can be added and removed whether their parent objects are active or inactive.

You can also create a copy of an existing server by selecting the server and choosing File⟶New⟶Server or the corresponding toolbar button. This adds a new server to the host machine and copies all non-unique properties from the selected server to the new one. You must assign the new server a unique name in order to save it.

To delete a component from the configuration, select it and choose Edit⟶Delete. You can delete any inactive component from the configuration. If the selected component has subordinate components, these are deleted also.

Editing Components

A JetNet configuration file defines three types of application components—the application itself, machines, and servers. You access these definitions through the JetNet manager. Definitions can be viewed and edited whether a component is active or not; however, you can change some settings only when the component is inactive. All changes that you make in the JetNet manager are written to the configuration file immediately.

To edit a component's settings, select its name from the hierarchy list and choose Edit→Properties or the Object Properties button from the toolbar. Properties for each component type are described in later sections of this chapter.

Selecting Another Enterprise's Configuration

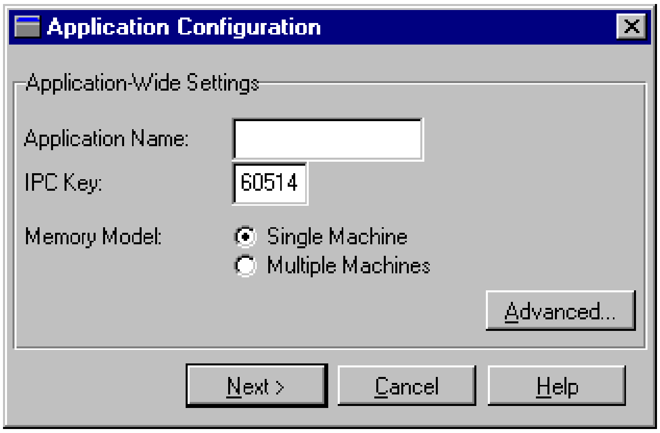
To edit another configuration file, choose File→Select Application. This invokes the Select File dialog where you choose the desired configuration file. When you select a file, Panther reads the configuration definition. If the JetNet manager was connected to the previous application (refer to page 4-4), the connection is automatically broken.

Setting Enterprise Properties

To access application properties, select the application name from the hierarchy list and choose Edit→Properties.

General Settings

The application properties that are initially displayed set an application's identity—its name, IPC key, and whether it runs on a single or multiple machines. For more advanced application properties, choose the Advanced push button.



Application Name:
IPC Key: 60514
Memory Model: [✓ Single Machine] Multiple Machines

Application Name

The application's logical name, a value of up to 30 characters. This name identifies the application in the hierarchy list.

IPC Key

Specifies the application's address of shared memory. The default value is system-assigned. You can assign a value between 32,769 and 262,142, inclusive. Panther uses this address to allocate application IPC resources so that they can be located easily by new processes as they join the application. This key is used internally to allocate the bulletin board, message queues, and semaphores that must be available to processes when they join the application.

Memory Model

Specifies whether the application runs on a single processor or across a network with multiple machines:

- Single Machine—Run the application on a single machine.
- Multiple Machine—Run the application on multiple machines.

When you choose this Multiple Machine, the current machine is automatically assigned the role of master machine. You can add additional machines to the network by choosing New—Machine from the File menu or the New Machine button from the toolbar.

Roles

This push button is accessible only for multi-machine applications. In a networked application, you must identify the master machine and, optionally, the machine that acts as its backup. The first machine to be defined for the application is the default master machine.

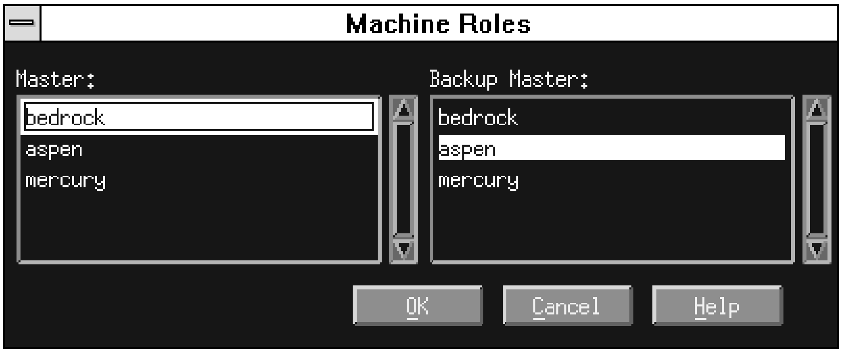
You should always specify a backup master machine; this is especially important for 24-hour applications to allow a smooth transfer of operations in case the master machine unexpectedly ceases operation or is deliberately brought down for periodic maintenance. In a running application, the backup master is always prepared to take over the role of master machine.

In an inactive application, you reassign the master machine by running the JetNet manager from the machine that you want to designate as the new master. You must always reassign the backup master from the master machine.

In a running application, you can reassign machine roles by running the JetNet manager from any active machine.

How to Assign or Reassign Machines Roles

1. From the Application Configuration dialog, choose the Roles push button. This invokes the Machine Roles dialog:



Machine Roles

Master: bedrock (selected), aspen, mercury

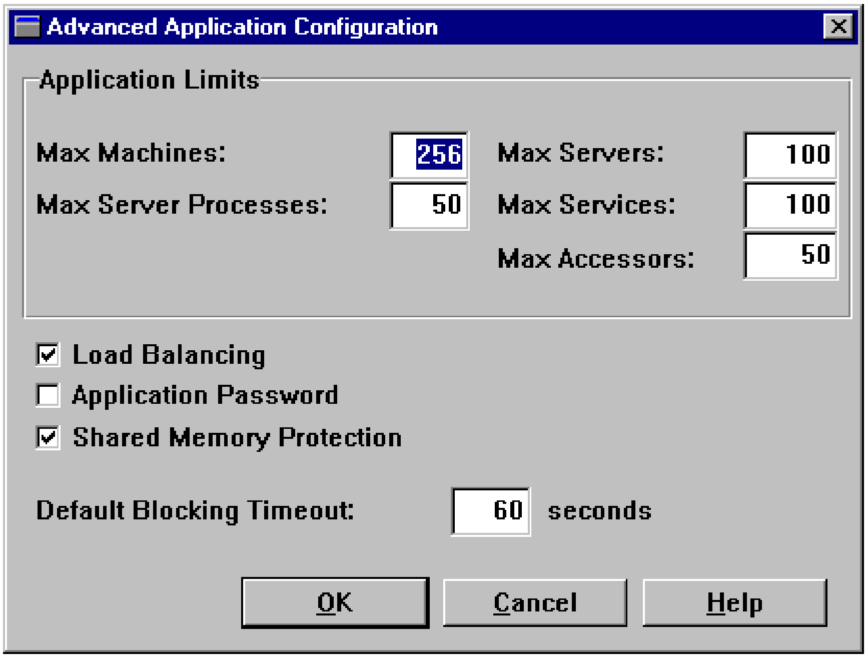
Backup Master: bedrock, aspen (selected), mercury

Two list boxes show all machines in the application. The Master list highlights the current master; the Backup Master list highlights the current backup master, if any.

2. Select the desired machine from the appropriate list. When your assignments are complete, choose OK.

Advanced Settings

The Advanced Application Configuration dialog sets limits such as the maximum number of machines and servers that the application supports, and general application properties such as the application password.


Advanced Application Configuration

Application Limits

Max Machines:	256	Max Servers:	100
Max Server Processes:	50	Max Services:	100
		Max Accessors:	50

- ☑ Load Balancing
- ☐ Application Password
- ☑ Shared Memory Protection

Default Blocking Timeout: 60 seconds

Max Machines

The maximum number of machines that can be used by the application. The default value is 256.

Max Server Processes

The maximum number of server processes allowed for the application at any one time. This value must be greater than 0 and less than 8192. The default value is 50.

The value that you should set for this property should account for one listener server on each workstation, and the maximum number of instances anticipated for each server. Because this property directly increases semaphore and shared memory costs, deployed applications should have this property set to the lowest value that ensures acceptable performance. In a development environment, set this property sufficiently high to account for additional resource requirements.

Max Servers

The maximum number of servers allowed for the application at any one time. The value that you set for this property should account for one listener server on each workstation. This property's value must be between 100 and 32,767, inclusive. The default value is 100.

Max Services

The maximum number of services that can be advertised by the application at any one time. This property's value must be between 1 and 32,767, inclusive. The default is 100. Because this property directly increases shared memory costs, set it to the lowest value that allows the application to advertise all desired services.

Max Accessors

Specifies the initial value that is set for a new machine's Max Accessors property (refer to page 3-13).

Load Balancing

Determines whether to load balance client requests among active servers. When load balancing is on, JetNet monitors the total number of all services waiting to be handled by each server. It uses this information to direct requests to the least-busy server that can handle each new request.

All requests are equally weighted; this means that JetNet simply calculates a given server's load from the total number of services waiting to be processed, and ignores their cumulative processing time. To make maximum use of load balancing, organize services into service groups according to their processing requirements and associate these groups with servers as needed. You can also optimize throughput on the machine level by setting Network Load for each machine (refer to page 3-16).

Note: Load balancing is not required when services are offered by only one server.

Application Password

If enabled, this check requires clients to supply a password when joining the application. This password is checked against the password supplied by the system administrator.

When Password is enabled, you supply or change the application password by choosing Change Password. This invokes the Application Password dialog where you enter the new password twice. If the two entries do not match, an error message is posted and you must repeat the process.

The application password provides level-2 authentication. Level-3 authentication is provided in the Oracle Tuxedo version. For more information, refer to page 2-13 in the *Programming Guide*.

Shared Memory Protection

Shields the system tables that are kept in shared memory from clients and servers. Enable this option when you are developing an application that is not yet fully debugged and tested. In finished applications, disable this option for faster response time.

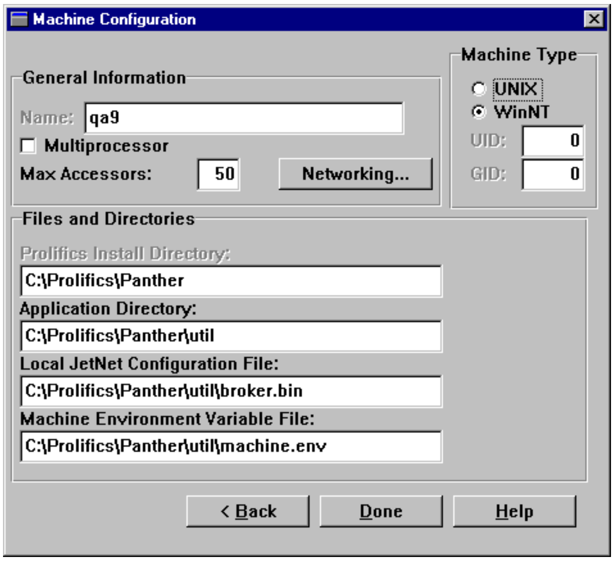
Default Blocking Timeout

Specifies how much time elapses before a call times out and returns to its caller. Calls are blocked only if the `tp_block` property is set to `PV_YES`. The default is 60 seconds. If the application uses a server executable that has the debugger linked in, increase this value sufficiently to allow for slower response time.

The entered value must be a multiple of 10. If an entry's last digit is non-zero, the JetNet manager changes it to 0.

Setting Machine Properties

A Panther application can run on a single machine or across a network on multiple machines. To access machine properties, select the machine name from the hierarchy list and choose Edit—Properties.


Machine Configuration

General Information
Name: qa9
☐ Multiprocessor
Max Accessors: 50
Networking...

Machine Type
○ UNIX
● WinNT
UID: 0
GID: 0

Files and Directories
Prolifics Install Directory:
C:\Prolifics\Panther
Application Directory:
C:\Prolifics\Panther\util
Local JetNet Configuration File:
C:\Prolifics\Panther\util\broker.bin
Machine Environment Variable File:
C:\Prolifics\Panther\util\machine.env

< Back | Done | Help

Name

A logical name that identifies this machine to the network. Names should not include an embedded period (.) character.

Multiprocessor

Enable if the machine has multi-processor capabilities.

Max Accessors

The maximum number of processes, clients and servers, that can attach to the application on this machine. The initial value is set from the application's Max Accessors property. Because this property directly increases semaphore and shared memory costs, set it to the lowest value that ensures acceptable performance.

Machine Type

In a multi-machine application, identifies the machine type, either UNIX or Windows. This setting enables JetNet to manage communication between machines of different types. When you define a new machine, its machine type is initially set to the machine on which the JetNet manager is running. You can change this setting for any non-master machine; it is inaccessible for the application's master machine.

A UNIX machine can also specify its user and group IDs, UID and GID. When you define a new UNIX machine, its UID and GID values are initially set to the UID and GID of the machine on which the JetNet manager is running, if available. You can change these values to any integer between 0 and 2,147,483,647, inclusive.

A Windows machine prevents access to the UID and GID properties and sets them to 0.

Panther Install Directory

Shows where Panther is installed.

Application Directory

The working directory where servers running on this machine are started. The machine's log file (`ULOG`*mmddyy*) is also written to this directory.

Machine Environment Variable File

Specifies to execute this machine with the environment in the named file. This property's default is set to `machine.env` in the application directory. You can override the default by entering the pathname directly or by using the Browse push button to select the desired file.

This environment supplements the one already established when the machine is activated. For example, you can set a different server library for each server by setting `SMFLIBS` in its environment file. You cannot use this file to override settings in Panther Install Directory, Application Directory, and Local JetNet Configuration File.

For more information about the contents and format of a machine environment file, refer to page 2-5.

Local JetNet Configuration File

For the master machine, specifies the JetNet configuration file that is read when the application is booted. For non-master machine, this property specifies the pathname of the local copy of the configuration file. When this machine is activated, it copies the master machine's JetNet configuration file into this location.

If you are defining a new configuration file, the JetNet manager creates a file according to the pathname specified and stores your initial settings in it. Settings for `SMRBCONFIG` must *exactly* match this pathname.

Note: Under shared file systems such as NFS, make sure that local configuration files on different machines have unique pathnames.

Warning: The configuration file created by JetNet manager or `rbconfig` includes references to itself. If you change the master machine's Local JetNet Configuration File property, these internal references become invalid. To rename a configuration file, convert it to ASCII with `rb2asc` and edit the `TUXCONFIG` string to the new file; then run `rb2asc` again to convert the edited file back to binary.

Network Settings

Networking settings specify the information that each workstation requires to contact other computers and vice versa.

IP Address

This machine's network identifier, valid only for multi-machine configurations. The JetNet manager assigns a value to this property based on the machine's IP address. Modify this value if the machine is known by multiple IP addresses and you need to use a different one.

Network Device

Specifies a machine's network device. The default value is platform-specific—for example, `/dev/tcp`.

Listener Port

Identifies the port that is used by this machine's listener process. Enter any number between 32,768 and 65,535, inclusive. Be sure to reserve this port number for Panther use only. The default value is the default IPC Key value plus 10.

At boot time there is no bridge process to receive communication. Instead, each listener process on the non-master and backup master machines awaits a message from the master machine to begin the local boot process. The master machine uses the port number in each machine's Listener Port property to address its listening process. Consequently, the port number supplied to `rblisten` to invoke a machine's listening process must match this property.

For more information about `rblisten`'s role in booting a Panther application, refer to page A-7.

Bridge Port

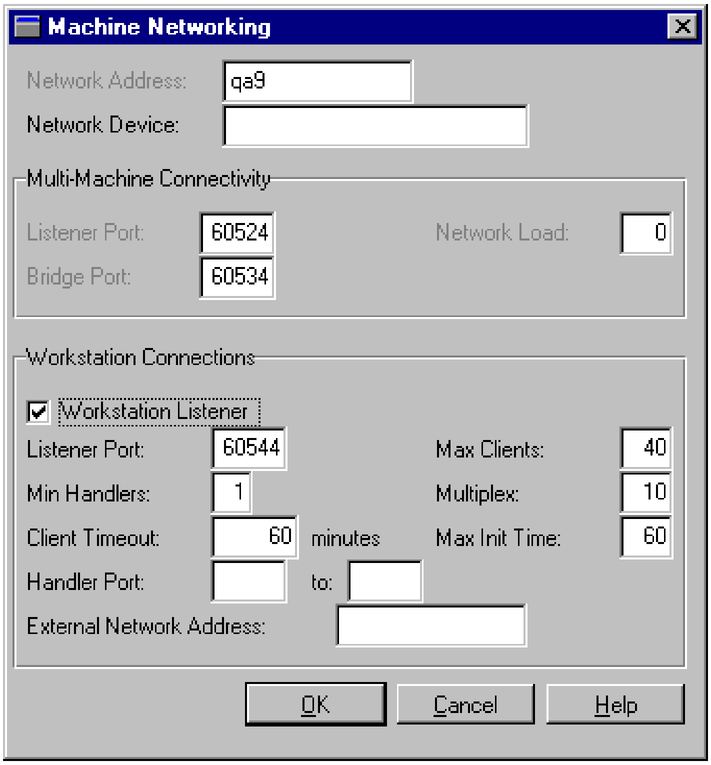
Identifies the port that this machine's bridge process uses to communicate with other machines in the network. Enter any number between 32,768 and 65,535, inclusive. The default value is the application's default IPC key value plus 20.

When the machine's bridge process is booted, it appropriates this port for its own use; consequently, you should reserve this port number for Panther.

Network Load

Use this property to inflate the activity of servers on this machine as perceived by clients on remote nodes. Activity is measured by the number of enqueued service requests; so two servers on different machines with the same number of enqueued requests appear equally available to all clients, regardless of their network proximity. The value specified for Network Load augments this number for remote clients. You can enter any value between 0 and 32,767, inclusive. The default value is 0.

For example, a network of two machines, `fred` and `wilma` has one server on each, `fred_srv` and `wilma_srv`. If `fred_srv` has 6 requests pending and `wilma_srv` has 9, `fred_srv` ordinarily gets the next service call, whether the caller is local or remote. Because routing calls across the network is expensive, you can set each machine's Network Load property to a level that deters excessive network traffic. So, if `fred`‘s Network Load property is set to 10, the next remote client with a service call sees `fred_srv` as having 16 pending requests, so the request broker routes the call to `wilma_srv`; however, a local client always sees `fred_srv's` level of activity as it actually is, so its next call is processed locally.


Machine Networking

Network Address: qa9
Network Device:

Multi-Machine Connectivity
Listener Port: 60524
Bridge Port: 60534
Network Load: 0

Workstation Connections
[✓] Workstation Listener
Listener Port: 60544
Min Handlers: 1
Client Timeout: 60 minutes
Handler Port: to:
External Network Address:
Max Clients: 40
Multiplex: 10
Max Init Time: 60

Workstation Connections

In order to permit workstation clients to connect to the application on this machine, enable the Workstation Listener toggle button. This starts the workstation listener process on this machine. When `client_init` is invoked from a workstation, it uses the workstation's settings in `SMRBHOST` and `SMRBPORT` to request a client connection. The workstation listener at this host and port intercepts the connection request and, if possible, finds a handler for the connection.

Note: When you add a machine to an application's configuration from a PC workstation, you cannot enable the machine's Workstation Listener toggle button (on the machine's Network Settings dialog). To activate a machine's workstation listener process, you must run JetNet manager on the server machine.

You configure a machine for workstation client connections with the following properties:

Listener Port
: Identifies the port that is used by the workstation listener. Workstation clients use this port number when they try to connect to the application on this machine. This property's value must be between 32,768 and 65,535, inclusive. Be sure to reserve this port number for Panther use only. The default value is the application's default IPC key value plus 30.

  For more about enabling workstation client connections, refer to page 2-9.

Min Handlers
: The minimum number of handlers that this machine's workstation listener makes available to clients at any given time. The workstation listener starts this many handlers when it is booted and makes sure that the number of handlers never decreases below this minimum until it shuts down. More handlers are made available as needed in order to accommodate clients that attach to the application on this machine, up to the value set for Max Clients. You can specify up to 255 handlers. The default value is 1.

Client Timeout
: The amount of time in minutes that a client can idle before it is disconnected by its workstation handler. A client is considered idle if it makes no requests. Clients that get unsolicited message notifications without responding are also considered idle. Use this option for client platforms that are unstable—for example, a workstation that can be turned off without calling `client_exit`.

  To prevent timeouts, set this property to blank. The default value is 60.

Handler Port
: Identifies the port range that is used by the workstation listener. By default, the values are between 2048 and 65535, inclusive.

External Network Address
: The network address used by the workstation listener as its listening address. Enter the host machine and port number as a string following two slashes or as a hex value. For example, if the machine is `bedrock` with a TCP/IP address of 123.1.123.12 and the port number is 9876, the network address could be specified as any of the following:

```
//bedrock:9876
```

```
//123.1.123.12:9876
```

```
0x000226947B017B0C
```

Max Clients
: The maximum number of clients that can attach to the application on this machine. The default value is 40.

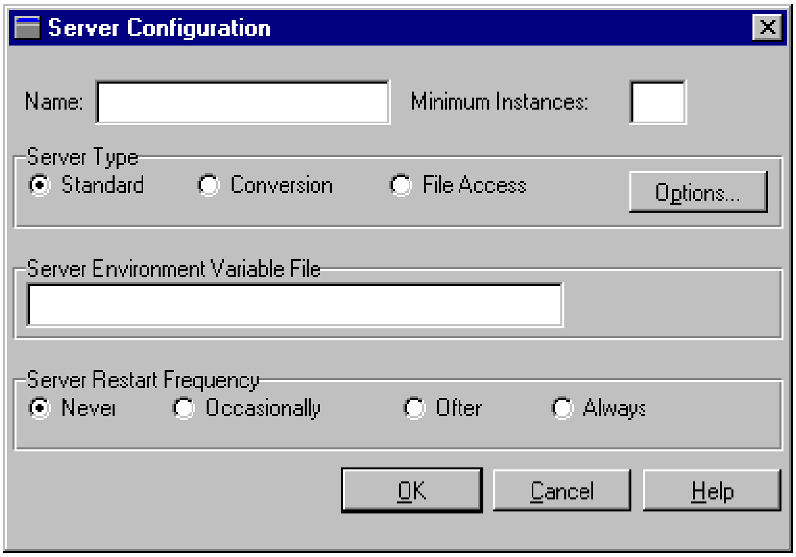
Multiplex
: The maximum number of workstation clients that each workstation handler can support at one time. The workstation listener ensures that new handlers are started as necessary to handle new workstation clients. This property's value must be between 1 and 4096, inclusive. The default value is 10.

Max Init Time
: The amount of time in seconds that is allowed for client initialization to complete before it is timed out by the workstation listener. This property's value can be between 1 and 32,767, inclusive. The default value is 60.

Setting Server Properties

All services in a Panther application are processed by a server. Servers are defined as part of each machine's configuration. You can define standard servers that advertise JIF-defined services, and conversion servers that provide services to client screens converted from a two-tier model. Both types can connect to a one or more database engines. You can also define file access servers for access to remote Panther libraries.

To access server properties, select the server name from the hierarchy list and choose Edit→Properties. The following properties are common to all server types:



Name

The name of this server, up to 30 characters. All server names must be unique within an application and must not be the same as a machine name.

Minimum Instances

The number of instantiations of this server created when it is activated. After the server is activated, you can increase or decrease the number of its instances as needed. The default value is 2.

Server Type

Specifies to use one of the three server executable types that provide services to clients:

- Standard—Advertises JIF-defined services.
- Conversion—Available only to applications that are converted from two-tier applications; the services that are advertised by the conversion server are not defined in the JIF; instead, services are defined in the server executable and cannot be modified. These services include standard database transactions such as `VIEW` and `UPDATE`. These services are only available to converted client screens and can be processed only by converted service components.

- File Access—Offers shared access to libraries and repositories on any machine that is included in the configuration.

Note: The machine environment file of a file access server's host must set `PATH` to `${SMBASE}/util` (refer to page 2-5).

Server Environment Variable File

Specifies to execute this server with the environment in the named file. This property's default is set according to the server type that you choose: `proserv.env` for a standard server or `progserv.env` for a conversion server. You can override the default by entering the pathname directly or by using the Browse push button to select the desired file. By default, file access servers use the settings in `machine.env`; however, you can add additional settings in `devserv.env`.

This environment supplements the one already established when the server is activated. For example, you can set a different server library for each server by setting `SMFLIBS` in its environment file. You cannot use this file to override the machine settings for Panther Install Directory, Application Directory, and Local JetNet Configuration File.

For more information about the contents and format of a server environment file, refer to page 2-6.

Server Restart Frequency

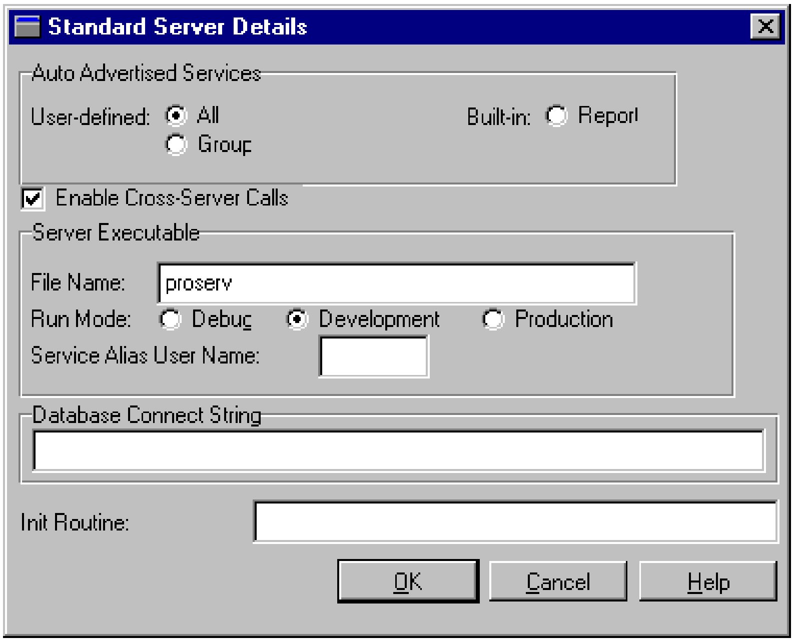
Specifies whether the server automatically restarts after it unexpectedly terminates and how soon:

Server restart option	Time elapsed until restart
Never	
Sometimes	Twice within four hours.
Often	Twice every 10 minutes.
Always	Removes all limitations. The server can be restarted an unlimited number of times.

Server Details

You can set one or more properties that define a server's functionality, depending on its type—standard, conversion, or file access. For example, a standard server specifies which JIF-defined services to advertise, whether it is enabled for debugging, and its database connection.

You access server detail properties by choosing Options from the Server Configuration dialog. This invokes a dialog that is appropriate to the server's type. The following sections describe properties for each dialog:


Standard Server Details

Auto Advertised Services

User-defined: [✓ All] Group

Built-in: Report

[✓] Enable Cross-Server Calls

Server Executable

File Name: proserv

Run Mode: Debug [✓ Development] Production

Service Alias User Name:

Database Connect String

Init Routine:

Standard Server

Auto Advertised Services

There are two categories of auto-advertised services, User-defined and Built-in.

Under User-defined, select one of these options to determine which services this server advertises:

- All—The server broadcasts all JIF-defined services
- Group—The server only broadcasts services that are in the specified group. If you choose Group, supply the name of the group as it is defined in the JIF.

If you leave both options unselected, you must define services for this group through the server's initialization routine (refer to page 3-26).

Under Built-in, select Report to enable this server to process reports. A server that has this check box set advertises the default report service. In addition, a file access server on the same machine will need to be specified. For more information about this service, refer to page 9-23 in *Reports*.

Enable Cross-Server Calls

Specifies whether to allow this server to request services from another server. If this property's check box is set, the server is allocated its own reply queue. Because setting this property increases the number of message queues required by the application, you should do so only if necessary. By default, this property is set.

Server Executable

The name of the executable to run when this server is activated. `proserv` is the default value. You can also specify a standard server that has the Panther debugger linked in, by default `prodserv`. You can enter the file name directly or use the file browser. If the file name omits a path, JetNet looks for the executable in the application directory (specified in the machine's Application Directory property), or in `/bin`.

After you set the server executable's file name, specify whether it is configured for development or production environments by setting one of these options:

- Debug—Identical to Development except that the server runs in debug mode. The debugger starts after default event handlers are established and before the database connection and the server initialization routine.

The server runs in debug mode only if the executable has the debugger linked in.

Note: If you choose Debug for a server, also edit its server environment file so that it sets `DISPLAY` appropriately and `LD_LIBRARY_PATH` to Motif shared libraries. You might also need to reset the application's Default Blocking Timeout property.

- Development—Establishes the default event handlers for a development environment (refer to Table 3-1).
- Production—Establishes the default event handlers for a production environment (refer to Table 3-1). Choose this option for a server if the application is ready for deployment. If the specified server executable has the debugger linked in (`prodserv`), choose Production only in order to simulate a production environment; deployed applications should always use a standard server that does not have the debugger linked in (`proserv`).

Table 3-1 shows which default handlers are established for development and production servers. For detailed information about these handlers, refer to Chapter 6, "JetNet/Oracle Tuxedo Event Processing."

Table 3-1 Default handlers for development and production servers

Event	Development handler	Production handler
advertise	`sm_tp_advertise_log`	`sm_tp_advertise_cond_winopen`
exception	`sm_tp_exception_print_all`	`sm_tp_exception_promote_error`
jif_changed	`sm_tp_jif_changed_read`	Same
message	`sm_tp_message_print_string`	Same
pre_request	`sm_tp_pre_request_ignore`	Same
post_request	`sm_tp_post_request_ignore`	Same
pre_service	`sm_tp_pre_service_winopen`	`sm_tp_pre_service_winopen_or_select`
post_service	`sm_tp_post_service_winclose`	`sm_tp_post_service_winclose_or_deselect`

Table 3-1 Default handlers for development and production servers *(Continued)*

Event	Development handler	Production handler
request_received	`sm_tp_request_received_jif_check`	Same
unadvertise	`sm_tp_unadvertise_log`	`sm_tp_unadvertise_cond_winclose`
unload	`sm_tp_unload_call_origin`	Same
server_exit	`sm_tp_server_exit_log_down`	Same
dbms onentry	`sm_tp_dbms_cmd_log`	None
dbms onerror	`sm_tp_dbms_error_print_all`	`sm_tp_dbms_error_print_all`

Service Alias User Name

Specifies to advertise services under their aliases. The value can be up to 8 characters and is prepended to JIF alias entries to construct the advertised service names in this format:

`userName+JifAliasEntry`

For example, you might define a server that advertises services from the banking service group and sets `lisa` as the service alias user name. Given the following JIF entries for this group:

Service name	Alias entry
`deposit`	`1000`
`withdrawal`	`1001`
`transfer`	`1002`
`balance`	`1003`

The server advertises these service aliases:

```
lisa1001
```

```
lisa1002
```

```
lisa1003
```

`lisa1004`

Servers that advertise aliases are typically set up for the exclusive use of developers who want to modify existing services without affecting the running application. For more information, refer to page 5-8.

Database Connect String
: Establishes a connection to the desired database engine through a `DBMS DECLARE CONNECTION` command. The command is executed after default handlers are established and before the initialization routine executes.

One database connection string is allowed per server. If you want to enable multiple database engine connections, you can do so through the server's initialization routine, specified through its Init Routine property.

Init Routine
: Name of a JPL or installed C function to execute when this server is initialized. For example, you might want to call a JPL procedure that uses the `advertise` command to set the services advertised by this server. Enter the function name and optionally any arguments that it requires, supplied as constant values. The routine is called after default event handlers are established and the database connection is made.

Conversion Server

Conversion servers have three properties:

Server Executable
: The name of the executable to run when this server is activated. `progserv` is the default name for a conversion server. You can enter the file name directly or use the file browser. If the file name omits a path, JetNet looks for the executable in the application directory (specified in the machine's Application Directory property), or in `/bin`.

Database Connect String
: Establishes a connection to the desired database engine through a `DBMS DECLARE CONNECTION` command. The command is executed after default handlers are established and before the initialization routine executes. One database connection is allowed per server.

Because a server connects to a single database, make sure that none of the services that it advertises depends on a different database connection.

Cache Service Components

Specifies whether to cache service components used by the conversion server after they are called. If this check box is unset, service components are opened each time the conversion server processes a service call, and closed when the service call returns. Set this property's check box if the services are frequently called.

File Access Server

File access servers have a single property, File Access Server ID. This property identifies the server by name. If left blank, the property is set to the name of the host on which the server resides.

A Panther application can get a file access server's ID at runtime through the application property `devserv_id`. Report services use this property to return the path of report metafile output to the invoking client (refer to page 9-23 in the *Reports*).

Reasons for setting this property include partitioning requests to your `devserv` processes by creating multiple `devserv` groups

- Controlling the performance of your application by creating multiple `devserv` groups in order to partition requests to the `devserv` processes. For example, in development you might have two `devserv` groups on a machine—one for providing remote library access and another for providing remote report file access.
- Providing machine transparency. For example, if everyone is using a specific host name in their environment variables, such as `SMFLIBS`, and the `devserv` process is moved to a new machine, setting the File Access Server ID to the old host name allows everyone to continue to use their old settings.

CHAPTER

4 Managing the Enterprise

This chapter describes how to use the JetNet manager and other utilities to boot, monitor and manage a running Panther application. It also outlines methods for redistributing network load in a multi-machine application, and ways to monitor and act on errors.

Monitoring an Enterprise

When you select an application, JetNet manager's opening dialog shows the hierarchy of all components. These include objects that are defined in the configuration—the application itself, machines and servers; they are described in Chapter 3, “Configuring the Enterprise.” In an active application, you can also monitor the following components:

- Server instances—An active server has at least one active instance. When a server is activated, it initially has as many instances as its Minimum Instances property specifies. Thereafter, you can increase and decrease the number of server instances as needed. Instances are shown below their server and use the name of the server executable.
- Services—Standard servers that are active list their services under each instance. Depending on its definition, a server advertises either all JIF-defined

services or those that belong to a service group definition. Services are listed for informational purposes only; you cannot perform any action on them.

- Clients—All clients that are attached to the application are listed, including the JetNet manager. Clients are only visible when you connect the JetNet manager to the application from the master machine (refer to page 4-4).
- Workstation listeners— An active machine that is configured to allow workstation client connections always has a workstation listener process. This process intercepts connection requests and, if possible, finds a handler for the connection. For more information about setting workstation listener properties, refer to page 3-17.

Initially, only the top-level component—for the application itself—is shown. Double-click on a component to toggle subordinate components in and out of view. A component with subordinates is prefixed with a - or + symbol to indicate whether they are visible or not.

To the right of the hierarchy list, the Details list displays information about the selected component, such as its name, type, and state (active or inactive). The dialog also contains a Status box that displays informational messages.

Activating and Deactivating Components

The JetNet manager lets you activate and deactivate application components that are visible in the hierarchy list through the Edit menu options Activate and Deactivate. You can deactivate any component except a client and the master machine.

Note: To deactivate clients, use forcible deactivation (Edit→Forcibly Deactivate). Refer to page 4-5 for details. To deactivate the master machine, deactivate the application from the master machine.

The scope of objects accessible to activation and deactivation depends on where you are running the JetNet manager. For example, you can activate and deactivate an application only from the master machine. The following figure shows what activation and deactivation options are available from each type of site:

Activate/Deactivate:	From:
Application	master machine
Non-master machine*	master/non-master machine, client workstation
Server	master/non-master machine, client workstation
* You cannot deactivate the machine on which the JetNet manager is running	

Deactivating a component can also cause subordinate components to deactivate. For example, deactivation of a server deactivates all instances of that server.

The following sections discuss activation and deactivation according to component types and the issues that are associated with each type.

Enterprise Application

You can activate an application with the JetNet manager or with the command-line utility `rbboot`. In either case, activation can take place only on the master ma chine. Before activating an application, verify the following conditions:

- All machines have the executables for which their servers are configured.
- Each machine has `SMRBCONFIG` set to the same value as its Local JetNet Configuration File property (refer to page 3-14).
- In a multi-machine application, the listener process is running on all machines. Start the listener process with `rblisten`.

To deactivate an application from the JetNet manager, you must be running the JetNet manager from the master machine. You can also shut down an application with the command line utility `rbshutdown`.

If an application has client connections, attempts to deactivate can yield partial shutdown, where those machines that have client connections remain alive. In this case, the application also remains active. To shut down an application that has client connections, choose File⟶Forcibly Deactivate (refer to page 4-5).

Machine

You can activate and deactivate any machine except the one on which the JetNet manager is running and the master machine. In order to activate a non-master machine, the listener process must already be running on it (refer to page A-7).

If you try to deactivate the machine on which the JetNet manager is running, all subordinate components including servers are deactivated; however, the machine itself remains active.

Note: To activate the master machine, you must activate the entire application from the master machine, either through the JetNet manager or `rbboot`. Similarly, deactivate the master machine by deactivating the application from the master machine, either through the JetNet manager or `rbshutdown`.

Servers

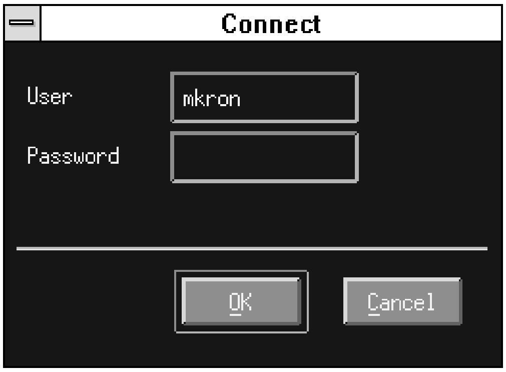
You can activate any server. Before you activate a server, make sure that the appropriate server executable is installed in the same location as the server definition specifies (refer to page 3-23).

When you activate a server, it initially has as many server instances as its Minimum Instances property specifies. Thereafter, you can increase and decrease the number of server instances as needed. For more information, refer to page 4-6.

If you deactivate a server, JetNet automatically removes all instances.

Connecting and Disconnecting

If the current application is active, you can connect the JetNet Manager to it as a client by choosing File→Connect to Application. When you choose this option, the Connect dialog displays:



User mkron
Password

Log in with your user name. If a password is set for the desired application (refer to page 3-12), you must enter it, too. After the JetNet manager is connected, it and all other connections are listed by user name in the hierarchy list. The JetNet manager shows client connections to an application only when it is itself connected to the application.

When you disconnect from an application (File→Disconnect from Application), the JetNet manager terminates its client connection and removes from view all other client connections.

Forcibly Deactivating Components

You can force deactivation of application components by choosing Edit→Forcibly Deactivate. When you choose this option, a confirmation dialog asks whether to proceed. Choose this option instead of Deactivate for one of the following reasons:

- To ensure shutdown on the machine and application level. For example, normal deactivation fails for any machine that has clients connected to it. Forcible deactivation disconnects clients and allows machine shutdown.
- Parts of an application are in an undefined state and are not responsive to deactivation attempts.

- To disconnect a client. Normal deactivation is invalid for client connections.

You can forcibly deactivate four objects: application, machines, server instances, and clients. Forcible deactivation is invalid for servers.

Forcibly deactivating a component causes JetNet to forcibly deactivate all subordinate components. For example, deactivation of a machine also deactivates its server instances and disconnects its clients.

Adding and Deleting Components

You can add servers and machines to an active application exactly as you do with an inactive application. All additions are immediately written to the application's configuration file. You can also delete servers and machines; however, they must first be inactive. If you delete a machine that has servers configured, the JetNet manager deletes the machine and its subordinate components on confirmation. If you delete an application object, the entire application is removed. You cannot delete an application's master machine.

Adding and Removing Server Instances

When a server is activated, it initially has as many instances as its Minimum Instances property specifies (refer to page 3-20). If a server is experiencing a high volume of service requests, you can facilitate throughput by increasing the number of instances. Conversely, you can remove instances from an under-utilized server.

How to Add a Server Instance

1. Select the server from the hierarchy list.
2. Choose Edit→Add Instance.

How to Remove a Server Instance

1. Select the instance from the hierarchy list.
2. Choose from the Edit→Deactivate

If the instance is processing a service request and the `tp_block` property is set to `PV_YES`, the instance is not deleted until the service completes or times out. If you attempt to deactivate instances of a workstation listener (WSL) server that is responsible for maintaining workstation client connections, the JetNet manager issues a warning message and asks whether to disconnect these clients.

Note: If you deactivate all instances of a server, JetNet deactivates the server.

Changing Machine Roles

An application that runs on multiple machines—especially one that must run continuously—should always have a machine designated as backup master. In a running application, the backup master is always prepared to take over the role of master machine in case the master machine unexpectedly terminates.

You might also want to reassign the master or backup master machines in a running application—temporarily, in order to bring down the master machine for periodic maintenance; or permanently because of changes in the network.

Recovering From Master Machine Failure

If the master machine failure in a three-tier application fails, or its DBBL and BBL processes fail, the application continues to run but the DBBL is not recreated on the backup master machine. In order to recover from abrupt termination on the master machine, stop all application servers and BBLs when possible and reboot the application to restart the DBBL.

Reassigning Master and Backup Machines

You can reassign the master and backup machines by running the JetNet manager from any active machine and invoking the Machine Roles dialog (refer to page 3-8). When you reassign the master, Panther performs the necessary migration of control without disrupting operations.

Disabling and Reenabling Workstation Connections

In an active application, you can stop a machine's workstation listener and reconfigure the machine to disallow workstation connections as follows:

1. Select the machine's workstation listener server from the Application Status dialog and deactivate it.
2. In the machine's Network Settings dialog, unset the Workstation Listener toggle button.

To restart an active machine's workstation listener, follow the same procedure in reverse: set the machine's Workstation Listener toggle button, then activate its workstation listener server.

Handling Load

You can efficiently distribute application processes and optimize the use of resources in several ways:

- Define service groups and assign them to servers so that service requests can be routed to the servers that can handle them most efficiently. For example, you might have a number of services that can be processed quickly and are frequently requested; several other services are lengthier and are requested less often. In this case, it makes sense to specify the two types of services in separate groups; one server can advertise the group of quick services, with enough instances to handle peak load; another server can advertise the group of lengthy services, with only enough instances to handle occasional requests.
- Use the Load Balancing (page 3-11) and Network Load (page 3-16) properties to route service requests to the server machines that are most available to handle them.

If service requests frequently time out or take a long time to complete, you can increase the setting for the application property Default Blocking Timeout (page 3-12). You might also add machines to the configuration and redistribute servers accordingly.

You should also modify IPC settings to handle the application's resource requirements. IPC requirements can vary according to these property settings:

- Max Accessors—Set on the machine level, this property specifies the maximum number of processes, clients and servers, that can attach to the application on a given machine. This property directly increases semaphore and shared memory costs.
- Enable Cross-Service Calls—Set on individual servers, this property specifies whether to allow this server to request services from another server. Setting this property increases the number of message queues required by the application.

The number of semaphores required on each machine is equal to the value of Max Accessors. To calculate the number of message queues, count one queue for each BBL and the DBBL, one for each client and server, and two for each bridge process. If a server's Enable Service Calls property (in the JetNet configuration file) is set to Yes, also count one queue per server instance.

Queue-related kernel parameters (refer to page 2-12) should be tuned to manage the flow of buffer traffic between clients and servers:

- The maximum total size of a queue in bytes (`MSGMNB`) must be large enough to handle the largest message that the application allows. The queue should usually be about 75 to 80 percent full.
- The maximum size for a message (`MSGMAX`) must be set to handle the largest buffer that the application might send.

- The maximum queue length (`MSGTQL`) must be set high enough to handle application operations.

To gauge a queue's average fullness or length, you must run the application. Finding optimal settings for tunable parameters is typically a trial-and-error process.

If you have a large system, you might want to analyze the effect of parameter settings on the size of the operating system kernel. If it is unacceptable, consider reducing the number of application processes or distributing the application to more machines in order to reduce the Max Accessors setting on each one.

Status and Error Messages

Status and error message can be posted in one of three places:

- The JetNet manager's status window
- Machine log file
- `stderr`, which gets server error output

Machine log file names have the format `ULOG`*mmddyy*. Log files on PC workstation clients have the format `UL`*mmddyy*`.log`. On UNIX, machine log files must be located on the path specified by the machine's Application Directory property. On Windows, the `ULOGDIR` setting in the machine's Environment panel specifies their location.

CHAPTER

5 Defining Services in JetNet and Oracle Tuxedo Applications

In JetNet and Oracle Tuxedo applications, services are subroutines that do the work required for an application to access a resource manager, usually a database. They are invoked by service requests made by clients or other services.

To expedite responses to service requests, the application can run multiple instantiations of a server. Service requests are routed to servers in the way that provides the fastest response.

This chapter covers:

- Service components, including the service definition in the JIF.
- Service aliases.
- Creating services with the screen wizard.
- Writing and calling service procedures.
- Grouping services.

Services

A service can consist of three parts:

- A routine that implements the service.
- A service component (optional) that provides a physical means of sending, receiving, and processing data.
- A service definition in the JIF.

To create a service, you can:

- Use the screen wizard or the screen editor to create a service component—the graphical or visual representation of a service.
- Write the service routine that implements the service.

The latter method is convenient if you are creating services that do not access a database. You might also choose to create the service components manually if you converted an application using the `clnt2svr` utility and special handling is required, for example, if a client screen implements partial commands via the transaction manager. For more information on `clnt2svr`, refer to page A-2.

Service Routine

Service routines are responsible for optionally receiving data from the client, performing some task, and optionally returning data to the client. A service routine can perform any task, such as building a query for accessing a database.

Service routines are built for you when you use the screen wizard to create the server portion of your application—that is, the service component. These services perform the database transactions required by your application. For more information on creating services with the screen wizard, refer to page 5-19.

You can also write service routines. These can be written in the same way you write any other Panther application code. JPL is most convenient, but you can code a service as a C or Java function if that suits your application needs.

The JPL service code can reside in a library that is accessible to the server, or it can be implemented as screen-level JPL (in the JPL Procedures property) on a service component (described in the Service Component section). The routine cannot be attached at widget-level since the routine must be recognized outside the scope of the widget. At runtime, the service code procedure is sought first at screen-level (on the service component), then in public modules, and finally in open libraries on the server.

For information on how to write a service routine, refer to page 5-9.

Service Component

A *service component* is a graphical service. It's a Panther screen that should look, for the most part, like the client screen it is servicing. However, service components reside on the server (in a server library such as `server.lib`) and so are not visible to the user at runtime. The service component should contain the same widgets as the client screen so that it can handle the data that flows between the client screen and the service.

In brief, the client makes a service request and passes information to the service. The service accesses its corresponding service component which receives the information and performs any runtime processing, including accessing the database via the transaction manager or the database interface. In this way, the service component can carry out a variety of database-related services, such as finding, inserting, deleting and updating records.

The easiest way to create service components is by using the screen wizard. You can create them at the same time you create the client screen. When service components are created in this way, the service routines are automatically provided, so you don't have to write the routines at all. In addition, service components can hold the routines for more than one service, and multiple services can use the same service routine or the same service component.

Service components can be created and edited in the screen editor, just like any Panther screen. Refer to page 12-1 in the *Application Development Guide* for further information on creating service components.

JIF Service Definition

When a service call is made, or a server is told to advertise services, the application server obtains information about the services by consulting the JIF, which at runtime resides on the application server in `common.lib`.

A JIF service definition requires the following information:

- Routine name—The name of the JPL procedure or C/Java function that implements the service.
- Transport method—The data type of the message that transports data to and from the service, specified individually for incoming and outgoing data. You can choose to use either `JAMFLEX` or, if no data is required, `none`. With the Oracle Tuxedo middleware adapter, additional buffer types, `FML`, `FML32`, and `STRING`, are available.

You can create and modify the JIF in the JIF editor. For information on using the JIF editor, refer to Chapter 25, “JIF Editor,” in *Using the Editors*.

Optional Service Attributes

You can optionally specify several other service attributes in the JIF:

- The service component associated with the service.
- Synchronous or asynchronous calling modes.
- Normal blocking timeouts.
- An exception handler.
- An unload handler.
- Reply to the calling agent.
- Execution outside the active XA-transaction (only for the Oracle Tuxedo middleware adapter).
- Priority.
- Caching of service component.

- Transaction manager operations: Select, Insert, Update, Delete, or Link Validation.

For further information on these service features, refer to page 25-5 in the *Using the Editors*.

Creating Graphical Services

Services can be represented as a screen, called a service component, that resides on the server. You can create client screens and associated service components with the screen wizard or you can build service components by dragging database objects from a repository to screens using the screen editor.

Creating Services with the Screen Wizard

Using the screen wizard makes service creation practically effortless. To facilitate client and server development, the wizard lets you build the client screen and service component at the same time.

The JPL code that implements the transactions via service calls and the transaction manager are provided for you, and made public via the service component's JPL Procedures property. Unless specialized tasks are necessary for your application, you do not need to write any service code at all.

Database transaction services are named by the screen wizard and implemented via the Service properties on the master table view of the client screen: Delete Service, Insert Service, Select Service, and Update Service.

While client screens and service components in the screen wizard are created as pairs, that is, one service component for each client screen, you can also develop service components that can service multiple client screens. This can be done by creating a service component for each table view that client screens might use. In this way, a service component can service multiple client screens, and in turn, client screens might make service requests to more than one service component. If a client screen must access multiple service components, Service properties must be defined for all table views on the screen. In other words, the master table view on the client screen will call services specific to the master section of the screen, and the detail and subdetail sections will call services specific to their own table view. Therefore, the Service properties for each table view on the screen must be identified.

In general, it is best to create both client screen and corresponding service component at the same time. Save client screens in the client library, and save service components in the server library. Even if you don't plan on using the resulting client screen for your application interface, you can use it to test its associated service component; consider saving such client screens in a test library.

Once services are created in the screen wizard, you must define them in the JIF. When you invoke the JIF editor:

- Make sure the service names match those in the Service properties on the table view widgets on the client screens.
- Identify the JPL procedure or C/Java function containing the service routine code.
- Identify the associated service component, and set any other service attributes.
- Save the JIF in the common library on the application server.

Once services are added to the JIF, all servers that are currently running are made aware of changes to the JIF. The new services are immediately available for the application to advertise. Refer to page 5-14 for more information.

For more information on using the screen wizard, refer to Chapter 5, “Screen Wizard,” in *Using the Editors*.

Building Services with the Screen Editor

To create a service component using the screen editor, you must include and identify all the components necessary to implement the service. In addition to building the screen, you must code the service routines and set the appropriate property values on the client screens that will use the service component.

For screens that use the transaction manager, both client screen and service component must contain the same database information—the same table views, the same columns, and so on. To create a service component from scratch with the screen editor:

1. Copy the database-related widgets from the repository or from the client screens that will use this service component.
2. For the master table view on client screens, set the appropriate Service properties for handling the database transaction (if the screen is using the transaction manager).
3. Write the service code that will be used by the service component and make it available to the service component, by either:
   - Including the JPL procedure directly in the service component's (screen-level) JPL Procedures property, if the code is pertinent only to a single container.
   - Storing the procedure in a library module and make the module public via the service component's entry function.
4. Define the services in the JIF on the application server.
5. Save the service component in the server library.

Modifying Service Components

For the most part, service components must have the same contents and property values as the client screens that use them. Therefore, changes you make on a client screen must also be made to its corresponding service component.

To modify a service component, open it in the screen editor and make the appropriate edits. Some typical modifications might include adding a transaction manager hook function as screen-level JPL or defining the Use In Where property of a widget.

Initiating a Service

To initiate a service, a client screen must have a way to make a service call. A service call can be implemented as JPL code (using the `service_call` command) attached to a widget on the client screen, or by naming the service in the Service properties of a client screen's master table view. Services identified as Service properties are handled by the JetNet transaction model `jetrb1` on screens that use the transaction manager.

For more information on the JetNet transaction model, refer to page 35-12.

Using Service Aliases to Test Services

Once the service has been added to the JIF and the service component and service code are available on the application server, you can test its behavior. To allow several developers to work on the same application, each developer has the ability to test their own version of a service using service aliases.

With service aliases, a developer's name can be appended to the service name. The original service name is available for all other users of the application; the developer is able to change functionality without breaking the application for other users.

To use service aliases:

- In the editor under Options→Service Alias, the Service Alias option must be configured. A user name must be entered; it allows up to an eight character alphanumeric string. The Use Service Aliases check box must be selected.
- In the application server setup, a unique string must be specified for the application server in the Service Alias User Name field.

The JIF automatically assigns a unique name to each service when the service is added to the JIF. This unique service name, in combination with the developer's name, provide the service alias.

To discontinue service aliases:

- In the editor under Options→Service Alias, deselect the Use Service Aliases check box.

Writing Service Routines

A service routine is responsible for:

- Receiving requests from clients, possibly including input data.
- Performing a service, for example, using the data submitted by the client to find a record in the database.
- Returning result status and possibly data to the client agent (service calls can originate from another server as well as from a client).

Table 5-1 lists the basic functions that a service routine should perform and includes the JPL, library function, and SQL that can be used to implement them.

Table 5-1 Service functions and corresponding Panther commands and SQL

Function	Code	Source
Receive arguments from client agent	`receive`	JPL command
	`sm_receive`	Panther library function
Fetch data from database	call `sm_tm_command` ("`VIEW`") or ("`SELECT`")	JPL command to Panther library function for the transaction manager

Table 5-1 Service functions and corresponding Panther commands and SQL *(Continued)*

Function	Code	Source
	`dbms QUERY SELECT ...`	JPL command to database interface
Force screen validation	`sm_s_val`	Panther library function
Update the database	call `sm_tm_command ("SAVE")`	JPL command to Panther library function for the transaction manager

	`dbms RUN UPDATE ...`
	`dbms RUN DELETE ...`
		JPL command to database inter face

Return results to client agent	`service_return`	JPL command

The following procedure is an example of a JPL service routine that receives the account id and amount from the client and uses that information, by way of the transaction manager, to perform a bank account withdrawal.

```
proc withdraw()
vars message

// service WITHDRAWAL
receive arguments ({account_id, amount})
call sm_tm_command("SELECT")
if (account_id->num_occurrences <= 0)
{
    service_return failure ({message = "Invalid account."})
}
// check if the amount to be withdrawn is more
// than the withdrawal limit
if (amount > max_withdrawal)
{
    message = \
        "Withdrawal limit is " ## max_withdrawal
    service_return failure ({message})
}

account_balance = account_balance - amount
if (account_balance < 0)
{
    message = \
        "Account overdraft attempt on account " ## account_id
    log message
    service_return failure ({message})
}

call sm_tm_command("SAVE")
service_return ({message = @NULL, balance = account_balance})
}
```

The following service routine uses data it receives from the client to apply business logic. Storing business logic on the application server can facilitate application maintenance since you only need to update the service routine in order to effect new business practices. In this example, the service routine uses the customer's id number and the gross amount of an order to determine at what amount a percentage of discount should be applied.

```
// Use gross_amt to determine level of discount for
// a customer. Apply the discount and return net_cost

proc discount()
receive arguments ({customer_id, gross_amt})
call sm_tm_command("VIEW")
if (customer_id->num_occurrences <= 0)
{
    service_return failure ({message = "Invalid customer id"})
}
if (gross_cost > 1000)
{
    net_cost = gross_cost * (1 - discount)
}
else if (gross_amt > 100)
{
    net_cost = gross_amt * (1 - discount / 2)
}
else
    net_cost = gross_amt

service_return success ({net_cost})
```

Storing and Invoking JPL Service Code

You can write JPL service routines directly in the screen editor. The service and your application's requirements, for development and production, will determine where it makes the most sense to store and invoke service code:

- Directly with the associated service component in its screen-level JPL (in the JPL Procedures property).

- Via the JIF specification for the service when there is no service component associated with the service.
- Made public on server initialization or by the service component, if there is one.

Service Code and Service Components

A service routine that resides directly on the service component is immediately available when the client makes the service request. Service components are opened or made available either when their associated service is advertised by the server, or when the service is requested.

Panther's built-in development and production `pre_service` handlers make the necessary service component available, and therefore, the routine is made available.

The built-in `post_service` handlers close or deselect the service component after the service completes.

Changes you make to the service code are immediately available to your application without having to restart servers or recompile.

JIF-Invoked Services

If you have service routines that are not associated with a service component, there are two ways you can make these routines available:

- For the development phase, you can let the JIF invoke the routine.
- For deployment, you can public the routines on server initialization.

For development purposes, store a JPL service routine in a module—one service routine per module—and do not include the `proc` statement. When the service is requested, the JIF is consulted to determine the name of the service's routine—in this case, the procedure and module name would be the same. At runtime, if a procedure by the given name is not found, the service seeks a module having the specified name, and executes its unnamed procedure (refer to page 19-2 in the *Application Development Guide* for information on the unnamed procedure in a JPL module).

In this way, you can easily access, update, and retest code without affecting the server.

Once you are ready to deploy your application, edit the JPL module and add a `proc` statement to identify the procedure (same as the module name). You can then `public` these JPL service routines when the server is initialized.

Public Services

If the service routine does not reside directly with a service component, the JPL procedures can be made public either from the service component's entry function, or on server initialization.

When JPL is made public, it is available to the entire application until the server is brought down, or until the module is unloaded. For development purposes, making modules public on server initialization is the least flexible, because the server offering the services must be brought down and reinitialized in order to retest service code that has been modified.

During the development cycle, consider unloading public modules from the service component's exit function. In this way you can update the code, if necessary, and retest it without affecting the server that advertises the service.

Service Groups

Once you determine what services are required by your application, you can create *service groups*. You can group services for your application if you have just one application server or multiple servers. Service groups are useful because:

- Your servers can easily advertise sets of services, rather than all services defined in the JIF.
- You can logically group services based on their duration or response time.
- When a service is added to or updated in a group in the JIF, the new or changed service is immediately available to the application.

You define service groups in the JIF. Refer to page 25-12 in the *Using the Editors* for information on using the JIF editor.

Criteria for Grouping Services

To determine what services should be grouped, consider establishing a logical coherence within the application's tasks, such as:

- Which services should be advertised when the server is first brought up? Additional services can be made available by the server later.
- Are some services needed at specific times? Some services run continuously but others are shut down at a specific time. For example, in a banking application, services used by ATM clients that a required to run 24 hours might be grouped in one service group, while services used by bank teller clients can be grouped in another service group whose server is shut down during non-business hours.
- Is there is a substantial difference in the response time of the services? Services that respond quickly—for example, in less than five-seconds—can be in a service group on one server, and another group of services that takes longer to complete might reside in a server.

In general, it is easier and more efficient for servers to advertise groups of services. For instance, if a server's initialization routine specifies that all services should be advertised, *all* is interpreted as all services defined in the JIF (refer to page 3-23 for JetNet or to page 8-17 for Oracle Tuxedo).

If your application has multiple (unique) servers, each server should have at least one service group that contains all of the services that the server handles. Additional servers might be necessary if some services use an XA-compliant database interface, while others do not, or if your application accesses more than one database.

Adding Services to Existing Service Groups

During development, you can easily add or update services associated with a service group. When a service is added or updated in the JIF and it is saved to the application's common library, all running servers are notified to reread the JIF. Service groups that are advertised at server initialization are readvertised, and the new or updated service is immediately available to your application.

For details on how to define service groups in the JIF, refer to page 25-12 in the *Using the Editors*. For details on advertising a service group when starting your servers, refer to page 3-23.

Service Messages and Data Types

In JetNet and Oracle Tuxedo applications, clients and servers exchange data through messages, and the data type of those messages must be specified. For example, a service call can have 0 to 2 messages, depending on how the service is defined, for request data supplied to the server, and reply data expected by the client when the service returns.

Messages can be composed of simple strings; or they can contain combinations of numeric and string data.

In JetNet applications, messages are in `JAMFLEX` data buffers.

In Oracle Tuxedo applications, messages can be data buffers using the `JAMFLEX`, `FML`, or `FML32` data types, or messages can be strings using the `STRING` data type.

For example, when a client calls a bank deposit service, it supplies a message to the service that is composed of several data fields, such as the bank account number and deposit amount. The client also expects the service to return with a reply message whose data includes the bank balance. So, the service call specifies two messages, one for input data and another for reply data, where each message can itself contain multiple data fields:

```
service_call "DEPOSIT"( \
   {account_id, amt}, \
   {errMsg, acctBal} )
```

In this example, the service request message consists of the account number and deposit amount; on return, the service can supply the client with an error message in case the service fails, and the updated account balance. Each message contains two data fields.

The previous example contains two `JAMFLEX` messages. A service call can also specify messages of different types. For example the following call to service `OPEN_ACCT` specifies a `STRING` message type for input data and a `JAMFLEX` buffer for reply data:

```
service_call "OPEN_ACCT"("Fred Jones", {acct_num, start_bal})
```

The following sections discuss service message types.

Buffer Data Types

Service messages can be defined as buffers that can contain multiple components of data of different types. In JetNet applications, messages are in `JAMFLEX` data buffers, which can contain string, integer, and floating point data. In Oracle Tuxedo applications, messages can be data buffers using the `JAMFLEX`, `FML`, or `FML32` data types.

All buffer types are represented in JPL as a comma-delimited list of fields in this format:

```
{[field [, field] ] }
```

The buffer can contain 0 or more fields, where each buffer field has this format:

```
fieldname [= prolfx-expr]
```

`fieldname` is the name of a field in the buffer, and *`prolfx-expr`* can be a Panther variable or constant. If you omit *`prolfx-expr`*, Panther assumes that *`fieldname`* has the same name as a Panther variable or widget and maps its data accordingly.

Note: If the buffer type is `FML` or `FML32`, *fieldname* must correspond to the name of a field defined in the FML file. Refer to page 5-17 for more information.

For example, a message can be specified as follows if the Panther variables and the corresponding buffer field names have the same names:

```
{EMP_ID, EMP_NAME, EMP_ADDR}
```

In the next example, the names of the Panther variables and the corresponding buffer field names are different, so explicit mapping is required:

```
{EMP_ID=id, EMP_NAME=name, EMP_ADDR=addr}
```

Default Mapping

Reply buffer data can be automatically mapped to Panther variables of the same name through ellipses. For example, this `service_call` command specifies to map all buffer fields to client variables that have the same name:

```
service_call "get_emp_age" ({emp_id},{...})
```

In the next example, the buffer field `emp_age` is mapped to Panther variable `age`. All other buffer fields are mapped to same-named client variables:

```
service_call "get_emp_age" ({emp_id}, {emp_age=age, ...})
```

NULL Arguments

You can specify the data in a message buffer to be `NULL` through the keyword `@tpi_null`. For example, this `service_return` command returns a `NULL` buffer to the service call:

```
service_return failure ({@tpi_null})
```

Arrays

Buffers can reference array occurrences. If a message field specifies the name of an array, Panther references each occurrence in that array. You can also specify ranges of occurrences. In the following example, the first 5 occurrences of `emp_id` are sent to the service:

```
service_call "get_emp_args" ({emp_id[1..5]}, {...})
```

FML and FML32 Buffers

A Oracle Tuxedo application can support `FML` buffers for messages, both `FML` and the enhanced `FML32`. For complete information on `FML` buffers, refer to your Oracle Tuxedo documentation.

You define `FML` fields in the `FML` file as one of the following data types: char, string, short, long, float, and double. Fields are listed by name, field number, and type. A fourth `Flag` field is unused. For example:

```
#Name     Number   Type       Flag
#----     ------   ----       ----
CUSTNUM      1     string      -
CUSTID       2     long        -
CUSTADDR     3     string      -
CUSTCITY     4     string      -
CUSTSTATE    5     string      -
CUSTZIP      6     long        -
```

Type conversion from the data type specified by the user to the `FML` field type is performed as the data allows. In the following example, if `amount` is of type float, and if amount = 3, then 3 is converted to a floating point number:

```
{amount = 3}
```

Converting from JAMFLEX to FML

You can convert `JAMFLEX` buffers to `FML` buffers later in the development cycle:

- Change the message argument data types for services and queues in the JIF.
- Add the corresponding fields in the Oracle Tuxedo `FML` file.
- Edit any `broadcast` or `notify` commands used in the application to specify the appropriate `FML` type.

STRING Data Types

A Oracle Tuxedo application can also support `STRING` message types. A `STRING` message can send or receive only a single string. A `STRING` input message can be any string expression, either a variable or string constant; a `STRING` reply message must be a variable.

For example, the following JPL procedure calls a service that expects `STRING` messages for input and output:.

```
proc lastname
vars last_name
service_call "get_last_name" ("Jim", last_name)
msg emsg "Last name is ", last_name
```

In this example, service `get_last_name` receives the string `Jim` as an input message. When the service completes, the return data is put in the output message and mapped to Panther variable `last_name`.

In the next example, the input message to the same service refers to a Panther variable:

```
service_call "get_last_name" (first_name[i], last_name)
```

In this case, the first name passed to the service is the value in the `i`*th* occurrence in array `first_name`.

Setting Service Message Types

The data types for service messages are set in the JIF as part of service or queue definitions. For each service or queue definition, you can specify the data type for its request and reply messages. You specify a data type for both messages, only one, or for none.

When the request broker processes one of these messages, it checks the JIF for its data type and uses this information in order to pack or unpack the message data. The format of a message must conform with its JIF definition; otherwise, an error occurs.

`service_call` messages can contain zero to two comma-delimited messages. Other commands such as `dequeue` and `service_forward` allow only zero or one message for either input or output.

Several JPL commands send messages that are independent of service definitions:

- `broadcast`
- `notify`
- `post`

These commands set their message's data type. For example, this `broadcast` command sends a message that informs the specified user how much time has elapsed since she logged in and how much time remains until the login lapses:

```
broadcast USER userName TYPE JAMFLEX \
   ({sinceLogin, timeLeft})
```

CHAPTER

6 JetNet/Oracle Tuxedo Event Processing

In JetNet and Oracle Tuxedo applications, events occur during the processing of service requests. These middleware events are one of several categories of events that occur in a Panther application. For information about the different types of application events, refer to Chapter 17, “Understanding Application Events,” in *Application Development Guide*.

When middleware events occur, they are forwarded to *handlers* for processing. Panther provides built-in event handlers for all request broker event types. You can also write and install your own handlers in JPL or C. These handlers can perform all required processing on their own, or they can call the built-in handlers and overlay these with desired enhancements.

Table 6-1 lists all middleware event types that Panther recognizes in JetNet and Oracle Tuxedo applications:

Table 6-1 Middleware events

Event type	Description	Location
`advertise`	A service has been advertised	Server
`exception`	An error or unusual change in the normal flow of program execution	Client or server

Table 6-1 Middleware events *(Continued)*

Event type	Description	Location
`jif_changed`	The JIF has been changed	Client or server
`message`	A client receives an unsolicited message	Client
`post_request`	A service request is completed	Client or server
`post_service`	A service completes execution	Server
`pre_request`	A service request is initiated	Client or server
`pre_service`	A service is about to begin execution	Server
`request_received`	A service request is received by the server	Server
`server_exit`	A server is brought down in an orderly fashion	Server
`unadvertise`	A service has been unadvertised	Server
`unload`	Data is received from an external source that can be written (unloaded) to Panther variables	Client or server

Event Sequence

During the typical life span of a Panther enterprise application, most or all middleware events occur. Some of these, such as `advertise` events, happen independently of other events. For example, in an multi-server enterprise, servers can be activated and deactivated in a running application, thereby generating `advertise` and `unadvertise` events. In the meantime, clients continue to issue service requests, and those servers that are available process them; these actions spawn their own set of service-related events, such as `request_received` and `unload`.

Of all middleware events, only those that are connected to service requests are dependent on each other, and always occur in this sequence:

Action	**Events**
Client initiates request	1. `pre_request`
Server receives request	2. `request_received` 3. `pre_service` 4. `post_service` 5. `unload`
Server returns service to client	6. `post_request`
Service returns to client	7. `unload`

Handler Scope and Installation

All middleware event handlers are installed at one of several scopes, which are hierarchically ordered from most general (default scope) to most specific (request scope). A handler can be installed at one or more scopes, depending on its event type. When a request broker event occurs, Panther looks for its handler at the most specific scope that is valid for the event. If no handler is installed at that scope, Panther continues to search for the event's handler among other valid scopes, each more general than the last, until it finds one.

Event handlers can be installed at four scopes, listed here in ascending order of precedence (general to specific):

- Default—Panther provides built-in handlers which are installed at this scope and cannot be replaced or removed. A default handler is installed for each event type; Panther uses it if no handler for that event is installed at another scope.
- Application—Valid for all request broker events, handlers are installed at application scope through the appropriate runtime properties. An application

property is defined for each request broker event type (refer to page 1-103 in *Quick Reference*). An application scope handler supersedes the corresponding default handler until the application exits or the handler is explicitly uninstalled.

For example, to install an `unload` handler at application scope, set the `hdl_unload` property:

```
@app()->hdl_unload = "user_unload"
```

You can uninstall a handler from application scope by setting the corresponding property to an empty string. For example:

```
@app()->hdl_unload = ""
```

- Transaction—Handlers can be installed via the `xa_begin` command for `exception` and `unload` events that occur within a transaction. If installed, these handlers are used during the transaction, and are deinstalled when the transaction ends.

- Request—Handlers can be installed via the `service_call` command for `exception` and `unload` events that occur within the requested service. If installed, these are used while the service undergoes processing, and are deinstalled when the service returns.

For example, an `unload` handler can be installed at any scope. When an `unload` event occurs, Panther first checks whether an `unload` handler exists for the applicable service. If no request handler is found and the `unload` event occurred within a transaction, it checks whether an `unload` handler is installed for that transaction. If no transaction handler is found, Panther looks for a handler installed at application scope. If no application-scope handler is set, Panther uses the default `unload` handler.

If Panther cannot find the specified handler, a `TP_HANDLER_MISSING` exception of severity `TP_ERROR` occurs.

Writing Event Handlers

You can write a middleware event handler in JPL or C. Handlers written in JPL can be stored in screen or library modules; handlers written in C must be installed as prototyped functions. (For more information, refer to “Prototyped Functions” on page 44-8 in *Application Development Guide*.)

Request broker event handlers that are written in JPL should be accessible to the application either as library modules that are made public (via the `public` command) or as screen modules (via the screen's JPL Procedures property):

- Application—scope handlers should always be available to the entire application; therefore, store these handlers in library modules and make them public at application startup through the base form's unnamed procedure. (For more information, refer to “Unnamed Procedure” on page 17-6, in *Application Development Guide*.
- Transaction—and request-scope handlers can be stored in a client screen or service component that is open when a service request or transaction is initiated and remains open until the processing associated with the request or transaction is complete.

Handlers for a request broker event type must conform to a *contract* which is specific to the event type, and specifies the handler's implementation. A handler contract specifies the number and type of parameters, and how to interpret its return integer value. For information about handler contracts, refer to event type descriptions later in this chapter.

Events Generated within Handlers

An event handler should avoid generating events of its own type as this can create an infinite loop. Because `pre_request`, `post_request` and `unload` events are necessary to make service requests, these events can be generated recursively. Handlers for these events should guard against generating events of the same type.

When a handler is invoked for an event, the following changes to normal processing occur:

- Handling of events of the same type is disabled until the current event has been processed.
- Handling of events that belong in pairs is disabled in pairs.

Built-in Handlers

The Panther distribution provides a number of built-in handlers that are internally installed. All handlers at default scope are built-in handlers. A number of the built-in handlers are not used as defaults; these are available for installation at other scopes. For example, two built-in handlers are available for `jif_changed` events: the default handler `sm_tp_jif_changed_read`, responds to any change in the JIF and readvertises its services; `sm_tp_jif_changed_ignore` ignores all changes. You might want temporarily to install `sm_tp_jif_changed_ignore` for a deployed application so it is unaffected by updates to the JIF, such as addition of new services that are undergoing development.

Some event types use different built-in handlers as the defaults for production and development environments. Default development handlers are called by standard servers that are configured for development; default production handlers are called by standard servers that are configured for production. For example, `sm_tp_advertise_cond_winopen` is the default handler for `advertise` events on a server that is configured for production; this handler conditionally caches in memory the service components of advertised services. A server that is configured for development uses `sm_tp_advertise_log`; this handler never caches service components, and also posts success messages to the request broker's log file.

Table 6-2 lists all built-in handlers and indicates which ones are used as defaults for development (*dev*) and production (*prd*). Descriptions of these handlers can be found in the event type descriptions that follow.

Table 6-2 Built-in event handlers and defaults

Event	Handler
`advertise`	`sm_tp_advertise_cond_winopen` (*prd*)
	`sm_tp_advertise_ignore`
	`sm_tp_advertise_log` (*dev*)
	`sm_tp_advertise_winopen`
`exception`	`sm_tp_exception_no_change`
	`sm_tp_exception_print_all` (*dev*)
	`sm_tp_exception_print_warning`
	`sm_tp_exception_promote_error` (*prd*)
`jif_changed`	`sm_tp_jif_changed_ignore`
	`sm_tp_jif_changed_read` (*dev/prd*)
`message`	`sm_tp_message_ignore`
	`sm_tp_message_print_string` (*dev/prd)*
`post_request`	`sm_tp_post_request_ignore` (*dev/prd*)
`post_service`	`sm_tp_post_service_ignore`
	`sm_tp_post_service_winclose` (*dev*)
	`sm_tp_post_service_winclose_or_deselet` (*prd*)
	`sm_tp_post_service_windeselect`
`pre_request`	`sm_tp_pre_request_ignore` (*dev/prd*)
`pre_service`	`sm_tp_pre_service_ignore`
	`sm_tp_pre_service_winopen` (*dev*)
	`sm_tp_pre_service_winopen_or_select` (*prd*)

Table 6-2 Built-in event handlers and defaults ***(Continued)***

Event	Handler
	`sm_tp_pre_service_winselect`
`request_received`	`sm_tp_request_received_ignore` (*prd*)
	`sm_tp_request_received_jif_check` (*dev*)
`server_exit`	`sm_tp_server_exit_ignore`
	`sm_tp_server_exit_log_down` (*dev/prd*)
`unadvertise`	`sm_tp_unadvertise_cond_winclose` (*prd*)
	`sm_tp_unadvertise_ignore`
	`sm_tp_unadvertise_log` (*dev*)
	`sm_tp_unadvertise_winclose`
`unload`	`sm_tp_unload_immediate`
	`sm_tp_unload_call_origin` (*dev/prd*)

Advertise and Unadvertise Events

`advertise` and `unadvertise` events occur on the successful return of the `advertise` and `unadvertise` commands, respectively. Both events occur only on servers.

`advertise` and `unadvertise` can specify a single service, a group of services, or all services. Each service generates its own set of `advertise` and `unadvertise` events.

Advertise and Unadvertise Handlers

`advertise` handlers are used to perform service initialization. `unadvertise` handlers are used to perform service cleanup.

Scope

`advertise` and `unadvertise` handlers can only be installed at application scope.

Contact

```
advertise_handler(char *cmdStr, char *serviceName,
   char *groupName, char *serviceContainer, int cacheScreen)
unadvertise_handler(char *cmdStr, char *serviceName,
   char *groupName, char *serviceContainer, int cacheScreen)
```

`cmdStr`

The `advertise` and `unadvertise` handler arguments.

serviceName
: The name of the service as specified in the JIF.

groupName
: The name of the service group specified in the `advertise` or `unadvertise` command. If none is specified, this argument is set to null string.

serviceContainer
: The name of a service component associated with the service, as specified in the JIF.

cacheScreen
: Determines whether to cache the service component in memory and so determines how the handlers access *serviceContainer*, with one of these values:

- `TP_ONADVERTISE`—The service component is opened and cached in memory when the service is first advertised.
- `TP_ONFIRSTCALL`—The service component is opened and cached in memory when the service is first called.

- TP_NOCACHE—The service component is opened each time the service is called. When the service returns, the container is closed.

Returns

The return codes from advertise and unadvertise handlers are ignored; a 0 return value is suggested.

Built-in Handlers

Panther provides four built-in advertise handlers:

- sm_tp_advertise_cond_winopen (default production handler) opens the service component of the advertised service and caches it in memory only if the JIF's service definition sets Cache Service Component attribute to TP_ONADVERTISE.
- sm_tp_advertise_log (default development handler) posts a message to the middleware APIs log file indicating that the server has successfully advertised a service. This handler never caches service components in memory, so it always shows the latest changes to a service.
- sm_tp_advertise_winopen opens the service component that is associated with the advertised service.
- sm_tp_advertise_ignore ignores advertise events.

Panther provides four built-in unadvertise handlers:

- sm_tp_unadvertise_cond_winclose (default production handler) closes the service component of the advertised service only if the JIF's service definition sets Cache Service Component to TP_ONADVERTISE.
- sm_tp_unadvertise_log (default development handler) posts a message to the middleware APIs log file indicating that the server has successfully unadvertised a service.
- sm_tp_unadvertise_winclose closes the service component associated with the unadvertised service.
- sm_tp_unadvertise_ignore ignores unadvertise events.

Exception Events

An exception event occurs while the middleware API is performing work on behalf of the application. All request broker commands can generate one or more exception events. Not all exception events are errors; however, all request broker errors are exception events.

Each exception event is associated with an integer code, string constant, and message, which are set in application properties tp_exc_code, tp_exc_names and tp_exc_msg, respectively. tp_exc_names is an array of all exception event type constants that are indexed by the corresponding exception codes. tp_exc_msg contains a string that describes the latest exception event.

For example, this JPL statement displays a message that describes the latest exception event to the user:

```
msg emsg "Error: " ## @app()->tp_exc_names[tp_exc_code] \
   ## "%NMessage: " ## @app()->tp_exc_msg
```

Each exception event sets tp_severity to a severity code, which is also supplied as an argument to the exception handler function.

For a complete list of exception event type constants, refer to page D-1. For exception severity codes refer to page 6-13.

Exception Handlers

When an exception event occurs, its handler performs the required processing. This processing might include displaying a message to the user, cancelling a request, or rolling back a transaction.

On entering an exception handler, Panther sets these application properties:

- tp_exc_code, tp_exc_names, and tp_exc_msg identify and describe the exception event that triggered execution of this handler.

- `tp_severity` is set to the default severity code for this `exception` event. The severity level is also supplied as an argument to the handler function (refer to page 6-12).

For example, the following `exception` handler alerts the user of an error condition whose severity level is equal to or greater than `TP_ERROR`:

```
proc err_print (cmdStr, cmdName, callid, severity)

   if ( severity >= TP_ERROR )

      msg emsg "Error: " @app()->tp_exc_code \
         "Message: " @app()->tp_exc_msg

   return severity
```

The handler should store the value of `tp_exc_code` and/or `tp_exc_names` before it calls any other middleware API-related functions; otherwise these properties are liable to be reset before control returns to the handler. Similar precautions are unnecessary for the handler's severity code, because the severity level is already locally available as a handler parameter.

Exceptions within an Exception Handler

If an `exception` handler generates its own `exception` events, Panther ignores them and does not call any handlers for them, and the `tp_severity` property remains unchanged. To catch exceptions within an `exception` handler, check the `tp_severity` property after each command in the handler. The original handler then decides whether errors within the handler should affect the severity that it returns.

Scope

`exception` events can occur on clients and servers. `exception` handlers can be installed at all scopes.

Contract

`exc_hdl(char *`*`cmdStr`*`, char *`*`cmdName`*`, char *`*`callid`*`, int `*`severity`*`)`

cmdStr

The entire text of the JPL command and arguments that generated the `exception` event.

cmdName

The name of the command only, stripped of any arguments.

callid

The identifier of the service call for which the command was issued. If no service call applies, *callid* is an empty string.

severity

The default severity for the `exception` event.

Returns

An `exception` handler must return a valid severity code. This code is written to the `tp_severity` property, and determines how the application responds to the `exception` event. If the handler returns an invalid return code, `tp_severity` is restored to the severity level it had on entry to the handler. For a description of all exception severity codes, refer to page 6-13.

Warning: If a JPL procedure omits a `return` statement, it returns a value of 0. `exception` handlers that are written in JPL should always explicitly return with the appropriate severity code.

Exception Severity Codes

All `exception` event handlers must return a severity code. Each `exception` event has a default severity, and the `tp_severity` property is set to the corresponding code on entry to the `exception` handler. The default severity code is also supplied as an argument to the handler function. If a request broker event generates multiple exceptions, the exception with the highest severity has precedence and sets `tp_severity`.

The following sections lists severity codes in ascending (lowest to highest) order and describes what action the application takes when a handler returns one of them.

`TP_NONE`

The `exception` event is ignored. Processing of the command or request continues and the `tp_exc_code` property remains unchanged. The status of an enclosing transaction is unaffected.

`TP_INFORMATION`

Processing of the command or request continues. The status of an enclosing transaction is unaffected.

`TP_WARNING`
: Processing of the command or request continues from where the event occurred. The status of an enclosing transaction is unaffected.

`TP_ERROR`
: Processing of the command or request continues, but at the next processing stage, if any. If no processing stage follows, the command terminates. If the exception occurs within a transaction, the transaction's status is set to

`TP_WILL_ABORT`.

`TP_MESSAGE`
: `exception` events of this severity are only raised during the process of exchanging data between clients and servers. An exception of this severity level causes the send/receive process to abort any further attempt to send/receive that message. Any other processing associated with that message is also aborted. If the exception occurs within a transaction, the transaction's status is set to `TP_WILL_ABORT`.

`TP_COMMAND`
: The function or command associated with this severity terminates. No further messages are sent or received. If the exception occurs within a transaction, the transaction's status is set to `TP_WILL_ABORT`.

`TP_REQUEST`
: Applicable only to exceptions that occur while a service request is being processed. Otherwise, a `TP_REQUEST` severity is reduced to a `TP_COMMAND` severity. For a client, all processing associated with the request is terminated. After the request aborts, a `post_request` event occurs. The value of the `tp_exc_code` property is set to the exception code. The status of an enclosing transaction is set to `TP_WILL_ABORT`. If an exception of severity `TP_REQUEST` is generated within a server and is related to its servicing of a client, then the service is terminated with an error status.

`TP_TRANSACTION`
: Aborts the associated transaction. For more information, refer to the `xa_rollback` command.

`TP_CONNECTION`
: If associated with a client exception, the associated connection automatically closes. Refer to the `client_exit` command. If no connection applies, the severity is reduced to the next severity level that is applicable, usually `TP_COMMAND`.

If associated with a server exception, the server aborts its current service and shuts down.

`TP_PANIC`

Operation of the agent, client or server, stops. A client process rolls back all outstanding transactions (via `xa_rollback`), aborts if possible all outstanding service requests, closes all open connections, and terminates. A server aborts its current service as if the severity were `TP_REQUEST`. and shuts down. The connection to the middleware API is severed.

Built-in Handlers

Panther provides four `exception` handlers:

- `sm_tp_exception_promote_error` (default production handler) promotes all exceptions of a `TP_ERROR` severity to `TP_COMMAND`. Thus, all `TP_ERROR` exceptions cause command termination. For other exceptions of severity `TP_WARNING` or higher, this handler displays a windowed exception message.
- `sm_tp_exception_print_all` (default development handler) outputs an exception message to the display or server log file for all exceptions.
- `sm_tp_exception_no_change` does nothing other than accept the initial severity.
- `sm_tp_exception_print_warning` displays a windowed exception message for exceptions of severity `TP_WARNING` or higher.

Jif_changed Events

`jif_changed` events occur when the `jif_check` command detects changes in the JIF since application startup or the last time the JIF was read. Each time the JIF is accessed, `jif_check` examines it for changes.

Jif_changed Handlers

A jif_changed handler should call the jif_read command to reread the JIF, then readvertise services through calls to unadvertise and advertise. If necessary, this handler can also unload public JPL modules that contain service definitions, then reissue the public command on them.

For example, the following handler rereads the JIF and readvertises all services:

```
proc jif_changed_hdlr()
    unadvertise all
    jif_read
    advertise all
    return 0
```

The jif_changed event handler can conditionally reread the JIF by checking the tp_return property, which tells how the JIF changed with one of these values:

- TP_JIF_OLD—Version number decreased, indicating an older version of the JIF is now in place.
- TP_JIF_NOCHANGE—No changes.
- TP_JIF_NEWER—Version number increased, indicating a newer version of the JIF is in place.
- TP_JIF_NOACCESS—The JIF file or its library cannot be accessed.

For example, the following JPL rereads the JIF only if it the current version is more recent than the last:

```
if ( @app()->tp_return == TP_JIF_NEWER )
    jif_read
```

Scope

The jif_changed event is available to clients and servers. A jif_changed handler can only be installed at application scope

Contract

```
jif_changed_handler()
```

Returns

A negative return code causes the jif_check command to abort by raising a TP_USER_ABORT exception of severity TP_COMMAND.

Built-in Handlers

Two built-in jif_changed handlers are provided:

- sm_tp_jif_changed_read (default) rereads the JIF through the jif_read command, and readvertises services as specified by a given server's configuration. If the severity after executing jif_read is greater than TP_WARNING, this handler returns -1; otherwise it returns 0.
- sm_tp_jif_changed_ignore ignores jif_changed events and does not reread the JIF.

Message Events

message events occur when a client receives an unsolicited message, including those generated by the broadcast, and notify commands, or when a subscribed event is posted with the post command.

Message Handlers

A message handler decides what to do with unsolicited messages received by Panther. Typically, an application logs messages to a file or displays them immediately to the user.

If a client needs to process or save a message, the message handler must do so before it returns. Use the receive command with the MESSAGE keyword to access message contents.

Recognizing the Message Source

To ensure that unsolicited messages are received and interpreted correctly, install an application-wide `message` handler that relies on a common protocol for identifying message sources. For example, the protocol might establish that all agents of unsolicited messages use the same field in their message data to identify the message source. On receiving unsolicited messages, the `message` handler then checks this field to determine the nature of the message and how to respond.

In the following example, an unsolicited message is broadcast to a supervisor client to report a security violation. In this application, the first field of an unsolicited message is always named `source`, whose value indicates the message type. The `message` handler first issues the `receive` command to get the contents of `source`; subsequent processing depends on the contents of `source`:

```
broadcast CLIENT "supervisor" TYPE JAMFLEX \
    ({source="bcast_security", ACCOUNT=acct, DATE=date,\
         SECURITY=code, MSG=message})

// Message handler for all unsolicited messages


proc msg_handler(type, subtype)
vars source, account, date, security, message
vars companyNews, teamNews, stock, stock_quote
vars fileStream, acctMsg


// Identify message sender.
receive MESSAGE ({source})
if (source == "bcast_security")
{
    // receive security violation data
    receive MESSAGE ({account, date, security, message})
    // Alert the supervisor
    msg emsg "%A004Security alert: " ## message ## \
        "%NDate: " ## date ## \
        "%NAccount No. " ## account ## \
        "%NCode: " ## code
}


else if (source == "bcast_acct_data")
{
    // receive account data
    receive MESSAGE ({account, date, message})
    acctMsg = account##" "##date##"  "##message

    // write message data to log file
    fileStream = sm_fio_open("/u/acct/logfile", "a")
    if fileStream > 0
    {
       call sm_fio_puts (acctMsg, fileStream)
       call sm_fio_close(fileStream)
    }
}

...

else if (source == "post_comp_news")
{
    // receive posted company news message data
    receive MESSAGE ({ companyNews })
    msg emsg "Latest company news: " ## companyNews
}

...

return 0
```

Scope

`message` events are restricted to clients, and a `message` handler can only be installed at application scope.

Contract

`message_handler(char *`*`msgType`*`, char *`*`msgSubType`*`)`

msgType
: Set to the message data type as specified in the `broadcast` or `notify` command, either `JAMFLEX`, `FML`, `FML32`, or `STRING`. If the application is likely to broadcast different message data types, the `message` handler should evaluate the contents of this parameter and branch execution accordingly.

msgSubType
: Currently unused, reserved for future use.

Returns

Ignored; a 0 return value is suggested.

Built-in Handlers

Two `message` handlers are provided:

- `sm_tp_message_print_string` (default) displays any `STRING`-type messages to the user as a windowed message that requires acknowledgment. Messages of other data types are ignored.
- `sm_tp_message_ignore` ignores all `message` events.

Pre_request and Post_request Events

`pre_request` and `post_request` events occur when a client or server issues a service request. A `pre_request` event occurs just before the actual service request is initiated. A `post_request` event occurs when the service returns, whether or not the service completes normally. Each service request generates a `pre_request` and `post_request` event.

`pre_request` and `post_request` events are typically used to track the number of service requests, and how long they take to return.

Pre_request and Post_request Handlers

A `pre_request` handler is responsible for aborting the service request if execution is inappropriate.

A non-negative (>=0) return code from a `pre_request` handler allows service request processing to continue. A negative return code aborts the request by generating a `USER_ABORT` exception of severity `TP_REQUEST`; a `post_request` event follows immediately follows.

Scope

pre_request and post_request events are available to clients and to servers acting as clients—that is, wherever service requests can occur (refer to service_call). pre_request and post_request handlers can only be installed at application scope.

Contract

```
pre_request_handler(char *callid, char *serviceName, HR>
   char *cmdStr)
post_request_handler(char *callid, char *serviceName, HR>
   char *cmdStr)
```

callid
: The identifier that Panther assigned to the service request.

serviceName
: The name of the invoked service.

cmdStr
: The text of the service_call command and its arguments that issued the service request.

Returns

Ignored; a 0 return value is suggested.

Built-in Handlers

One default handler is provided for each event type:

- The pre_request handler sm_tp_pre_request_ignore ignores pre_request events.
- The post_request handler sm_tp_post_request_ignore ignores post_request events.

Request_received Events

request_received events are generated on a server when it receives service requests. The request_received event occurs before Panther verifies that the service is defined in the JIF.

Request_received Handlers

The request_received handler performs any processing that is related to receipt of a service request. Because a request_received event occurs before Panther verifies a service definition, its handler typically calls jif_check to determine whether the JIF has changed.

A request_received handler typically works together with a jif_changed handler to ensure that all changes to the JIF are known to the server before it processes a service request. For example, if the request_received handler calls jif_check, this command detects any changes to the JIF and generates a jif_changed event. The jif_changed handler can then readvertise the services configured for this server (refer to page 6-16).

For example:

```
// Check whether JIF has changed. If it has, a jif_changed
// event is raised and jif_read executes

proc request_received_hdlr (callid, serviceName)
   jif_check
   return 0
```

request_received handlers abort a service request if it is inappropriate to proceed.

Scope

request_received events are executed only on servers. request_received handlers can only be installed at application scope.

Contract

`request_received_handler(char *callid, char *serviceName)`

callid

The identifier assigned to the service request.

serviceName

The name of the service as invoked by the client.

Returns

A non-negative return code from a `request_received` handler allows continued processing of the service request. A negative return value aborts the service request.

Built-in Handlers

Two `request_received` handlers are provided:

- `sm_tp_request_received_ignore` (default production handler) ignores `request_received` events.
- `sm_tp_request_received_jif_check` (default development handler) calls the `jif_check` command. If the severity after executing the `jif_check` command is greater than `TP_WARNING`, the handler returns -1 and the request is cancelled. Otherwise, 0 is returned. The `jif_check` command raises a `jif_changed` event if the JIF has changed.

Server_exit Events

A `server_exit` event is generated when a server deactivates in an orderly fashion. It is usually the last event that user-written code handles before server deactivation.

Server_exit Handlers

The `server_exit` handler should clean up any resources allocated while the server was active that the middleware API nor Panther are unaware of. It can also log a message that the server is deactivating. For example:

```
// Implementation of a "log_down" server_exit handler

proc server_exit
    log "Server has been brought down."
    return 0
```

Scope

`server_exit` events are restricted to servers, and the handler can only be installed at application scope.

Contract

```
server_exit_handler()
```

Returns

Ignored; a zero return is recommended.

Built-in Handlers

Two `server_exit` handlers are provided:

- `sm_tp_server_exit_log_down` (default) posts a message to the middleware APIs log file indicating that the server is being deactivated.
- `sm_tp_server_exit_ignore` ignores `server_exit` events.

Pre_service and Post_service Events

The `pre_service` and `post_service` events occur every time a server receives a service request:

- The `pre_service` event follows the `request_received` event and verification of the service's JIF's definition, and precedes execution of the service routine.
- A `post_service` event occurs after the service routine completes execution.

`pre_service` and `post_service` events are paired events—each `pre_service` event is eventually matched by a `post_service` event for that same service. Both events are enabled or disabled together.

Pre_service and Post_service Handlers

The `pre_service` handler should abort the service if execution is inappropriate. `pre_service` and `post_service` handlers typically keep track of the number of requests to a service and how much time a service requires to process. These handlers also can associate a service component with the service: the `pre_service` handler opens the service component, while the `post_service` handler closes it.

A non-negative return code from a `pre_service` handler allows continued execution of the service. A negative return code aborts the service: a `TP_USER_ABORT` exception of severity `TP_REQUEST` is generated, and the middleware API informs the client that the service has been aborted with a failure status. A `post_service` event is then generated.

Scope

`pre_service` and `post_service` events are restricted to servers. `pre_service` and `post_service` event handlers can only be installed at application scope.

Contract

```
pre_service_handler(char *callid, char *serviceName,
   char *serviceContainer, int cacheScreen)
```

```
post_service_handler(char *callid, char *serviceName,
   char *serviceContainer, int cacheScreen)
```

callid
The identifier assigned to this service request.

serviceName
The name of the service as invoked by the client.

serviceContainer
The name of a service component associated with the service (from the JIF).

cacheScreen
Tells whether the service component is cached in memory and so determines how the handlers access *serviceContainer*, with one of these values:

- `TP_ONADVERTISE`—The service component is opened and cached in memory when the service is first advertised.
- `TP_ONFIRSTCALL`—The service component is opened and cached in memory when the service is first called.
- `TP_NOCACHE`—The service component is opened each time the service is called. When the service returns, the container is closed.

Returns

Ignored.

Example

The following is an example of a `pre_service` handler:

```
proc pre_service_hdlr(callid, srvcName, srvcContainer, cacheFlag)

vars rcode, log_msg
{
// if there is no service component, do nothing
if (cacheFlag == TP_NOCACHE)
    {
        rcode = sm_jwindow(srvcContainer)
    }
else
    {
        rcode = sm_n_wswlwct(srvcContainer)
        if (rcode < 0)
        {
        // if select failed & ONFIRSTCALL, try to open
            if (cacheFlag == TP_ONFIRSTCALL)
            {
                rcode = sm_jwindow("&&" ## srvcContainer)
            }
        }
    }

if (rcode < 0)
    {
        log_msg = "Error opening " ## srvcContainer
        log log_msg
        return -1
    }
return 0
}
```

The following is an example of a post_service handler:

```
proc post_service_hdlr(callid, srvcName, srvcContainer, cacheFlag)

//if there is no service component, do nothing
if (srvcContainer == ""
return 0
{
if (cacheFlag == TP_NOCACHE)
    {
        call sm_close_window()
    }
else
    {
        // handle as TP_ONFIRSTCALL and/or TP_ONADVERTISE
        call sm_wdeslect()
    }
}
```

Built-in Handlers

Four pre_service handlers are provided:

- `sm_tp_pre_service_winopen_or_select` (default production handler) selects or opens the specified service component depending on whether it is already cached, as specified by the *cacheScreen* parameter (refer to page 6-26).
- `sm_tp_pre_service_winopen` (default development handler) always opens the specified service component as a stacked window and, if configured for debugging or development, logs a `Starting service` %s message. If the container is not found, the handler logs an error and returns -1, causing the service to abort.
- `sm_tp_pre_service_winselect` makes the specified service component the active window in the window stack. If the container is not found, the handler logs an error and returns -1, causing the service to abort.
- `sm_tp_pre_service_ignore` ignores `pre_service` events.

Four `post_service` handlers are provided:

- `sm_tp_post_service_winclose_or_deselect` (default production handler) deselects or closes the specified service component depending on whether caching is specified by the *cacheScreen* parameter (refer to page 6-26).
- `sm_tp_post_service_winclose` (default development handler) closes the specified service component.

  If a standard server is configured for debugging or development, then a `Service` %s completed message is logged.
- `sm_tp_post_service_windeselect` restores the specified service component identified by *serviceContainer* to its original position in the window hierarchy.
- `sm_tp_post_service_ignore` ignores `post_service events`.

Unload Events

An `unload` event occurs when message data is received from an external source whose contents can be written to Panther variables. Message data can be unloaded for these reasons:

- A client receives message data from a returning service. The `unload` handler implements the mapping of reply message data to client variables as specified earlier by the `service_call` command.
- A client executes a `receive` command to obtain an unsolicited message. The `unload` handler implements the mapping of unloaded data as specified by the `receive` command.
- A server executes a `receive` command to obtain message data from a client or another server. The `unload` handler implements the mapping of unloaded data to service variables, as specified by the `receive` command.

All service requests generate an `unload` event on the client even if an error occurs and no data unloads. The application can then decide whether to write to the specified variables, possibly only clearing them of old data.

Unload Handlers

An `unload` handler is responsible for unloading data into Panther variables through the `unload_data` command, and setting conditions for performing the unload.

Note: `unload` handlers can be triggered anywhere, including within another `unload` handler. Therefore, `unload` handlers should not raise another `unload` event, and thus give rise to an infinite loop condition. For instance, if a service call is issued from within an `unload` handler, an `unload` event occurs for that service when it returns. The handler is called again to handle the event, and so on.

Scope

`unload` events are available to both servers and clients, and handlers can be installed at application, transaction, and request scopes.

Contract

```
unload_handler(char *callid, int msgSource, int receiptMethod)
```

callid

Identifies the message's service call. If the message is not associated with a service request (for example, an unsolicited message), this parameter is empty.

msgSource

The source of the message whose receipt caused the `unload` event, indicated by one of the following constants:

msgSource constant	Location	Source
`TP_BLOCKING_RPC`	client/server	Normal blocking `service_call`
`TP_NONBLOCKING_RPC`	client/server	Nonblocking `service_call`
`TP_ARGUMENTS`	server	Arguments sent to the service
`TP_UNSOLICITED`	client	Unsolicited message

receiptMethod

Indicates the type of polling used to obtain the message with one of these constants:

receiptMethod constant	Location	Method
`TP_ASYNCHRONOUS`	client	Data received from asynchronous polling
`TP_BLOCKING`	client/server	Data received while blocking as part of `service_call` command
`TP_SERVER`	server	Message obtained through whatever mechanism is used by a server to receive service requests; only if *msgSource* is `TP_ARGUMENTS`
`TP_WAIT`	client/server	Data received as a result of invocation of wait

Returns

The severity code that an `unload` handler returns indicates whether it successfully processed the `unload` event. `TP_NONE` indicates success. Return any other severity code to indicate failure. A failed `unload` event generates a `TP_UNLOAD_FAILED` exception at the specified severity. If the return code is not a valid severity, a TP_UNLOAD_FAILED exception occurs at the TP_WARNING severity level. For a description of severity codes, refer to page 6-13.

Example

The following unload handler unloads message data only if the service request is successful:

```
proc unload_hdlr (callid, msgSource, msgRcptMethod)
{
    if (@app()->tp_svc_outcome == 0)
    {
        // unload the arguments
        unload_data
    }
    else
    {
        log "Unsuccessful return from service--no arguments \
            unloaded"
    }
    return TP_NONE
}
```

Built-in Handlers

Two unload handlers are provided.

- :sm_tp_unload_call_origin (default) unloads the service message data to the screen where the service call originated. If the screen is no longer active, this handler unloads the message data to the active screen. This handler ensures that results are written to the original screen when application processing has proceeded to a different screen.
- sm_tp_unload_immediate unloads the message data to the active screen, regardless of whether the message data originated from it.

CHAPTER

7 Transaction Model for JetNet

On the client side, service requests are made to access the database by user input on a client screen. These requests can be directed to the transaction manager, which through the middleware transaction model, decides what services need to be called to satisfy the requests.

In two-tier applications, the transaction model generates transaction manager events; in three-tier JetNet applications, the transaction model generates service requests.

`jetrb1` is the transaction model provided for three-tier JetNet applications. When a client uses the transaction manager, `jetrb1` determines which service request to make to satisfy the transaction manager command. `jetrb1` provides processing for the following transaction manager events: `TM_SELECT`, `TM_VIEW`, `TM_DELETE`, `TM_DELETE_EXEC`, `TM_INSERT`, `TM_POST_SAVE`, and `TM_VAL_LINK`.

The service properties of the table views and link widgets on the client screen are checked to determine which services to call. If no services are specified in the properties, the default, built-in services are called.

Built-in Services

The built-in services are used on:

- Three-tier applications that have been converted from two-tier with the `clnt2svr` utility.
- Screens that use the request broker but do not have any Service properties set.

The services perform transaction manager operations for the client by executing sm_tm_command. The operations handled by the built-in services are: SAVE, SELECT, VIEW, INSERT, DELETE, and VALIDATE LINK. These services are advertised by a conversion server. For more information on the conversion server, progserv, refer to page 2-2.The built-in services cannot be modified.

Modifying the Model

The jetrb1 transaction model should prove sufficient for your database processing needs, but should you need to modify the behavior you can do so. If the behavior that requires modifying is application-wide, you can modify the source code of the model directly. Otherwise, you can provide hook functions on a screen-by-screen basis for particular transaction manager events. Write a hook function and assign it to the Database Function property of the screen's table view.

You might want to modify the behavior of jetrb1 to augment the service call functionality, perhaps to add arguments to the service call, to make an additional service call, or to replace the service call with different one.

For example, if a SELECT operation requires non-screen variables, this would require changing the processing of the TM_SELECT event either in jetrb1 or in a hook function written to intercept this event. This could be necessary if a multi-screen transaction requires additional variables, perhaps from the LDB or from non-screen variables.

jetrb1 is a C source module located at $SMBASE/samples/jetrb1.c.

To modify it, first make a copy, then do your edits and rebuild your executables. You will need to add the object module in the makefile in the place reserved for your source modules.

Service Limitations

The built-in services are only called for full commands. If a screen implements partial commands, you will have to either write a service to implement the partial command (and specify the service in the appropriate service property) or modify jetrb1. For example, if a screen has master and detail sections, and both use grids, the detail portion of the screen will not update when the focus in the master section changes to another record.

If a service property is set on a non-root table view, then all table views must have their service properties set. If no service properties on the root table view are set but a service property is set on a subordinate table view on the screen, no action is taken. The built-in services will not be called, and the service in the property that is set will not be called.

Server Processing

On the server end, the middleware API receives service requests. The service requests are sent to the transaction manager, which, using the database transaction model, passes the requests to the SQL generator where the SQL is constructed. From there, the database driver (for your database) sends the appropriate instructions to the database. Responses back from the database travel the same path in reverse.

Transactional Control

Database transactions are database operations that are completed or canceled as a unit. For example, a change to a record in a database can comprise `INSERT`, `UPDATE`, and `DELETE` operations. These operations are executed as part of a transaction, so that if any one fails for any reason, all three are rolled back and the information in the database is restored to its former state. The operations are only completed and committed to permanent storage in the database when all three have succeeded.

Middleware systems do not provide transactional control; they rely on the transaction capabilities of the database as invoked by the application server. In a three-tier application, the transaction model on the application server determines how database transactional processing should be invoked. The Oracle Tuxedo middleware adapter supports XA transactions. For further information on transaction management, refer to page 8-3

CHAPTER

8 Oracle Tuxedo Features

If you are using Panther Oracle Tuxedo Edition, you can take advantage of several enhanced features provided by Oracle Tuxedo. This chapter describes these features:

- `FML`, `FML32` and `STRING` buffer types.
- XA-compliant transactions.
- Service request forwarding.
- Oracle Tuxedo event brokering.
- Oracle Tuxedo reliable queues.

This chapter also provides information on initializing and booting servers in the Oracle Tuxedo environment, including a debuggable server.

Service Data Buffer Types

Data is passed between agents—clients and servers—in data buffers. These data buffers are sometimes referred to as message buffers, or simply messages.

Within the JetNet middleware adapter, these data buffers can contain information in a JAMFLEX buffer. In the case of service call operations, up to two buffers can be passed, one for data incoming to the service, and one for data returned from the service to the calling agent.

Within the Oracle Tuxedo middleware adapter, in addition to the JAMFLEX buffer type, you can use FML, FML32 and STRING buffer types to transport data/messages between agents. The commands that use data transport are service_call, service_forward, service_return, receive, enqueue, dequeue, post, notify, and broadcast.

FML and FML32 Buffers

Oracle Tuxedo provides FML buffers (both FML and FML32) as a medium for handling collections of data fields. FML32 differs from FML in that certain identifiers permit larger sizes—allowing for 32-bit values as opposed to 16-bit values. For example, the maximum number of fields in an FML buffer is 8,191, and for FML32 about 30 million. Differences also apply to individual field sizes and total fielded buffer size. In general, in this documentation, references to FML apply to both types; differences in behavior are noted accordingly.

When you use FML buffers, fields are listed in an FML file. In the file, each field has a name, number, and type, either short, long, float, double or carray. For more information on FML buffers within JPL, refer to page 5-17.

When you use transaction manager defined services with FML buffers, you must define the following FML fields:

FML field	Type
smTmData	carray
smTmRowCount	long
SmTmTv	string

Oracle Tuxedo-specific variables are used by Oracle Tuxedo to locate FML and FML32 files:

- FLDTBLDIR and FLDTBLDIR32—List of directories where FML and FML32 files, respectively, can be found.

- `FIELDTBLS` and `FIELDTBLS32`—Names of the `FML` and `FML32` files, respectively, in the directories defined by `FLDTBLDIR`.

For more information on FML buffers and variables, refer to the Oracle Tuxedo documentation.

STRING Buffers

With the Oracle Tuxedo middleware adapter, Panther supports Oracle Tuxedo `STRING` type buffers. Either a string constant or a variable can be passed. Refer to page 5-18 for more information on these buffer types. Also, refer to the Oracle Tuxedo documentation.

XA Transaction Management

In a Oracle Tuxedo application, you can use the XA protocol to provide transactional control in addition to native database transaction support. Database and queuing operations can be grouped together and executed such that either all of them are completed or none of them are.

JPL commands which support the XA protocol are listed in Table 8-1.

Table 8-1 JPL commands providing XA transactional control

JPL command	Description
`xa_begin`	Begin a transaction.
`xa_commit`	Commit a transaction.
`xa_end`	End a transaction.
`xa_rollback`	Abort a transaction.

You want to use XA transactional control when operations that must be executed together span more than one service, or when more than one resource manager is being accessed. However, if the database operation is contained within one service call, and only one resource manager is being accessed, it is more efficient to use native database transaction support.

For example, a bank account transfer operation can be coded so that two service calls are required to debit one account and credit another. Both services comprise a single transaction as shown in the following example:

```
xa_begin
   service_call withdraw ( .... )
...
   service_call deposit ( .... )
...
xa_end
```

An alternative approach is to write a transfer service that incorporates the code from both procedures `withdraw` and `deposit`. Then, a transaction would not have to be used when calling this service. However, you would still need to use database transaction control around the database operations within the service, that is, the `dbms` begin, `dbms` rollback and `dbms` commit commands, to ensure the integrity of the database.

For more information about transactional support in each Panther database driver, refer to *Database Drivers*.

Note: You cannot mix native database transaction control and XA-transaction control in the same server. If there is an XA connection to the database, all transactional control must be performed with the XA protocol.

Message Forwarding

You can use the `service_forward` command to forward service request data from one service to another. A `service_forward` differs from a `service_call` in that no reply is expected by the agent executing `service_forward`. The responsibility of replying to the initial client agent is handed over to the service that receives the forward request.

Event Brokering

With the Oracle Tuxedo middleware adapter, Panther provides access to Oracle Tuxedo's Event Broker/Monitor. This feature allows clients and servers to generate application-wide events that can be subscribed to by any agent on an individual basis. The Oracle Tuxedo event broker system of event processing is distinct from the other layers of Panther event processing: database event processing, request broker event processing, transaction manager event processing, and screen/widget event processing.

Event posting and subscribing provides another method of communication between agents, one that is more flexible than simple service calling. Communication is not restricted to a one-to-one relationship between a client agent and a server; any agent can post an event that can be received by any number of other interested agents.

Use event brokering for important, infrequent events. The application's performance can be affected if there is an overload of event notifications.

How to Use the Event Broker

Your application configuration must include running the Oracle Tuxedo-provided event broker server `TMUSREVT`. To take advantage of Oracle Tuxedo system-specific events, your configuration must also run the server, `TMSYSEVT`, which process the predefined events.

For information on configuring these servers, refer to your *Oracle Tuxedo Administrator's Guide*.

Accessing the Event Broker

Three commands are used to access event brokering: `post`, `subscribe`, and `unsubscribe`.

There is no formal mechanism for defining application-wide events in your application code. The events are determined as part of the design process of the application, and exist as part of the application specification. Essentially, the events are defined at the time they are posted or subscribed to.

A subscriber can subscribe to an event that might not occur, for instance a stock-changed event when a price exceeds a certain value. When an agent subscribes to an event with the `subscribe` command, the Oracle Tuxedo middleware adapter does not verify that the event named has been defined elsewhere.

For clients, event notification is done via an unsolicited message. For servers, there are two methods: notification by a service call and notification by message queuing. These methods, and more information about event brokering, can be found in the descriptions for the `post`, `subscribe`, and `unsubscribe` commands.

Example: Stock-change Event

An event is a logical condition, defined by the agent that is either posting or subscribing. An event which registers the change in a stock price could be defined in several ways. This example illustrates different ways which this can be done. Assume that the FML fields defined for the `stock.*` events are:

source
event_name
stock_name
price_change
price

A message handler designed to handle these event notifications unloads data from the FML fields into Panther variables:

```
proc msg_handler(type, subtype)
vars source, event_name, stock_name, price_change, price
{
    receive message ({source})
    if (source == "post")
        receive message ({event_name, stock_name, \
            price_change, price})
    else if (source == "notify")
        ...
    else if (source == "broadcast")
        ...
    return
}
```

In the `msg_handler` procedure, the FML field named `source` is set by all agents that generate unsolicited messages, including users of `broadcast` and `notify`. Messages sent by these commands would have different FML fields, and hence, a different `receive` statement.

Note: It is important that a standard method of identifying the source of unsolicited messages be established for the entire application. The handler has no knowledge of where the message originated, it simply has a buffer of data. The handler must know how to interpret message data. For more information on implementing an application-wide message handler, refer to page 6-3.

Posting a change event

If the price of a `cola` stock changes, the event posting could look like this:

```
post EVENT "stock_change" ({source="post", \
    event_name="stock_change", stock_name="cola",\
   price_change="+0.50", price="23.00"}) \
   TYPE FML
```

When posting an event, the event name (argument for `EVENT`) must not begin with a "."—this is reserved for Oracle Tuxedo predefined events.

Subscribing to any change event

An agent could subscribe to a `stock_change` event as follows:

```
subscribe EVENT "stock_change"
```

This notifies the user when any stock changes.

Subscribing to a specific change event

An agent can seek notification only when `cola` stock changes:

```
subscribe EVENT "stock_change" \
   FILTER "stock_name == 'cola'"
```

In this case, the *logical* event subscribed to is cola-change, though this is implemented by subscribing to the `stock_change` event using the `FILTER` option of the command to exclude all `stock_change` events that are not for the `cola` stock.

When subscribing to an event, the event name can be any Oracle Tuxedo regular expression (refer to the *Oracle Tuxedo Reference Manual* for a description of the syntax), for instance:

```
subscribe EVENT "stock.*"
```

Example: Enterprise Bank

The Enterprise Bank sample application makes use of the event broker. When a customer attempts to login to the ATM client, three failures to enter a PIN correctly result in disconnection. A security message is posted to the log file. The event broker posts the security event, which the server picks up. Event notification is done via a service call and the service logs the message to the log file.

The following code fragment from a procedure `init_atm` shows how the event is posted when a login failure condition occurs:

```
vars failed_pin_attempts=0

proc init_atm ()
{
   vars message
...
   /* validate PIN given by the customer */
   service_call "VAL_PIN" ({last_name, pin}, \
      {message, owner_ssn = user_info})


   /* check if validation was not successful */

   if ((@app()->tp_severity > TP_WARNING) || \
      (@app()->tp_svc_outcome == TP_FAILURE))
   {
      msg quiet message
      client_exit
      if (failed_pin_attempts < 2)
      {
         failed_pin_attempts = failed_pin_attempts + 1
      }
      else
      {
         post event "ATM_SECURITY" (last_name)
         failed_pin_attempts = 0
      }
      call sm_n_gofield("pin")
   }
...
   return 0
}
```

The `ATM_SECURITY` event is subscribed to at server initialization. The server initialization routine, `jdbinit.jpl`, loads the public module `server.jpl`, and calls the following procedure `server_subscribe`:

```
proc server_subscribe()
{
   vars message ret
   // Subscribe to ATM security events (3 failed pin
   // entry attempts)
   message = "Server subscribing to event:  ATM_SECURITY"
   log message

   subscribe event "ATM_SECURITY" \
      notification service "LOG_SEC_EVENT"

   // Subscribe to withdrawal limit exceeded events
   message = "Server subscribing to event:  WITHD_LIM_EXC"
   log message

   subscribe event "WITHD_LIM_EXC" \
      notification enqueue qspace "BANKQSPACE" \
        name "WITHD_EXC_Q"
    return 0
}
```

The subscription to the `ATM_SECURITY` event directs the event broker to perform notification of the event (if it occurs) via a service call to `LOG_SEC_EVENT`, which is implemented as screen-level JPL in the `cust.scr` service component. The `log_security_event` procedure logs the message:

```
proc log_security_event()
{
    vars message
    receive args (last_name)
    message = "ATM SECURITY event logged for " ## last_name
    log message
    service_return ()
}
```

Posting and Subscribing

The following steps describe the process involved in posting and subscribing to a event.

1. A client or server posts an event with the `post` command or subscribes to an event with the `subscribe` command. The name of the event is passed as an argument to both commands.

2. Once an event is posted, the event broker determines who has subscribed to the event—applying the rules of a `FILTER` expression where they are provided—and how subscribers should be notified.

3. Clients or servers that have subscribed to the event are notified of the event in the manner specified by the arguments used in `subscribe`. Clients are notified via an unsolicited message; a message handler must be provided that recognizes the posted events. Servers receive notification either by a service call or message queuing.

Unsubscribing

Agents unsubscribe from event notification with the `unsubscribe` command. Events can be unsubscribed from either collectively—using the `ALL` option—or individually—using the event's subscription ID. The subscription ID is required when unsubscribing from a specific event.

You can ascertain the subscription ID immediately after the initial subscribe by obtaining the value of the `tp_return` property. The following example shows how to obtain the event's subscription ID:

```
vars sub_id
...
   subscribe "stock_fall"
   sub_id = @all()->tp_return
...
   unsubscribe SID sub_id
```

Reliable Queues

Message queuing to reliable queues provides an alternative mechanism for interprocess communication between clients and servers. With the Oracle Tuxedo middleware adapter, Panther uses the

Oracle Tuxedo System/Q facility, a system of queue management which includes:

- Stable queues—Ensures enqueued message data is preserved, even if Oracle Tuxedo is shut down.
- A message queuing server, `TMQUEUE`—Responsible for enqueuing and dequeuing messages on behalf of clients and servers.
- A message forwarding server, `TMQFORWARD`—Responsible for forwarding dequeued messages from reliable queues to application servers for processing.

The queuing system provides a location where messages can be stored. These messages can be intended for a variety of purposes—for instance, an agent might want to store a message that contains data intended for another agent. The agent providing the data can enqueue the information onto a queue, where it is available for another agent (or even the same agent) to obtain by dequeuing.

Reliable queues give a three-tier application greater flexibility in client/server communication, providing more methods of controlling the message enqueue/de queue process than it does controlling service calls. Also, if

Oracle Tuxedo goes down, any uncompleted service calls are lost, but enqueued messages are still available when the system comes back up.

For example, an enqueue request can be used to "batch-up" non-time-critical service calls. By choosing a queue that is associated with a service (described below), the enqueuing agent can invoke the service indirectly, independent of its (the client's) execution. Later, at a convenient time, the client or any other agent, can obtain the results by dequeuing from a reply queue—after the original enqueued message has been dequeued and processed by a server.

To use reliable queues:

Your application configuration must include running the Oracle Tuxedo-provided `TMQUEUE` and, possibly, `TMQFORWARD` servers. For information on configuring these servers, refer to the *Oracle Tuxedo Administrator's Guide*. Oracle Tuxedo must be configured for queuing, and for the creation of the application's queues and the queuespaces in which they are grouped.

The queue management server, `TMQUEUE`, is used by the reliable queuing facility to enqueue and dequeue messages. The `TMQFORWARD` server is responsible for dequeuing messages, forwarding them to application services, and enqueuing the reply from the service onto a reply or failure queue.

Enqueuing a Message

The `enqueue` command places a message on a queue. In order to enqueue the message, the names of the queue and its corresponding queuespace are required. In addition, the message data must be in on of the following forms—`JAMFLEX`, `STRING`, `FML` or `FML32`—and match what is defined in the JIF.

Agents can specify and keep track of enqueued messages by either their:

- Correlation ID—Defined by the agent doing the enqueuing, and is subsequently available to the application.
- Message ID—A unique Oracle Tuxedo message identifier generated after `enqueue` executes successfully, and is obtained by the enqueuing agent. It is subsequently available to the agent that does the dequeue. You can reference the message with this ID as long as it remains on the original queue.

Enqueuing agents can also specify:

- The reply queue to which replies should be enqueued.
- The failure queue to which failure responses should be enqueued.
- A time when a message is accessible for dequeuing.
- An absolute priority for the message. The value can range from 1 to 100, the default priority of an enqueued message is 50.
- A location within the queue for the message. The location can be either at the top of the queue, or directly ahead of another queued message as specified by the message ID.
- A return code (integer value) for the message.
- That the enqueue operation be unaffected by normal blocking timeouts.
- That the enqueue operation be executed outside of the current transaction.

Dequeuing a Message

The `dequeue` command removes a message from a queue. To dequeue a message, the names of the queue and its corresponding queuespace are required. The arguments—`JAMFLEX`, `STRING`, `FML` or `FML32`—to receive the message are also required and must match what is defined in the JIF.

Dequeuing agents identify the message to dequeue by one the following:

- Message ID—Generated after `enqueue` executes successfully.
- Correlation ID—Created by the enqueuing agent when the message was enqueued.
- Using the default—Dequeues the message at the top of the queue.

A dequeuing agent can obtain the following information about the message:

- Reply and/or failure queues associated with the message when it was enqueued.
- The ID of the agent that enqueued the message.
- Application authentication key of the client that enqueued the message. For more information on client authentication, refer to on page 3-12 and refer to `client_init`.

- Its priority.
- Return code (integer value) established by the enqueuing agent.

The dequeuing agent can specify that the dequeue operation be unaffected by normal blocking timeouts, and that it be executed outside of the current transaction. It can also indicate that the dequeue should wait for a message if the queue is empty; otherwise dequeue returns immediately.

Defining Reliable Queues

Reliable queues are uniquely identified by their name and the name of the queuespace to which they belong. They can be either *independent* or *service* queues.

To identify and access queues (and their queuespaces):

Define them in the JIF using the JIF editor. Refer to page 25-15 in the *Using the Editors* for instructions on using the JIF editor to define a queue.

Note: The creation of queues is done independently of Panther software; it is part of your Oracle Tuxedo configuration.

Service Queues

Service queues are associated with a particular service and are invoked to process message data. To use service queues, the `TMQFORWARD` server must be running and configured to monitor your queues. Define service queues in the JIF by providing the following information:

- Queue name and queuespace to which the queue belongs. All queues exist within some named queuespace.
- Queue type: one of input, reply, or failure. If the service queue is an input type, the definition of the message parameters to the queue is taken from the definition of the input parameters of the associated service, as defined in the JIF. If the service queue type is either reply or failure, the data from the associated service's output parameters are assigned.
- Name of the service associated with the queue. For input service queues, the name of the service *must* be identical to the name of the queue.

Independent Queues

An independent queue does not have prescribed service call behavior. Independent queue definitions in the JIF require the following:

- Queue name and queuespace to which the queue belongs.
- Message data type: JAMFLEX, STRING, FML or FML32. If the data type is STRING, the name of the argument must be defined. If it is one of the other types, a default name is provided.
- Optionally, a reply and/or a failure queue specification.

Example

The Enterprise Bank sample application uses reliable queues to queue up new accounts for batch processing of a generic bank information mailing.

The Customer Maintenance screen cust_mnt.scr has an event function attached to the customer table view. The Function property identifies the function tm_cust. Part of the tm_cust. event function is reproduced here, showing how the customer information is enqueued to NEW_CUST_Q when a transaction manager insert event is encountered.

```
proc tm_cust (event)
{
vars cust_ssn, message

if (event == TM_VIEW || event == TM_SELECT)
{
...
}
...

else if (event == TM_INSERT_EXEC)
{
/* this is a new customer */
...

   // Queue a notice to send a BANK info packet
   // to this new customer

   enqueue qspace "BANKQSPACE" name "NEW_CUST_Q" \
      ({ owner_ssn,   \
         last_name,  \
         first_name, \
              mid_ini,       \
              cust_address1, \
              cust_address2}) NOREPLYQ NOFAILUREQ

       msg emsg "Saved records of customer :first_name \
          :last_name"
       return TM_CHECK
       }
```

To initiate the mailing, the New Customer Mailing List screen, `custmail.scr`, calls the service `get_newcust`. This service is coded as screen-level JPL on the service component, `get_cust`. The following procedure is the service call code on the client screen:

```
proc get_mail_list()
{
   service_call "GET_NEWCUST" ({owner_ssn, last_name, \
      first_name})
}
```

The service, `get_newcust`, on the service component, dequeues all the new customer entries on the queue:

```
proc get_newcust()
{
/* service GET_NEWCUST */

vars NumQEntries = 0
vars DeQueueStatus = TP_NONE
vars ssn fname lname initial address1 address2

   while (DeQueueStatus == TP_NONE)
   {
      dequeue qspace "BANKQSPACE" name "NEW_CUST_Q" \\
         ({ owner_ssn=ssn,       \\
            last_name=lname,    \\
         first_name=fname,       \\
         mid_ini=initial,        \\
         cust_address1=address1, \\
         cust_address2=address2})

      if (@jam->tp_severity == TP_NONE)
      {
         NumQEntries = NumQEntries + 1
         owner_ssn[NumQEntries]  = ssn
         last_name[NumQEntries]  = lname
         first_name[NumQEntries] = fname
      }
      else
```

```
            {
                DeQueueStatus = @jam->tp_severity
            }
        }
        service_return ({owner_ssn, last_name, first_name})
    }
```

Initializing Servers

When Oracle Tuxedo boots a server, it can read one or more user-supplied server arguments that are supported by Panther. These arguments let you associate a service group with the server, specify its database connection, and so on. For example, the following `CLOPT` entry specifies to initialize a production server that advertises services from the `usr_svcs` service group and connects to database entbank:

```
CLOPT="-- -group usr_svcs -production

    dbms declare dbsession connection for database "entbank""
```

Supply these arguments at the end of the server's `CLOPT` string with the following format:

`-- [ -all | -group` *`serviceGroup`*`] [ -rw ]` *`serverOption`* `]`

    `[ dbms` *`connectString`* `]` *`initRoutine`* `[ args ]`

`-all | group` *`serviceGroup`*

Advertises all services or the specified services in the named service group at startup.

`-rw`

Advertises the default report service for the server is responsible for generating reports. Refer to page 9-23 in the *Reports* for more information on the default report service.

serverOption

Specify one of the following server configuration options:

- `-devel`—Valid for servers that use either the `proserv` or `prodserv` executables, this option establishes the default event handlers on a server for an application that is undergoing development.
- `-production`—Valid only for servers that use the `proserv` executable, this option establishes the default event handlers on a server in a deployed application.
- `-debug`—Valid only for a server that uses the `prodserv` executable, which has the Panther debugger. Set this option to run Panther in debug mode. The debugger starts after default event handlers are established and before the database connection and the server initialization routine.

  Note: If you set this option, also edit its server environment file so that it sets `LD_LIBRARY_PATH` to Motif shared libraries, and `DISPLAY` to tell the X server where to display debuggable service component screens.

For information on handlers for development and production servers, refer to page 3-24.

`dbms` *connectString*

Specifies a database connection through a `DBMS DECLARE CONNECTION` command (refer to page 8-3 for information on connecting to a database). The string must be enclosed in quotes. The command is executed after default handlers are established and before the initialization routine executes.

initRoutine [args]

The initialization routine and any arguments it might use for server initialization. Enter the function name and, optionally, any arguments that it requires, supplied as constant values. The routine is called after default event handlers are established and the database connection is made.

APPENDIX

A Administration Utilities

This chapter describes command-line utilities that can help you develop and manage a Panther application. Utilities are listed in alphabetical order. Utility descriptions are organized into the following components, as applicable:

- Utility name and brief description.
- Syntax line and argument descriptions.
- Description of the utility.

To get a command-line description of a utility's available arguments and command options, type the utility's name with the `-h` switch. For example:

```
rbconfig -h
```

This yields the following output:

```
Usage: rbconfig [-f] [<binary file>]

-f      Output file may overwrite an existing file.
```

clnt2svr

Converts a two-tier application to a JetNet three-tier application

```
clnt2svr [-frv] sourceLib [-p prefixString]
```

-f
: Replace existing libraries.

`-p` *`prefixString`*
: Assign the specified *`prefixString`* to the client and server libraries. For example, if the prefix specification is bank, the client library is assigned the name, `bankcl.lib`, and the server library is `banksv.lib`. If a string is not provided, the library names default to `cl.lib` and `sv.lib`, respectively.

`-r`
: Retain the unnamed JPL procedure on the service component. This can be useful if the unnamed procedure declares variables or carries out any initialization required for the service component.

`-v`
: Output (verbose) the name of the each screen as it is processed and lists the properties that are being changed.

`sourceLib`
: Name of source library that contains screens built with two-tier architecture functionality.

Description

The `clnt2svr` utility converts any two-tier JAM or Panther application that uses the transaction manager to a three-tier application.

Before running the utility, make sure your two-tier client screens, or source, reside in a library. To store screens in a library, run the `formlib` utility.

The `clnt2svr` utility creates a three-tier client library and a server library from a single source two-tier library. The utility makes two copies of each screen from the specified library and moves one of the copies to the new client library (`cl.lib`) and moves the other as a service component to the new server library (`sv.lib`). The source library remains unchanged and intact.

Certain property values are set on the copies while other properties that are pertinent only to a client screen are removed from the corresponding service component in order to avoid unnecessary processing on the server.

Three-tier Client Screens

The client screens in `cl.lib` have the model property (under Transaction) set to `jetrb1`, the request broker transaction model. The client screens use the model to submit service requests to the server. The server is then responsible for the database interaction by the transaction manager.

Service Components

The service components in the `sv.lib` are stripped of the following screen-level property values (if they were set on the source client screen):

- JPL Procedures—The unnamed procedure is removed unless the -r option is used.
- Entry and Exit Functions.
- Menu and Menu Script Name.
- Pointer (cursor specification).
- Wallpaper Pixmap (screen background).
- Icon (image displayed when screen is iconified).

The following properties associated with widgets on service components are stripped of values or changed:

- Active, Inactive, and Armed Pixmap specifications are removed.
- Drop-Down Source for option menus/combo boxes is changed from External Screen to Constant Data since the lists do not need to be populated from an external screen.

rb2asc

Converts a binary JetNet configuration file to ASCII and vice versa

```
rb2asc -a[ -f] [asciiFile cfgFile]

rb2asc -b[ -f] [asciiFile]
```

`-a`	Convert binary files to ASCII
`-b`	Convert ASCII files to binary.
`-f`	The output file can overwrite an existing file.
`asciiFile`	The name of the ASCII file, either the target of ASCII conversion (with -a option) or the source of binary conversion (-b option). If you omit this argument, the default is `broker.asc` in the current directory
`cfgFile`	The name of the configuration file is to convert to ASCII. If you omit this argument, the default is one of the following in this order.

1. The configuration file specified by the environment variable `SMRBCONFIG`.
2. The configuration file specified by the environment variable `TUXCONFIG`.
3. `broker.bin` in the current directory.

Description

The `rb2asc` utility lets you convert a binary JetNet configuration file to ASCII and vice versa. Use this utility in order to put a configuration file under source control or to compare different files.

rbboot

Starts a Panther application

```
rbboot [-a] [cfgFile]
```

`-a`	Start up only administration servers.
`cfgFile`	The name of the JetNet configuration file. If you omit this argument, the default is one of the following, in this order

1. The configuration file specified by the environment variable `SMRBCONFIG`.
2. The configuration file specified by the environment variable `TUXCONFIG`.
3. `broker.bin` in the current directory.

Description

`rbboot` starts all Panther application components such as servers, as defined in *`cfgFile`*. If you omit specifying a configuration file, `rbboot` checks whether environment variables `SMRBCONFIG` or `TUXCONFIG` are set; if not, it looks for `broker.bin` in the current directory.

Before starting an application, verify the following conditions:

- All machines have the executables for which their servers are configured.
- Each machine has `SMRBCONFIG` set to the same value as its Local JetNet Configuration File property (refer to page 3-14).
- In a multi-machine application, the listener process is running on each machines. Start the listener process with `rblisten`.

Note: If an application takes more than two minutes to start up, `rbboot` times out and posts an error message. When this happens, `rbboot` exits without booting any other application servers. To boot the rest of the application, run `rbboot` again.

rbconfig

Creates a JetNet configuration file

```
rbconfig [-f] [cfgFile]
```

`-f`	Overwrite *`cfgFile`* if it already exists. If you omit this option and *`cfgFile`* exists, `rbconfig` issues an error message and exits.
`cfgFile`	The name of the new configuration file. If you omit this argument, the default is one of the following, in this order:

1. The configuration file specified by the environment variable `SMRBCONFIG`.
2. The configuration file specified by the environment variable `TUXCONFIG`.
3. `broker.bin` in the current directory.`rbconfig` creates a minimal

Description

JetNet configuration file that you can use as a starting point for application development. You can subsequently edit this file through the JetNet manager.

`rbconfig` creates a single-machine configuration that is enabled for workstation connections. No servers are defined. With this configuration, you can activate the application and connect to it from a PC workstation. You must define servers in order to make this a working application.

rblisten

Starts the listener process

```
rblisten -p portNum [-h host]
```

`-p` *`portNum`*	The port number to be used by the listener process. This argument and the port number that the configuration file specifies for *`host`*'s Listener Port property must be the same.
`-h` *`host`*	The name or IP address (in dot notation) of this host's network address where the listener awaits a message from the master machine to begin the boot process. If you omit this argument, `rblisten` uses the machine's default host name.

Description

`rblisten` starts the listener process on the current machine. This process must be running on each machine that is defined in an application's JetNet configuration before you boot the application. At boot time there is no bridge process to receive communication. Instead, each listener process on the non-master and backup master machines awaits a message from the master machine to begin the local boot process. The master machine uses the port number in each machine's Listener Port property to address its listening process (refer to page 3-15). Consequently, the port numbers supplied to `rblisten` and specified in the JetNet configuration file must match.

Starting a listener process on the master machine is optional when booting an application from that machine. However, the master machine must have a listener process in order to restart it from another machine.

`rblisten` authenticates most service requests by reading a file with a list of passwords and checking that any process requesting a service contains at least one of the passwords found in the file. If a file named `.adm/tlisten.pw` in the application directory is not found, the passwords are obtained from the file `$SMBASE/udataobj/tlisten.pw`. A zero-length or missing password file disables password checking which generates a warning in the ULOG file. Panther installs a default `tlisten.pw` in `$SMBASE/udataobj/tlisten.pw` with the password Panther.

Note: Add the appropriate call to rblisten to the system startup scripts file of each machine (for example, on SUN workstations `/etc/rc.local`) so that the utility runs automatically when the machine reboots.

rbshutdown

Shuts down a Panther application

```
rbshutdown [-f] [cfgFile]
```

`-f`	Forcibly deactivates the application and disconnects all clients connected to it. If you omit this option and the application has clients connected to it, `rbshutdown` leaves the application active and issues an error message.
`cfgFile`	The name of the JetNet configuration file. If you omit this argument, the default is one of the following, in this order:

1. The configuration file specified by the environment variable `SMRBCONFIG`.
2. The configuration file specified by the environment variable `TUXCONFIG`.
3. `broker.bin` in the current directory.

Description

`rbshutdown` shuts a Panther application and all associated application components such as servers as defined in *`cfgFile`*. If you omit specifying a configuration file, `rbshutdown` tries to get `SMRBCONFIG` or `TUXCONFIG` from the environment; if neither is set, it looks for `broker.bin` in the current directory. Use the `-f` option to ensure shutdown of an application that has clients connected to it.

Note: If a server is still booting when `rbshutdown` is invoked, the utility can time out before the servers are available for shutdown. Run `rbshutdown` again after all servers have finished booting.

APPENDIX

B Converting to a Three-tier Application

When you design your application there are many factors to take into account. For instance:

- The number of users who will be connected to the database, and the frequency of access to that database.
- Complexity of the database access.
- And, the overall load on the database and response time when the load is at its peak.

In general, Panther is designed to help you build large, enterprise-wide applications that utilize multiple servers and distributed databases. But, you can also build simple client/server applications that operate on a one-server machine. However, as application needs become more complicated because more frequent access to the database is required, or the number of users increase, it's time to consider a multi-tier, enhanced client/server solution.

With the `clnt2svr` (client-to-server) utility, you can convert a two-tier application that uses the transaction manager to a three-tier, enterprise-wide application. This chapter describes:

- Requirements for running a converted application.
- How to enhance the functionality of a converted application.

Converting an Application from Two- to Three-Tier

The `clnt2svr` utility converts any two-tier JAM or Panther application that uses the transaction manager.

Before running the utility, make sure your two-tier client screens, or source, reside in a library. To store screens in a library, run `formlib`.

The `clnt2svr` utility creates a three-tier client library and a server library from a single source two-tier library. The utility makes two copies of each screen from the specified library and moves one of the copies to the new client library (`cl.lib`) and moves the other as a service component to the new server library (`sv.lib`). The source library remains unchanged and intact.

Property Settings

Certain property values are set on the copies while other properties that are pertinent only to a client screen are removed from the corresponding service component in order to avoid unnecessary processing on the server.

Three-tier Client Screens

The client screens in `cl.lib` have the Model property (under Transaction) set to `jetrb1`, the request broker transaction model. The client screens use the model to submit service requests to the server. The server is then responsible for the database interaction by the transaction manager.

Service Components

The service components in the `sv.lib` are stripped of the following screen-level property values (if they were set on the source client screen):

- JPL Procedures—The unnamed procedure is removed unless the `-r` option is used.
- Entry and Exit Functions.

- Menu and Menu Script Name.
- Pointer (cursor specification).
- Wallpaper Pixmap (screen background).
- Icon (image displayed when screen is iconified).

The following properties associated with widgets on service components are stripped of values or changed:

- Active, Inactive, and Armed Pixmap specifications are removed.
- Drop-Down Source for option menus/combo boxes is changed from External Screen to Constant Data since the lists do not need to be populated from an external screen.

Requirements for Running a Converted Application

The installed middleware transaction model handles the service requests made by the client in a converted application. The model uses a built-in service to pass the request to the database transaction model and thereby handle most database interactions. To begin using your newly converted application, you need to:

- Make sure a conversion server is running. This ensures that the built-in services are available to the new client screens and service components. Refer to page 3-26 for information on initializing a conversion server.
- Set `SMFLIBS` to include the new client library in the client environment and do the same on the server to include the new server library. This ensures that the libraries are open on start up of the application.

Ensuring Usability

Once you get your new application up and running, there are some things that you should note and consider, depending on the kinds of processing or specifications that existed in your two-tier application. For example:

- If your source two-tier client screens used Continue operations—such as First/Last Record—via push buttons or menu options, these types of operations will not function in the three-tier architecture.
- There is no support for any database processing that does not use the transaction manager; that is, if you made `DBMS` calls directly to the database in your two-tier processing, these database interactions are not moved from the client to the server. You will have to define a service to handle such processing (refer to the next section for information on enhancing a converted application) or ensure that the client has a direct connection to the database.
- In general, any processing your two-tier screens performed (JPL execution, validation, transaction manager event functions) should be reviewed to ensure that the processing is being performed appropriately and by the appropriate agent. Depending on the type of processing, you need to determine whether the processing is best carried out on the client or on the server. For example, data entry validation might best be performed on the client, while database validation might be better performed on the server by way of a service call.
- Partial commands are not supported. A partial command is one that specifies a table view parameter and therefore operates on a portion of the tree, but not for all linked table views on the screen. Because the server cannot determine or maintain the state of information from one service call to the next, there is no guarantee that two sequential requests will be processed by the same server. Partial commands must be handled by defining a new service (refer to the next section for information on enhancing a converted application).
- JPL (or C) programs that make runtime property changes to your two-tier client screen, particularly Transaction and/or Database properties, will not function as expected once you convert your application. In general, these changes are not propagated from the three-tier client screen to its corresponding service component. For example, if you change the Parent Table at runtime, the service component is not aware of this change and the results returned to the client may not be as predicted.

Enhancing a Converted Application

You can implement a custom service to handle specific types of database interactions by assigning services via the client screen's table view. When you provide a service property value and define the service in the JIF, your application can implement a service for individual operations. For example, your application can use the built-in service to handle `SELECT` and `INSERT` operations, but use a custom service to handle an `UPDATE`.

For information on creating services and service components, refer to page 5-1.

To implement a custom service:

- Define the services in the JIF (refer to page 25-1 in the *Using the Editors* for instructions on using the JIF editor).
- You can implement a custom service for each server view on a screen. To do this you would specify the service in the appropriate Service property for the server view's master (root) table view. You can specify Delete, Insert, Select, and Update Services.
- Ensure that a server is initialized that advertises the new services.

APPENDIX

C Enterprise Bank

The Enterprise Bank sample application is provided as a demonstration of some of Panther's capabilities for building three-tier applications. After you become familiar with its functionality, use the Panther authoring environment to investigate how Enterprise Bank works. You should also look at the JPL modules that are used to call services from client screens and initiate database interactions from the service components.

The Oracle Tuxedo middleware adapter supports additional features, including support for Oracle Tuxedo System/Q message queuing and Oracle Tuxedo event brokering. These two features are used in Enterprise Bank, but are not visible to JetNet users. In this description of Enterprise Bank, features that rely on message queuing and event brokering are noted as such.

Instructions for building and running Enterprise Bank are given in `$SMBASE/samples/entbank/README.txt`.

The User's View of Enterprise Bank

Enterprise Bank is a small database application that models some of the simple tasks present in a real banking application. The database, `entbank`, was created with JDB, Panther's simple relational database manager that is used for building application prototypes. Using JDB allows you to test data entry and effect database transactions within Panther as part of the application development process.

However, since JDB is not an XA-compliant resource manager, transactions in Enterprise Bank are not controlled by the monitor. Only XA-compliant resource managers allow transactional control by a monitor. Also, XA transactional control is only available with Oracle Tuxedo.

Running Enterprise Bank

Enterprise Bank resides in the `samples/entbank` directory in your Panther installation. There you will find a `README.txt` file that has detailed instructions on how to build and run the application. These instructions include directions for correctly configuring your environment to run Enterprise Bank, as well as pertinent Oracle Tuxedo-specific instructions.

You can run Enterprise Bank either as an ATM customer client, or as a bank employee or administrator client.

Enterprise Bank Customer ATM Client

The ATM client allows you to perform only customer-related tasks, such as personal account deposits and withdrawals.

Starting the Customer ATM Client

To launch the ATM customer client, start Panther specifying the top-level customer Enterprise Bank screen as the command-line parameter:

```
$SMBASE/util/prodev atm
```

In Windows, double-click on the EntBank Customer ATM icon.

When Enterprise Bank is started, the ATM welcome screen is displayed.

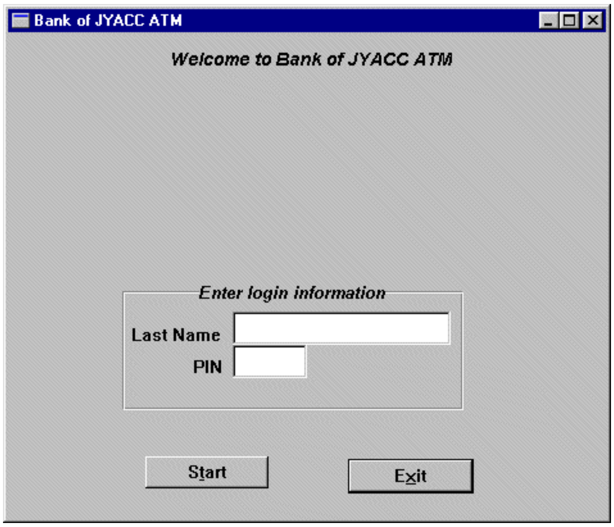


Figure C-1 Customer ATM login screen.

ATM Services

Logging on to the ATM requires a valid customer Last Name and PIN. You can use any of those that are present in the customer database table. Try Last Name `DUCK` and Password `ducky`. Notice that entry into the PIN field is not visible since this is confidential information. Once you have correctly entered the login data, choose the Start push button to initiate customer services.

Security Violation Alert

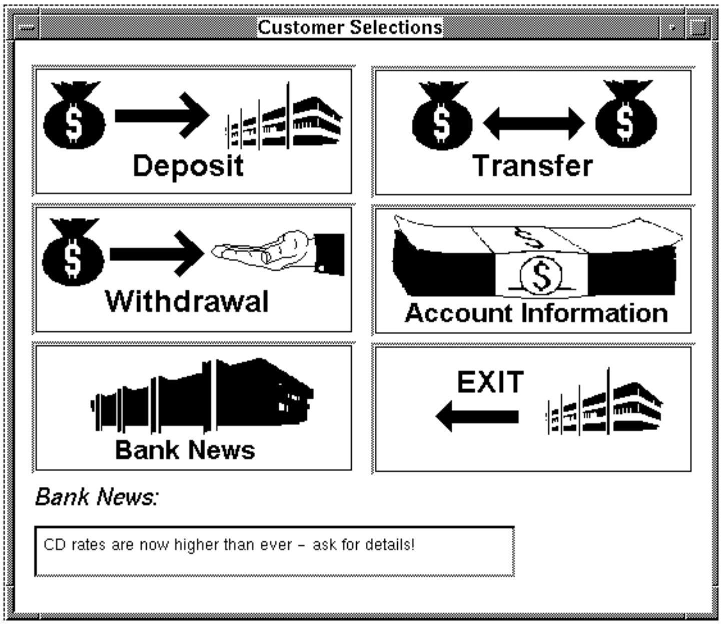
If the user fails to login correctly after three attempts, the client connection is terminated and a security violation event (`ATM_SECURITY`) is posted to the event broker. The server establishes its subscription to this event at initialization time. Notification of the event is done via a service call (service `LOG_SEC_EVENT`). This service logs a message to the log file that includes the name of the user that failed to login. This process uses the Oracle Tuxedo event brokering feature.

For more information on the event broker and more detail on the coding required to implement the `ATM_SECURITY` event, refer to page 8-8. Comprehensive information on the event broker is in your Oracle Tuxedo documentation.

Customer Selections

Once you have successfully logged on to the ATM, the Customer Selections screen opens. From this screen you can choose any of the following features:

- Make a deposit to an account.
- Make an account withdrawal.
- Transfer money between accounts.
- Make account balance inquiries.
- Find out what's new.
- Exit.



CD rates are now higher than ever – ask for details!

Figure C-2 Customer Selections screen presents the services that are available to a user of the Customer ATM client.

Make a Deposit

Choose the Deposit push button on the Customer Selections screen to open the Deposit screen. This screen contains a pulldown menu from which you can choose which of your accounts you wish to access. Enter the amount of the deposit either from a keypad or by direct text entry. Once the amount has been entered and the Deposit button chosen, the new balance appears. Choose Done to dismiss the screen.

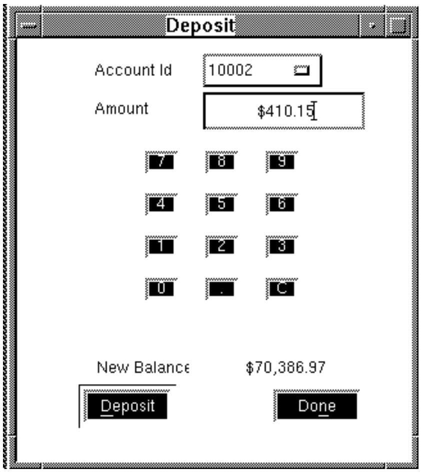


Figure C-3 Deposit screen permits either keypad or direct text entry of the deposit amount, and provides a pulldown menu from which to select the account.

Make a Withdrawal

Choose Withdrawal on the Customer Selections screen to open the Withdraw screen. This screen is identical in function and appearance to the Deposit screen except that it has a Withdraw button instead of a Deposit button. Choose Done to dismiss the Withdraw screen.

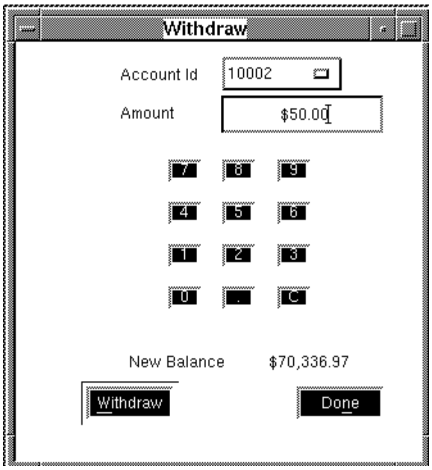


Figure C-4 A customer can make an account withdrawal from the Withdraw screen.

Withdrawal Limit Exceeded

If the user attempts to withdraw an amount in excess of the maximum withdrawal amount, the request fails and a `WITHD_LIM_EXC` event is posted to the event broker. The server establishes its subscription to this event at initialization time. Notification of the event is done via enqueuing a message to the `WITHD_EXC_Q` queue. This process uses the Oracle Tuxedo event brokering and reliable queue system.

For more information on the event broker, refer to page 8-5. For more information on message queuing, refer to page 8-11. Comprehensive information on the event broker and message queuing is in your Oracle Tuxedo documentation.

Transfer

The Transfer screen contains two pulldown menus: one for the debit account id and another for the credit account id. Enter the amount to transfer from the debit account to the credit account in the same manner as on the Deposit and Withdraw screens: through keypad entry or direct entry. After the accounts have been selected and the amount specified, choose the Transfer button to perform the transfer and display both new balances in the Debit Balance and Credit Balance text areas. Choose Done to dismiss the screen.

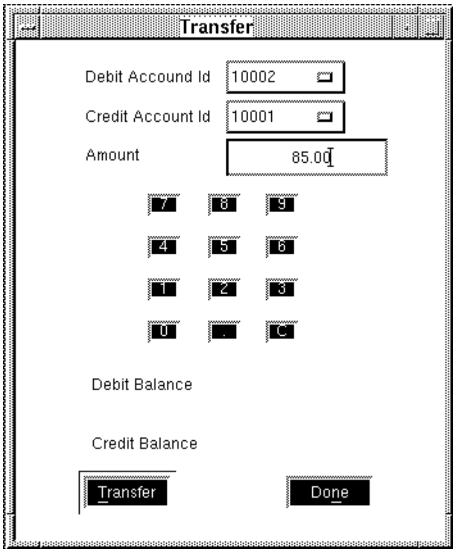


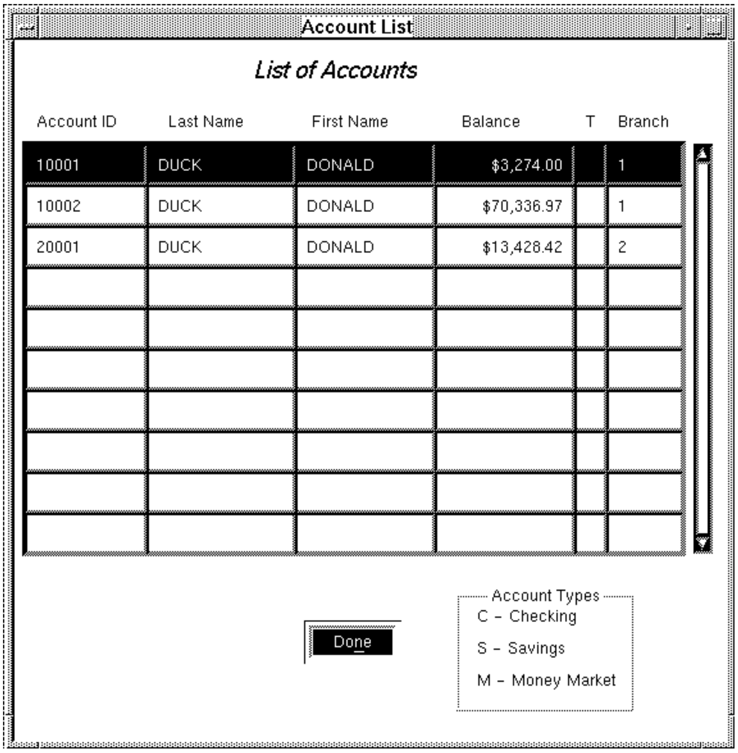
Figure C-5   Transfer screen permits an ATM user to transfer money between accounts.

Bank News

Choose Bank News to display a news message at the bottom of the Customer Selections screen.

Balance Inquiry

To view all of the current balances in your accounts, choose the Account Information button from the Customer Selections screen. The Account List screen opens listing the following information on all your accounts: account id, name, balance, account type, and branch id. A legend is displayed describing the account type designator. Dismiss the window by choosing Done.



Account List

List of Accounts

Account ID	Last Name	First Name	Balance	T	Branch
10001	DUCK	DONALD	$3,274.00		1
10002	DUCK	DONALD	$70,336.97		1
20001	DUCK	DONALD	$13,428.42		2

Account Types
C – Checking
S – Savings
M – Money Market

Figure C-6 A customer can examine the status of his/her accounts on the Account List screen.

Exit Customer Services

When you have finished your customer transactions, exit the ATM client by choosing Done.

The Enterprise Bank Employee Client

The Bank Employee client allows you to perform employee-related tasks, such as opening accounts and updating account information.

Starting the Bank Employee Client

To start the employee client, start Panther, specifying the top-level employee Enterprise Bank screen as the command-line parameter:

```
$SMBASE/util/prodev branch
```

In Windows, double-click on the EntBank Employee WS icon.

When Enterprise Bank is started, the Employee Workstation welcome screen is displayed.

Logging on to the employee client requires a valid employee last name and password. You can use any of those that are present in the employee database table. Try last name `FLOYD` and password `farmers`. Once you have correctly entered the login data, choose the Start push button to initiate employee services.

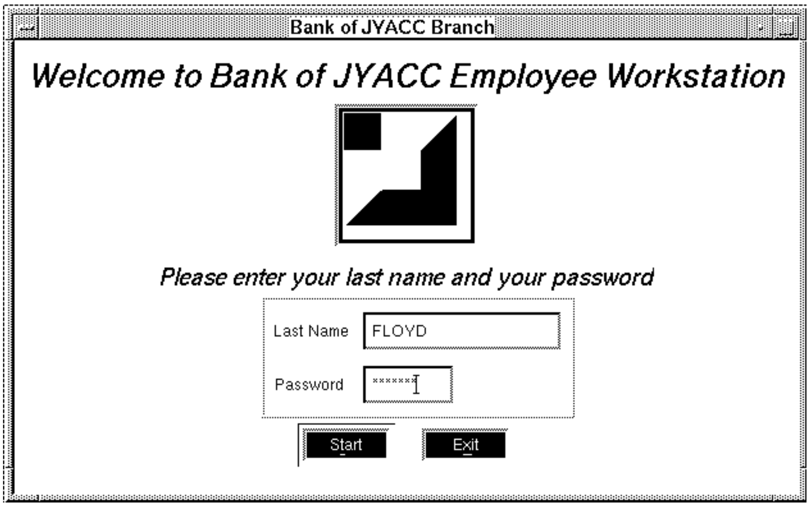

Bank of JYACC Branch
Welcome to Bank of JYACC Employee Workstation
Please enter your last name and your password
Last Name
FLOYD
Password
****
Start
Exit

Figure C-7 Logon as an employee: the Employee Workstation

Employee Services

The employee and administrator clients are both menu-driven. The actions that you can take are available from the Enterprise Bank menu bar. The options present on the menu bar give you access to all employee features.

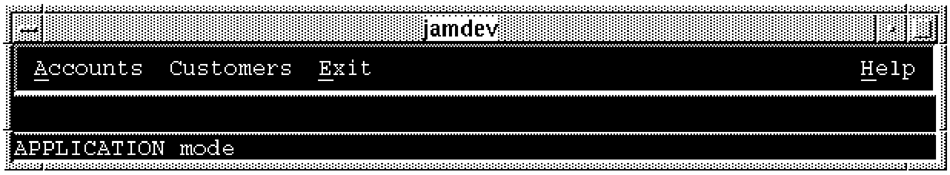
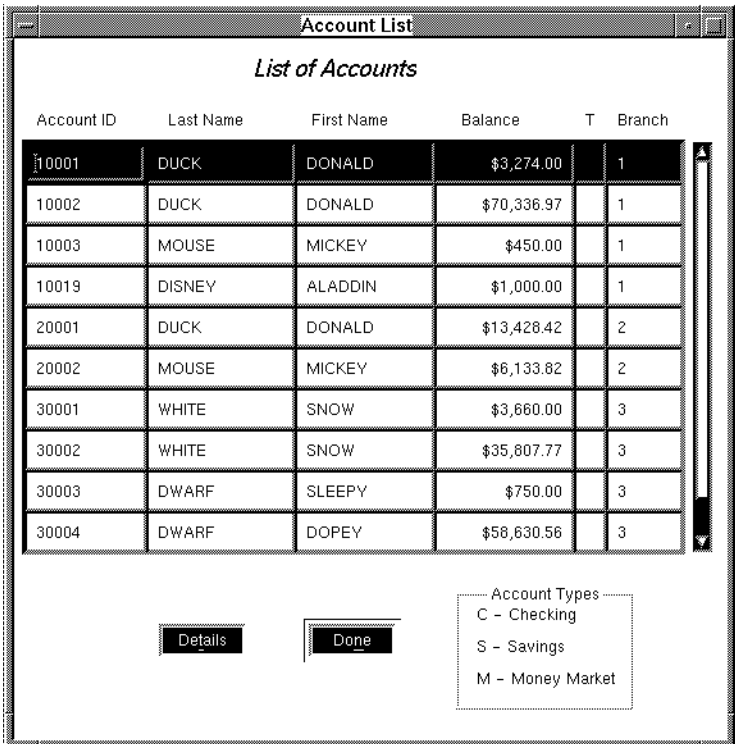


Figure C-8 Employee services are available from pulldown menus on the menu bar.

Accounts Menu Option

The Accounts menu option contains two selections on its pulldown menu: Account List and Account Maintenance.



List of Accounts

Account ID	Last Name	First Name	Balance	T	Branch
10001	DUCK	DONALD	$3,274.00		1
10002	DUCK	DONALD	$70,336.97		1
10003	MOUSE	MICKEY	$450.00		1
10019	DISNEY	ALADDIN	$1,000.00		1
20001	DUCK	DONALD	$13,428.42		2
20002	MOUSE	MICKEY	$6,133.82		2
30001	WHITE	SNOW	$3,660.00		3
30002	WHITE	SNOW	$35,807.77		3
30003	DWARF	SLEEPY	$750.00		3
30004	DWARF	DOPEY	$58,630.56		3

Account Types
C – Checking
S – Savings
M – Money Market

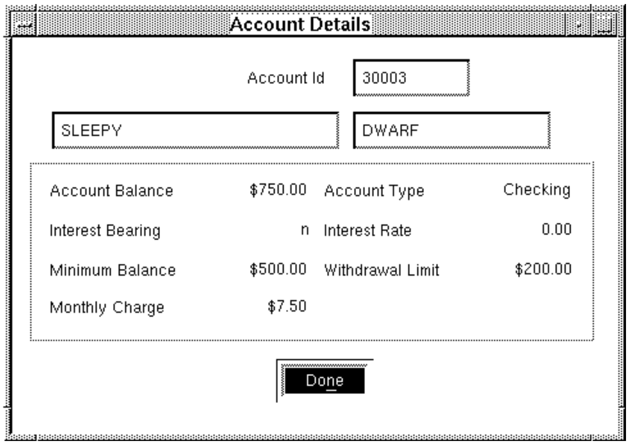
Figure C-9 Account List screen displays all customer accounts sorted by account number.

Display a Complete List of Accounts

Choose Accounts→Account List to open the Account List screen. All account numbers are listed along with the following account information: customers name, the balance of the account, the account type and the branch ID of the account. Dismiss the Account List screen by choosing Done.

Display a Single Account

To examine detailed information for an account, select the account on the Account List screen. Choose the Details push button to open the Account Details screen.



Account Id	30003		
SLEEPY	DWARF		
Account Balance	$750.00	Account Type	Checking
Interest Bearing	n	Interest Rate	0.00
Minimum Balance	$500.00	Withdrawal Limit	$200.00
Monthly Charge	$7.50		

Figure C-10 Examine the details of a given account by bringing up the Account Details screen from the Account List screen.

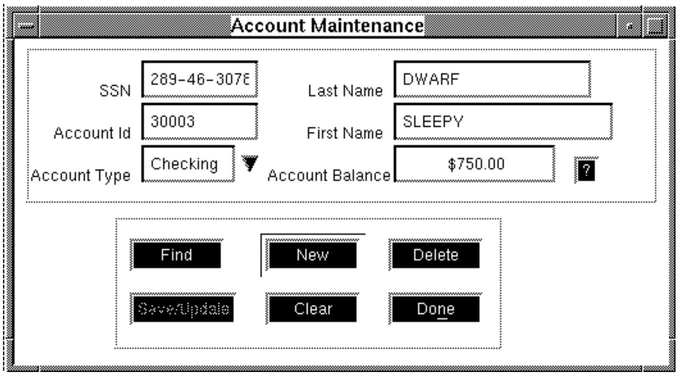
This screen lists the following info for the selected account:

- An indicator for interest bearing accounts.
- The minimum balance required for the account.
- The monthly charge for the account.
- The current interest rate of the account.
- The withdrawal limit on the account.

Dismiss the Account Details screen by choosing Done.

Examine a Customer's Accounts

The Account Maintenance screen is used by an employee to examine a customer's existing accounts, add a new account, or delete an existing account. Choose Account→Account Maintenance from the menu bar to open the screen.


Account Maintenance

SSN: 289-46-3078

Last Name: DWARF

Account Id: 30003

First Name: SLEEPY

Account Type: Checking

Account Balance: $750.00

Figure C-11 A bank employee can update, add, or close an account with the Account Maintenance screen.

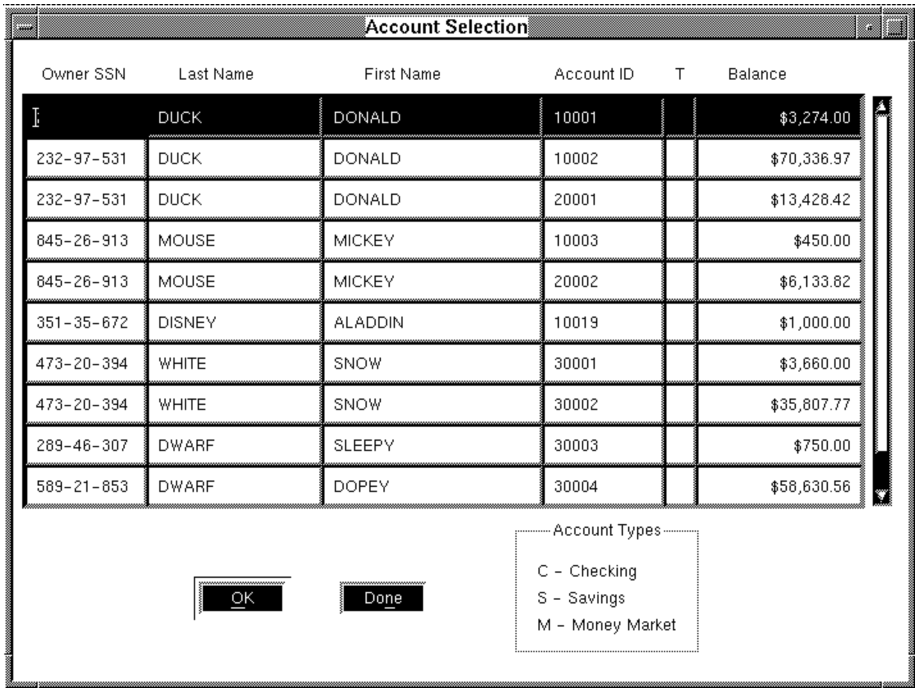
To retrieve a specific account, enter the customer's name, SSN or Account Id. This screen employs a pattern searching mechanism that permits you to enter any character[s], including the wildcard ("%") character, in the name or SSN fields. Choose the Find button after the pattern search data has been entered. If more than one account satisfies the search criteria, the Account Selection screen will open from which you can select the account (see below). If only one account matches the search criteria, that account's data will appear in the Account Maintenance screen. Once the information is on the screen, the account updating features become enabled:

- The Delete button permits you to close the account.
- The New button permits you to enter new data for a new account. When the new account data has been correctly entered, choose Save to add the new account.

Select an Account for Maintenance

The Account Selection screen is used to select an account for maintenance when the specific identifying information is not at hand. It is opened by entering pattern search data in any of the identification fields on the Account Maintenance screen.

This screen displays the following information within scrolling lists: SSN, name, account id, account type, and account balance. To select an account, select the SSN, then choose the OK button. Alternatively, dismiss the screen by choosing Done.



Owner SSN	Last Name	First Name	Account ID	T	Balance
	DUCK	DONALD	10001		$3,274.00
232-97-531	DUCK	DONALD	10002		$70,336.97
232-97-531	DUCK	DONALD	20001		$13,428.42
845-26-913	MOUSE	MICKEY	10003		$450.00
845-26-913	MOUSE	MICKEY	20002		$6,133.82
351-35-672	DISNEY	ALADDIN	10019		$1,000.00
473-20-394	WHITE	SNOW	30001		$3,660.00
473-20-394	WHITE	SNOW	30002		$35,807.77
289-46-307	DWARF	SLEEPY	30003		$750.00
589-21-853	DWARF	DOPEY	30004		$58,630.56

Account Types

C – Checking
S – Savings
M – Money Market

Figure C-12 Account Selection screen lets you select an account for updating in the Account Maintenance screen.

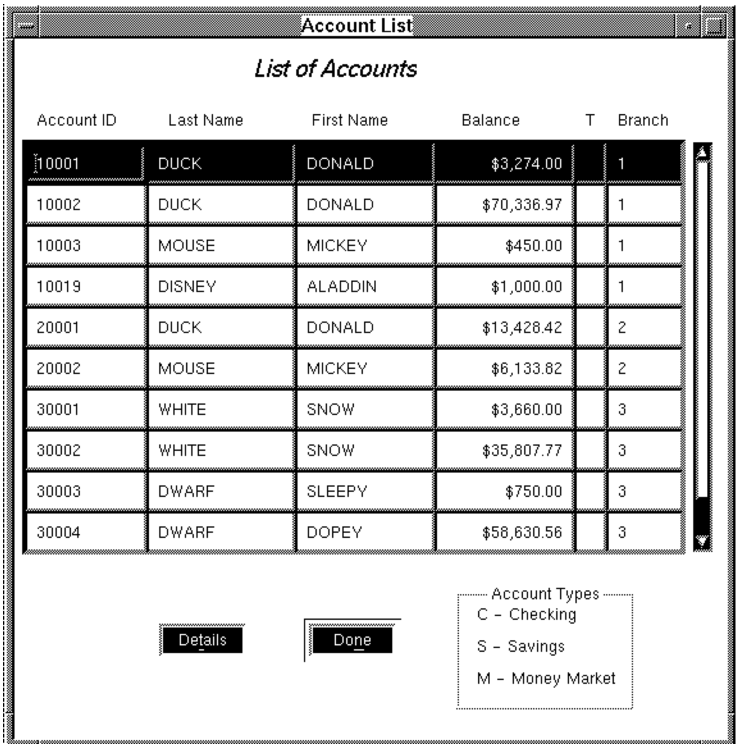
Customers Menu Option

The Customers menu option contains three selections on its pulldown menu: Customer List, Customer Maintenance, and New Customer Mailings.

Display a Complete List of Customers

Choose Customers→Customer List to raise the Customer List screen. This screen displays a list of all bank customers showing name and SSN. You can select any of the SSN entries and choose the Details button to raise the Customer Details screen that provides more customer information.

Note: Accessing the Customer Details screen from the Customer List screen demonstrates the use of asynchronous service calls. The data for the Details screen comes from two separate tables, both of which use the same key: `owner_ssn`. In the JPL code for the `c_detail` screen, service calls are made to the services `FINDCUST` and `GET_ACCT`. Since the call to `GET_ACCT` does not depend on the completion to the call to `FINDCUST`, the call to `FINDCUST` is made asynchronously.



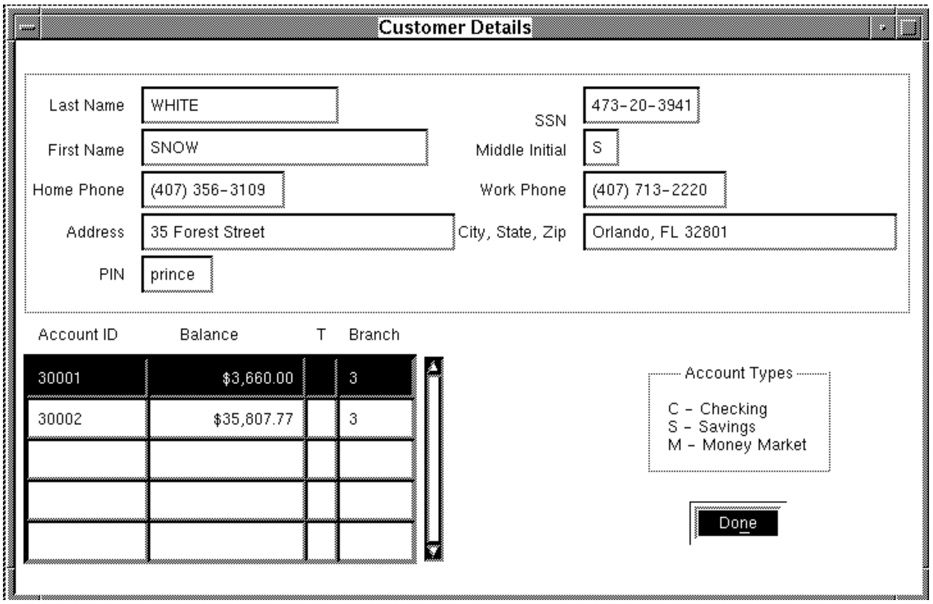
Account ID	Last Name	First Name	Balance	T	Branch
10001	DUCK	DONALD	$3,274.00		1
10002	DUCK	DONALD	$70,336.97		1
10003	MOUSE	MICKEY	$450.00		1
10019	DISNEY	ALADDIN	$1,000.00		1
20001	DUCK	DONALD	$13,428.42		2
20002	MOUSE	MICKEY	$6,133.82		2
30001	WHITE	SNOW	$3,660.00		3
30002	WHITE	SNOW	$35,807.77		3
30003	DWARF	SLEEPY	$750.00		3
30004	DWARF	DOPEY	$58,630.56		3

Account Types
C – Checking
S – Savings
M – Money Market

Figure C-13 Customer List screen shows all bank customers.

Display a Single Customer

The Customer Details screen displays the following customer information: name, SSN, address, home phone, work phone, and PIN. For each account that the customer holds, the following fields are displayed: account id, account balance, account type (C: checking, S: savings. or M: money market), and account branch id.


Customer Details

Last Name	WHITE	SSN	473-20-3941
First Name	SNOW	Middle Initial	S
Home Phone	(407) 356-3109	Work Phone	(407) 713-2220
Address	35 Forest Street	City, State, Zip	Orlando, FL 32801
PIN	prince		

Account ID	Balance	T	Branch
30001	$3,660.00		3
30002	$35,807.77		3

Figure C-14 Customer Details screen contains all the information for a customer.

Update Customer Information

Choose Customers⟶Customer Maintenance to open the Customer Maintenance screen. This screen is used to update existing customer information or add a new customer. The screen contains the following data entry fields: Last Name, First Name, Middle Initial, SSN, Home Phone, Work Phone, Address, City, state, Zip, and PIN. To add a new customer, choose the New button, enter the data, and then choose save/Update.

To change an existing customer's information, enter either the customers SSN or name and choose the Find button. You can use the pattern searching mechanism to obtain customer data by entering any combination of characters with the wildcard ("%") character in these fields. The customer's data is fetched and changes can be made.

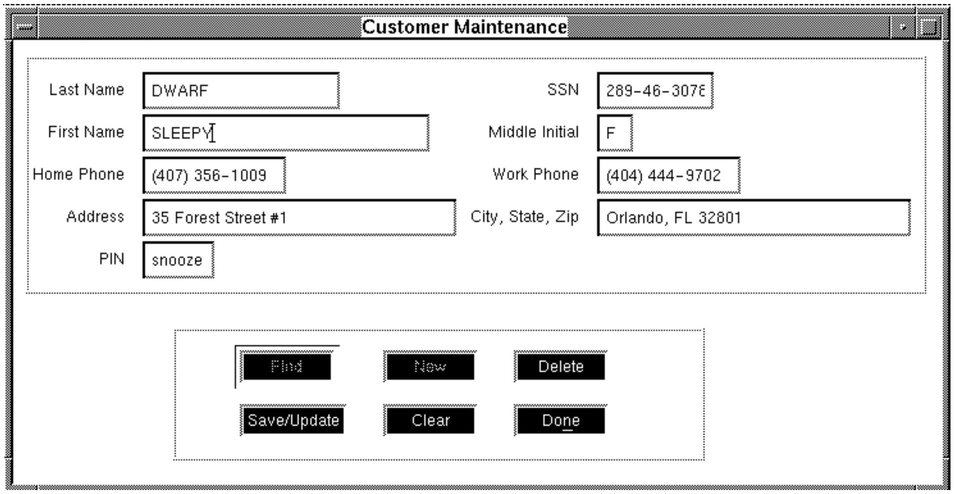
Several buttons are available to take action after editing a customer's data: Save/Update to commit the change, Clear to clear all data, and Delete to remove a customer. Choose Done to dismiss the screen.

New Customer Mailings

Enterprise Bank includes a feature to do a generic bank information mailing to all new customers. This process uses Oracle Tuxedo message queuing. Message queuing allows the queuing up of non-time-critical tasks that are more efficiently processed in batch mode. When Customer Maintenance is used to add new customers, customer data is sent to the `NEW_CUST_Q` message queue for processing at a later time, when the mailing documents would be produced.

To process all new customers for mailing, choose Customers→New Customer Mailings to open the New Customer Mailing List screen. You have the option of selecting specific customers from those that are queued up, or of choosing to send the mailing to all new accounts in the queue. Choose Retrieve List to see a listing of all customers currently in the queue.

For more information on message queuing, and more detail on the coding required to use the `NEW_CUST_Q` queue, refer to page 8-15. Comprehensive information on Oracle Tuxedo System/Q message queuing is in your Oracle Tuxedo documentation.


Customer Maintenance

Last Name	DWARF	SSN	289-46-3078
First Name	SLEEPY	Middle Initial	F
Home Phone	(407) 356-1009	Work Phone	(404) 444-9702
Address	35 Forest Street #1	City, State, Zip	Orlando, FL 32801
PIN	snooze		

Figure C-15 Customer Maintenance screen.

The Enterprise Bank Administrator Client

The Bank Administrator client permits you to access all employee features plus bank personnel data, as well as providing the ability to broadcast messages to all clients logged on to the system.

Starting the Bank Administrator Client

To start the administrator client, start Panther and give the top-level employee/administrator Enterprise Bank screen as the command-line parameter:

```
prodev branch
```

In Windows, double-click on the EntBank Employee WS icon.

When Enterprise Bank is started, the Employee Workstation welcome screen is displayed.

To logon to the employee client as an administrator requires a valid administrator last name and password. There is no separate table in the database for administrators, they can be found in the employee table and have their Administration Privileges field set to "Y." Try last name `CAPONE` and password `chicago`. Once you have correctly entered the login data, choose the Start button to initiate administrator services.

Administrator Services

The employee and administrator clients are both menu-driven. The actions that an administrator can take are available from the Enterprise Bank menu bar. The options present on the menu bar give you access to all administrator features.

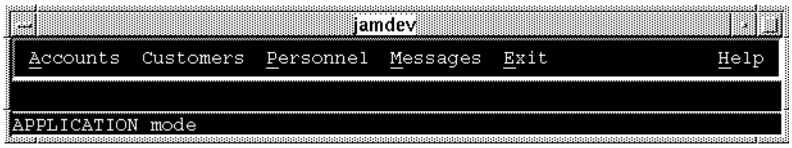


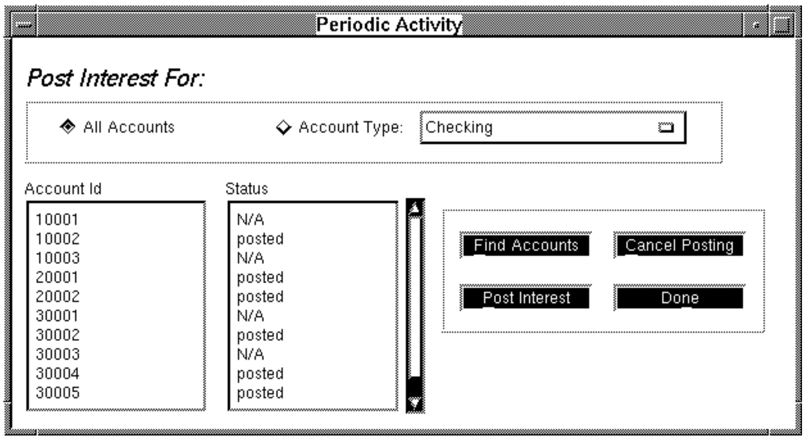
Figure C-16 Administrator services are available from the pulldown menus on the menu bar.

Accounts Menu Option

The Accounts menu option contains four selections on its pulldown menu: Account List, Account Maintenance, Post Interest, and Account Type Settings. The Account List and Account Maintenance have already been described for the Employee client.

Post Interest to Accounts

Choose Accounts→Post Interest to open the Periodic Activity screen, from which an administrator can post interest to any of the bank accounts. The administrator chooses between All Accounts or Account Type radio buttons to specify which accounts are to have interest posted to them. When choosing Account Type, the administrator selects the account type from a pulldown menu that contains the three bank account types: Checking, savings, and Money Market.



Account Id	Status
10001	N/A
10002	posted
10003	N/A
20001	posted
20002	posted
30001	N/A
30002	posted
30003	N/A
30004	posted
30005	posted

Figure C-17 Post interest to any interest bearing account on the Periodic Activity screen.

Once the account type choice is made, choosing Find Accounts fetches the accounts that pertain to the search criteria and displays them in the scrolling window that lists the accounts by Id number.

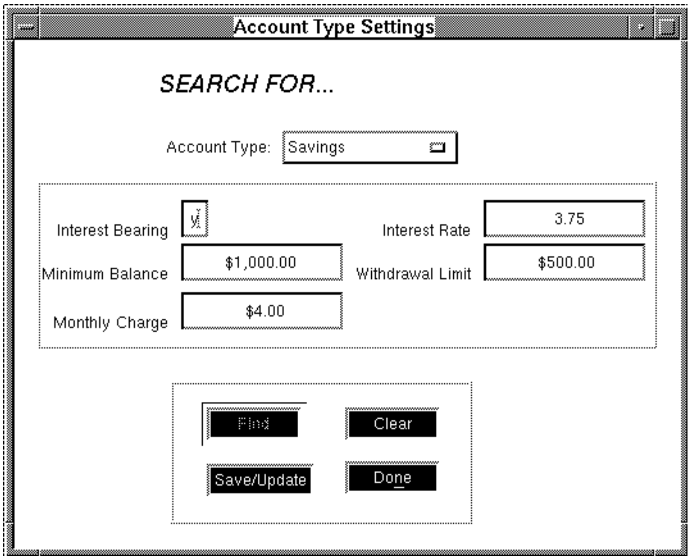
To post interest for the accounts selected, choose the Post Interest button. The status for each account is posted in the second scrolling list. To cancel posting to accounts, choose the Cancel Posting button. Choose Done to dismiss the screen.

Modify an Account Type

Choose Accounts→Account Type settings to raise the Account Type setting screen. This screen is used by an administrator to view or change account type features. An Account Type pulldown menu permits the administrator to choose which account type to edit. This screen contains the following data entry fields:

- An Interest Bearing field that can be toggled to Y or N that will determine whether or not the account type will generate interest.
- A Minimum Balance filed that permits the administrator to set the minimum balance requirement for the account.
- An Interest Rate field to set the interest rate for the account type.

- A Withdrawal Limit field.



Account Type: Savings

Interest Bearing: y
Interest Rate: 3.75
Minimum Balance: $1,000.00
Withdrawal Limit: $500.00
Monthly Charge: $4.00

Figure C-18 An administrator can update account parameters such as interest rate on the Account Type Settings screen.

Several push buttons are available to take action after editing. After an account type had been selected and the settings possibly changed, the administrator can choose Save/Update to commit the change, cancel the edit by choosing Clear, or dismiss the screen by choosing Done.

Customers Menu Option

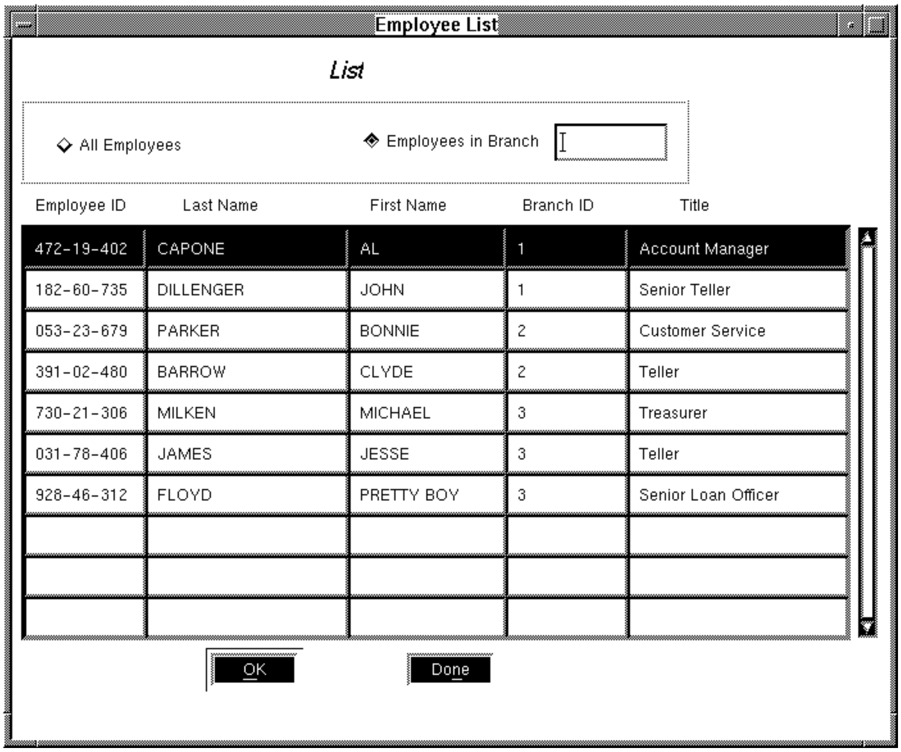
The Customers menu option contains the same two selections as it does on the Employee client: Customer List and Customer Maintenance, which have already been described above.

Personnel Menu Option

The Personnel menu option provides two choices: Employee List and Employee Maintenance.

Display a List of Employees

Choose Personnel→Employee List to open the Employee List screen. This screen provides a list of all bank employees. The administrator can choose to either view all employees or just employees specific to a branch, by way of two radio buttons and text entry field for branch Id. Once the choice has been made and the OK button has been chosen, the following data pertaining to each employee is displayed within scrolling windows: employee id, name, branch id, and title. The information on this screen is for viewing purposes only. Choose Done to dismiss the screen when finished.

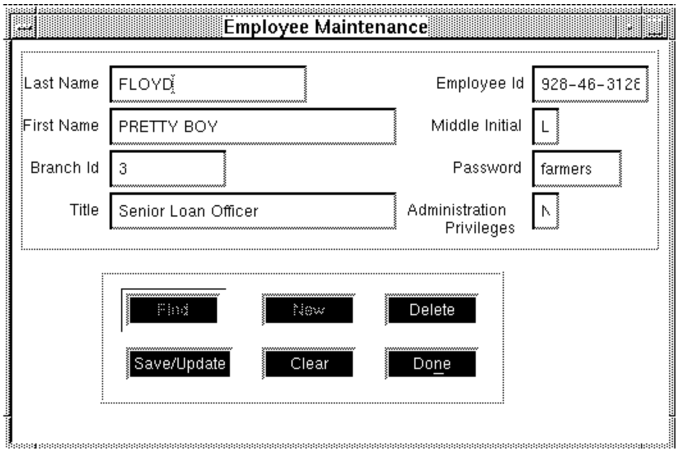


Employee ID	Last Name	First Name	Branch ID	Title
472-19-402	CAPONE	AL	1	Account Manager
182-60-735	DILLENGER	JOHN	1	Senior Teller
053-23-679	PARKER	BONNIE	2	Customer Service
391-02-480	BARROW	CLYDE	2	Teller
730-21-306	MILKEN	MICHAEL	3	Treasurer
031-78-406	JAMES	JESSE	3	Teller
928-46-312	FLOYD	PRETTY BOY	3	Senior Loan Officer

Figure C-19 An administrator can lookup an employee on the Employee List screen.

Add/Update or Delete an Employee

Choose Personnel→Employee Maintenance to open the Employee Maintenance screen. This screen is used to make changes to an employee's data, add a new employee, or delete an employee. The screen has the following data entry fields: Last Name, First Name, Employee Id, Middle Initial, Branch Id, Password, Title, and Administration Privileges (an employee can have administrator privileges when this field is set to "Y"; otherwise it is set to "N").



Last Name FLOYD
First Name PRETTY BOY
Branch Id 3
Title Senior Loan Officer
Employee Id 928-46-3128
Middle Initial L
Password farmers
Administration Privileges N

Figure C-20 Employee Maintenance screen permits an administrator to update personnel files.

To add a new employee, choose New, enter the employee's information, then choose Save/Update.

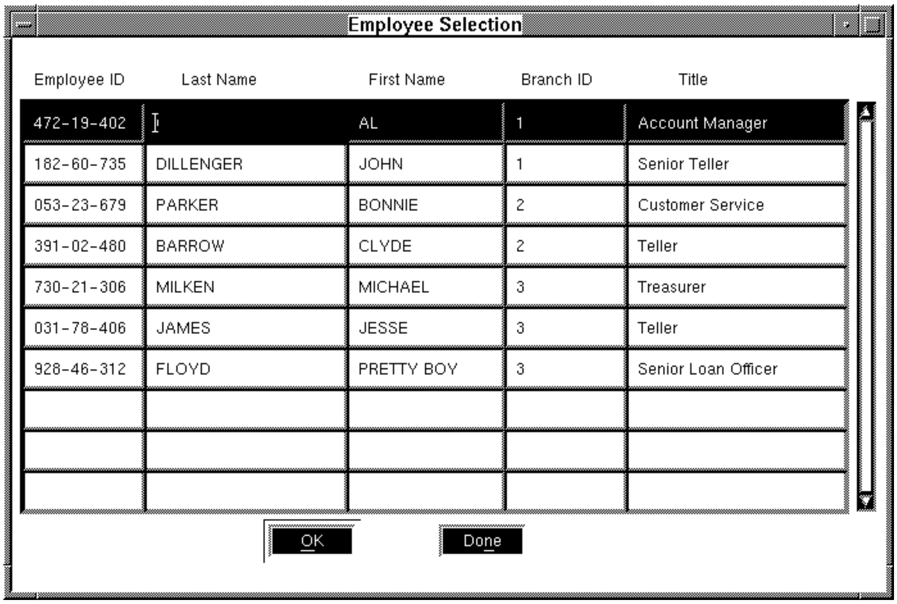
To retrieve an existing employee's data, enter the employee's name or Employee Id. This screen employs a pattern searching mechanism that permits you to enter any character[s], including the wildcard ("%") character, in any of these identification fields. Choose the Find button after the pattern search data has been entered. If more than one employee satisfies the criteria, the Employee Selection screen will open from which an individual employee can be selected (see below). If only one employee matches, that employee's data will appear in the Employee Maintenance screen. Once the employee's data is in the screen it can be edited.

Several buttons are available to take action after editing: choose the Save/Update button to commit the change, cancel the edit by choosing the Clear button, or remove the employee by choosing the Delete button. When editing is finished, dismiss the window by choosing Done.

Select an Employee for Maintenance

The Employee Selection screen is used to select an employee's data for updating when the specific identifying information is not at hand. It is raised by entering pattern search data in any of the identification fields on the Employee Maintenance screen.

This screen displays the following information within scrolling lists: employee id, name, branch id, and title. To select an employee, select the corresponding Employee Id in the scrolling window, then choose the OK button. Alternatively, dismiss the screen by choosing Done.



Employee ID	Last Name	First Name	Branch ID	Title
472-19-402		AL	1	Account Manager
182-60-735	DILLENGER	JOHN	1	Senior Teller
053-23-679	PARKER	BONNIE	2	Customer Service
391-02-480	BARROW	CLYDE	2	Teller
730-21-306	MILKEN	MICHAEL	3	Treasurer
031-78-406	JAMES	JESSE	3	Teller
928-46-312	FLOYD	PRETTY BOY	3	Senior Loan Officer

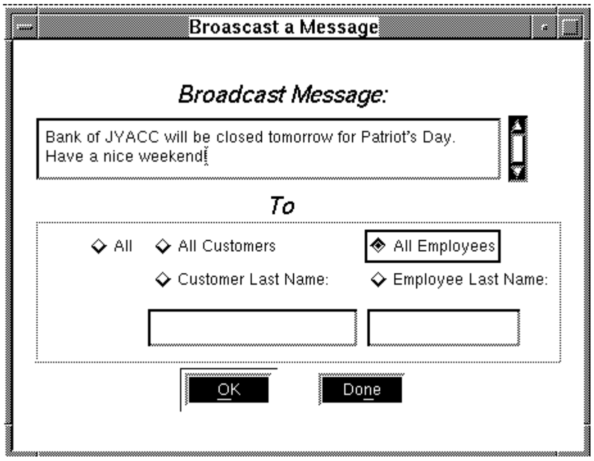
Figure C-21 Employee Selection screen permits an administrator to select an employee's data for updating in the Employee Maintenance screen.

Broadcasting a Message

Administrators have the ability to broadcast a message to any clients who are using the system. Choosing Messages from the menu bar raises the Broadcast screen.

Broadcast a Message

Five radio buttons permit the administrator to choose whom to broadcast to: all clients, all customers, one customer, all employees, or one employee. If one customer or one employee is desired, the Customer or Employer Last Name button is selected, and the last name is entered into the respective text widget. The message to be broadcast is typed into the Message text widget. To send the message, choose the OK button. Choose Done to dismiss the screen.



Bank of JYACC will be closed tomorrow for Patriot's Day.
Have a nice weekend!

Figure C-22 An administrator can broadcast a message to any clients logged onto the system with the Broadcast a Message screen.

Designing Enterprise Bank

The development of Enterprise Bank proceeded with the creation of the `entbank` database. After that, development followed a top-down approach:

- Creation of the client screens.
- Writing of the services, in JPL.
- Creation of the service screens.

`JAMFLEX` buffers are used for data transport in all services except those that make use of the extended Oracle Tuxedo features—message queuing and event brokering. The customer mailings feature uses `FML` buffers, and the withdrawal limit exceeded and security violation alert features use `STRING` buffers.

During development of Enterprise Bank, the service screens associated with the services are opened and closed with each request to a service. This is done within the pre_ and post_service handlers defined in the server initialization JPL. The handlers are set (via the `hdl_pre_service` and `hdl_post_service` application properties) to `pre_service` and `post_service` respectively. These handlers make calls to the default development handlers `sm_tp_pre_service_winopen` and `sm_tp_post_service_winclose` respectively.

Here is the handler code from the server initialization JPL:

```
proc pre_service( callid, service_name, container_name )
{
   if (server_logging)
       log "===> Starting service ':service_name' \
          in container ':container_name'"
   return sm_tp_pre_service_winopen( callid, \
          service_name, container_name )
}


proc post_service( callid, service_name, container_name )
{
   if (server_logging)
       log "===> Ending service ':service_name' \
          in container ':container_name'"
```

```
    return sm_tp_post_service_winclose( callid, \
            service_name, container_name )
}
```

For a real application, in production mode, the pre_ and post_service handlers would be replaced with `sm_tp_pre_service_winopen_or_select` and `sm_tp_post_service_winclose_or_deselect`, to minimize unnecessary overhead of opening and closing service components. When the service component should be opened depends on the value of the Cache Service Component attribute specified for the service in the JIF. Depending on the value, the opening and closing of the service component can occur when the service is first advertised, the first time it is called, or each time it is called. For information on pre_ and post_service default handler behavior, refer to page 6-27.

The following code shows how this is implemented; it is excerpted from the server initialization JPL module:

```
// For production style environment (when using runtime
// Panther executables instead of development
// executables), set this to 1
global production_env = "0"
...

proc dbms_init()
{
    ...
    if (production_env)
    {

        // for production environment, service components
        // are opened when services are advertised, closed
        // when services are unadvertised, and
        // selected/deselected with each request

        @jam->hdl_pre_service = \
            "sm_tp_pre_service_winselect"
        @jam->hdl_post_service = \
            "sm_tp_post_service_windeselect"
        @jam->hdl_advertise = \
            "sm_tp_advertise_winopen"
        @jam->hdl_unadvertise = \
            "sm_tp_unadvertise_winclose"
    }
    else
    {

        // for development environment, service components
```

```
            // are opened and closed with each request to
            // a service

            @jam->hdl_pre_service = "pre_service"
            @jam->hdl_post_service = "post_service"
        }
        @jam->hdl_jif_changed = "jif_changed"
        @jam->hdl_exception = "exc_hand"
        @jam->hdl_server_exit = "server_exit"

        return 0
    }
```

APPENDIX

D JetNet/Oracle Tuxedo Exception Event Types

Each exception event type that is generated by the middleware adapter can be identified by one of the constants shown in the following table and its corresponding integer code. These constants are accessible in JPL and C functions; they are also stored in the application variable `tp_exc_names` (stripped of the `TP_` prefix), and are indexed according to the corresponding integer codes.

Table D-1 Exception event type constants and integer codes

Exception type constant	Code	Description
`TP_ALREADY_CANCELLED`	1	Attempting to cancel a request that has already been cancelled
`TP_BEGIN_FAILED`	2	Unable to begin a new transaction
`TP_COMMIT_FAILED`	3	Unable to commit a transaction
`TP_COMMIT_PARTIAL`	4	Transaction has (or may have been) partially rolled back
`TP_COMMIT_ROLLEDBACK`	5	Unable to commit a transaction because it has already been rolled back
`TP_CONNECTION_CLOSE_FAILED`	6	Unable to terminate a connection to the middleware

Table D-1 Exception event type constants and integer codes *(Continued)*

Exception type constant	Code	Description
`TP_CONNECTION_LIMIT`	7	The connection limit for the middleware session has been exceeded
`TP_CONNECTION_OPEN_FAILED`	8	Unable to initiate a connection to the middleware
`TP_DATAFUNC_FAILED`	9	Failure reported from `DATAFUNC` function
`TP_EVTBROKER_ACCESS_FAILED`	10	Unable to access event broker server
`TP_EXPLICIT_CANCEL`	11	A service request has been cancelled by the `service_cancel` command
`TP_GROUP_NOT_IN_JIF`	12	The service group has not been defined in the JIF
`TP_HANDLER_MISSING`	13	An invoked handler cannot be located
`TP_IDENTIFIER_TRUNCATED`	14	An identifier has been truncated
`TP_INTERNAL_ERROR`	15	Internal error
`TP_INVALID_ARGUMENT`	16	Name, syntax, or use of an argument is invalid
`TP_INVALID_ARGUMENT_COMPONENT`	17	Component of an argument is invalid
`TP_INVALID_ARGUMENT_LIST`	18	Argument list is invalid
`TP_INVALID_BUFFER`	19	Data buffer received from a client or a service is of the wrong type as specified in the JIF; or a client has received an unsolicited message that is of a type not supported
`TP_INVALID_BUFFER_VERSION`	20	Received data buffer from a client or a service is of an incompatible version; or a client has received an unsolicited message that is of an incompatible version
`TP_INVALID_CALL`	21	Service call does not exist
`TP_INVALID_CLIENT_COMMAND`	22	Function or JPL command is only available to servers, not clients
`TP_INVALID_CLIENT_OPTION`	23	Function or JPL command option is only available to servers, not clients
`TP_INVALID_COMMAND`	24	Function or JPL command is invalid

Table D-1 Exception event type constants and integer codes *(Continued)*

Exception type constant	Code	Description
`TP_INVALID_COMMAND_SYNTAX`	25	Function or JPL command syntax is invalid
`TP_INVALID_CONNECTION`	26	Specified connection does not exist
`TP_INVALID_CONTEXT`	27	Attempt to perform action out of context
`TP_INVALID_FORWARD`	28	A conversational service cannot be forwarded
`TP_INVALID_VARIABLE_REF`	29	Unable to resolve reference to the Panther variable
`TP_INVALID_MONITOR_COMMAND`	30	Function or JPL command is not available for the middleware adapter
`TP_INVALID_MONITOR_OPTION`	31	Function or JPL command option is not available for the middleware adapter
`TP_INVALID_OPTION`	32	Option is invalid in this context
`TP_INVALID_OPTION_VALUE`	33	Value for option is invalid
`TP_INVALID_SERVER_COMMAND`	34	Function or JPL command is available only to clients, not servers
`TP_INVALID_SERVER_OPTION`	35	Function or JPL command option is available only to clients, not servers
`TP_INVALID_SERVICE`	36	Service is invalid
`TP_INVALID_TRANSACTION`	37	Transaction does not exist
`TP_JIF_ACCESS_FAILED`	38	JIF or the Panther library containing it could not be accessed
`TP_JIF_LOWER_VERSION`	39	New JIF has lower version than the current one
`TP_LOGFILE_ERROR`	40	Unable to write to the ULOG file
`TP_MONITOR_ERROR`	41	Error reported from middleware adapter
`TP_NONTRANSACTIONAL_ACTION`	42	Requested action cannot be performed within a transaction
`TP_NONTRANSACTIONAL_SERVICE`	43	The service cannot be executed within a transaction

Table D-1 Exception event type constants and integer codes *(Continued)*

Exception type constant	Code	Description
`TP_NO_OUTSIDE_TRANSACTION`	44	Option `OUTSIDE_TRANSACTION` is ignored because no transaction exists
`TP_NO_OUTSTANDING_CALLS`	45	The specified requests are no longer outstanding
`TP_NO_OUTSTANDING_MESSAGE`	46	There are no outstanding unsolicited messages
`TP_NO_SERVICES_ADVERTISED`	47	No services advertised or unadvertised
`TP_NO_SIGNALS`	48	Client is not capable of signal-based notification
`TP_OUT_OF_MEMORY`	49	Unable to allocate sufficient memory; program will exit
`TP_PERMISSION_DENIED`	50	Unable to perform action because permission has been denied
`TP_POSTING_FAILED`	51	Posting a transactional event to either a service or to a storage queue failed
`TP_QUEUE_BAD_MSGID`	52	Invalid message identifier
`TP_QUEUE_BAD_NAMESPACE`	53	Invalid resource manager identifier
`TP_QUEUE_BAD_QUEUE`	54	Invalid or deleted queue name
`TP_QUEUE_CANT_START_TRAN`	55	Error starting separate transaction for queuing operation
`TP_QUEUE_FULL`	56	No space left on queue for any additional messages
`TP_QUEUE_MSG_IN_USE`	57	Selected message (or all messages) is in use by another transaction
`TP_QUEUE_NO_MSG`	58	No message was available for dequeuing
`TP_QUEUE_NOT_IN_QSPACE`	59	Unable to find the specified queue in the specified queue space in the JIF.
`TP_QUEUE_RSRC_NOT_OPEN`	60	Resource manager is not currently open
`TP_QUEUE_SPACE_NOT_IN_JIF`	61	Unable to find specified queue space in JIF.
`TP_QUEUE_TRAN_ABORTED`	62	Transaction enclosing queuing operation was aborted

Table D-1 Exception event type constants and integer codes *(Continued)*

Exception type constant	Code	Description
`TP_QUEUE_TRAN_ABSENT`	63	Queuing operation was done when transaction state was not active
`TP_QUEUE_UNEXPECTED`	64	Undocumented queuing error produced by monitor
`TP_REQUEST_LIMIT`	65	The limit on the number of outstanding requests has been exceeded
`TP_ROLLBACK_COMMITTED`	66	Unable to roll back the transaction because it has already been committed
`TP_ROLLBACK_FAILED`	67	Unable to roll back the transaction
`TP_SERVICE_FAILED`	68	Service returned a failure status
`TP_SERVICE_NOT_IN_JIF`	69	Service could not be found in the JIF
`TP_SERVICE_PROTOCOL_ERROR`	70	Service has violated protocol and has been abnormally terminated
`TP_SUBSCRIPTION_LIMIT`	71	Maximum number of subscriptions has been reached
`TP_SUBSCRIPTION_MATCH`	72	Subscription matches one already listed with event broker
`TP_SVCROUTINE_MISSING`	73	Unable to locate service routine
`TP_SVC_ADVERTISE_LIMIT`	74	The limit on the number of advertised services has been exceeded
`TP_SVC_WORK_OUTSTANDING`	75	There is work which this service has begun that has not completed
`TP_SVRINIT_WORK_OUTSTANDING`	76	Server init routine has begun work that has not yet completed
`TP_TIMEOUT`	77	Action terminated due to timeout condition
`TP_TRANSACTION_LIMIT`	78	A new transaction would exceed transaction limit for the middleware session
`TP_UNLOAD_FAILED`	79	Failure reported from unload event handler
`TP_UNSUPPORTED_BUFFER`	80	Specified buffer type is not supported

Table D-1 Exception event type constants and integer codes *(Continued)*

Exception type constant	Code	Description
`TP_USER_ABORT`	81	Action has been explicitly aborted
`TP_WORK_OUTSTANDING`	82	Work is still being performed within this transaction
`TP_XA_CLOSE_FAILED`	83	Unable to close XA-connection resource managers
`TP_XA_OPEN_FAILED`	84	Unable to open XA-connection resource managers

APPENDIX

E Application Setup Checklist

Once the software is installed, the following steps provide a basic Panther application server for a JetNet/Oracle Tuxedo application.

Setting Up the Application Server

Populate the Application Directory

Unix Environment

On a UNIX application server, create an application directory containing:

`setup.sh`	A setup file with the location of the Panther software installation, the license file, and the middleware configuration file at your site. For the default setup file, copy setup.sh from the config directory of your Panther installation.
`client.lib`, `server.lib` and `common.lib`	Three standard application libraries. Copies of these libraries are in the `samples/newapp` directory of your Panther server installation.

machine.env, proserv.env and progserv.env (if using progserv)	Three standard environment files: machine.env for the machine settings, proserv.env for the standard server, and progserv.env for the conversion server (only used with applications converted from two-tier). Copies of these files are in the samples/newapp directory of your Panther server installation.
devserv, proserv, prodserv (optional), and progserv (optional)	Symbolic links to, or copies of, the server executables: devserv for the development access server and proserv for the standard server. If needed, create links or copies of progserv for the conversion server and prodserv for the server with debugger available services. The server executables are located in the util directory of your Panther installation.
broker.bin	The middleware configuration file. To create a middleware configuration file, refer to page 3-2.

Windows Environment

On a Windows application server, create an application folder containing:

client.lib, server.lib and common.lib	Three standard application libraries. Copies of these libraries are in the samples\newapp directory of your Panther server installation.
machine.env, proserv.env and progserv.env	Three standard environment files: machine.env for the machine settings, proserv.env for the standard server, and progserv.env for the conversion server (only used with applications converted from two-tier). Copies of these files are in the samples\newapp directory of your Panther server installation.
devserv.exe, proserv.exe, prodserv.exe and progserv.exe	Copies of the server executables: devserv.exe for the development access server and proserv.exe for the standard server. If needed, copy progserv.exe for the conversion server and prodserv.exe for the server with debugger available services. The server executables are located in the util directory of your Panther installation.

Configure the Middleware

Create a Configuration File

The middleware configuration file determines the machines and application servers needed for the application. To create a middleware configuration file:

❑	Configure the environment. In Windows, check the settings of `jetman32.ini`. In UNIX, run the application's version of `setup.sh`.
❑	Start JetMan.
❑	Choose File→New→Application.
❑	On the Application Configuration window, enter the application name, and choose Next.
❑	Check the settings for the machine type, the Panther installation, the application directory, the middleware configuration file, and the machine environment variable file.
❑	For remote client access, choose Networking. On the Networking window, select Workstation Listener, and choose OK.
❑	Choose Done, and wait for the configuration file to be completed.

Configure Each Server

Once the configuration file is created and JetMan displays the application:

❑	Expand the application by double-clicking on the application name or choosing View→Expand Subtree.
❑	With the machine highlighted, choose File→New→Server for each type of server to add to the application. A minimum setting would have one standard server (`proserv`) and one file access server (`devserv`) for each ma chine.

	For a standard server (`proserv`):	
	❑	Enter the name: `ProservMyApp`.
	❑	Under Server Type, select Standard.
	❑	Choose Options.
	❑	Under Auto Advertised Services, choose All.
	❑	If using remote reports, choose Report.
	❑	If using service aliasing to test services, enter the Server Alias User Name.
	❑	Under Database Connect String, enter the command needed to connect to the database.
	❑	Under Init Routine, enter the function to be called on initialization of the server.
	❑	When Standard Server Details is complete, choose OK.
	❑	When Server Configuration is complete, choose OK.
	For a file access server (`devserv`):	
	❑	Enter the name: `DevservMyApp`.
	❑	Under Server Type, select File Access.
	❑	When Server Configuration is complete, choose OK.

Applications which are running remote reports must have a file access server on the same machine as the standard server in order to access and distribute the report files.

Start the Application Server

To start the application server in JetMan, highlight the application and choose Edit→Activate. An alternative is to use the command line utility `rbboot`. The Status window shows the messages for each server process.

Stop the Application Server

To stop the application server in JetMan, choose Edit→Deactivate or, if clients are connected, Edit→Forcibly Deactivate. An alternative is to use the command line utility `rbshutdown`.

Setting Up the Workstation Client

Workstation (or remote) clients set `SMRBPORT` and `SMRBHOST` in order to access the remote application server. For Windows, these settings are stored in `prol5w32.ini or prol5w64.ini`.

Native (or local) clients set `SMRBCONFIG` in order to access the middleware configuration file on the same host machine.

APPENDIX

F Deployment Checklist for JetNet

Directory Structure for JetNet Applications

Distribute the files and libraries used by your Panther application in a single directory, call it the *application directory*. The directory should include such things as your application's executables and Panther-specific libraries. In addition, it should include the following subdirectories:

- bin directory for UNIX only—Includes JetNet administrative executables. Required for three-tier processing.
- configuration directory—Includes the runtime components that make up your application, such as your application libraries and those files that are specific to running your application.
- library directory for UNIX only—Includes JetNet shared libraries. Required for three-tier processing.
- locale directory—Includes the routines used by JetNet. Required for three-tier processing.
- udataobj directory—Includes files used by JetNet. Required for three-tier processing.

Checklist for Deployment

The tables in this section list the components you should include in a distribution for the specific platform. Depending on your particular application, there might be other considerations and files which you might include. Those considerations are covered later in this chapter.

Preparing a Windows Distribution

Table F-1 lists the files and libraries required on a Windows installation. The table also includes where these files can be found in the Panther distribution. In general, you or should make copies of those files as opposed to using the originals. In all likelihood, your Panther application has been using the components it needs while you've been developing it. This list will serve as a means of making certain all the pieces you need are deployed to the application users.

Table F-1 Checklist for contents of Panther Windows applications

File/Library	Found in Panther	Description
application directory contents:		
`cktbl32/64.dll`	`util`	Panther-specific DLL
database DLLs	`util`	Support Panther database drivers—Informix, ODBC, Oracle, Sybase
`*.ini`	`config`	Initialization files. `prol5w32/64.ini`, `jetman.ini` if using the JetNet manager.
`jetman.exe`	`util`	JetNet manager executable; required only if running the manager
`libsti32/64.dll`	`util`	Panther-specific DLL
`libsti.ini`	`config`	Graph-specific initialization file (copy this file to the Windows directory)

Table F-1 Checklist for contents of Panther Windows applications *(Continued)*

File/Library	Found in Panther	Description
`libxml2.dll`	`util`	Needed if XML files are to be imported.
`msvcr80.dll`	`util`	Microsoft Visual C++ 2005 runtime DLL. Needed if the Redistributable Package will not be installed.
`PanPDF32/64.dll`	`util`	Needed if PDF reports will be created.
`projpeg.dll`	`util`	Needed to process JPEG images.
`promfc32/64.dll`	`util`	Contains status line and frameset code.
`prores32/64.dll`	`util`	Panther Windows resource DLL
`prorun32/64.exe`	`util`	Runtime executable (rename for your application)
`rwres32/64.dll`	`util`	Report Writer Windows resource DLL
`wbuft.dll`	`util`	JetNet library
`wtuxws.dll`	`util`	JetNet library
config directory contents:		
`client.lib` includes:		
`client screens`		Panther screens that make up the user interface
`smwzmenu`		Binary menu script file; include if client screens created with screen wizard use the prototype menu bar/toolbar
`smwizard.bin`		JPL module made public by client screens created by the screen wizard
`JPL modules`		JPL code used by client screens
Graphics files		Image files (such as *.ico, *.bmp, *.jpg) referenced on client screens and/or toolbars
`styles.sty`		Transaction manager styles file for your application
`common.lib includes:`		
`wincmap.bin`	`config`	Binary configuration map file (maps Panther fonts to Windows-specific fonts, etc.).

Table F-1 Checklist for contents of Panther Windows applications *(Continued)*

File/Library	Found in Panther	Description
`jif.bin`		Binary service and queue definition file
`winkeys.bin`	`config`	Binary key files for mapping physical keys to Panther logical keys. Omit this file from the library if end-users can modify key mapping on installation.
`msgfile.bin`	`config`	Contains messages and information used by Panther
`*.fnt`	`config`	Graph-specific fonts referenced in graphs in your application
`grafcap`	`config`	Initialization file for graph support
`prorun5.lib`	`config`	Panther's runtime support library
`prorw5.lib`	`config`	Panther's runtime library for reports
`winkeys`	`config`	ASCII key file for mapping physical keys to Panther logical keys. Required if key mapping is user configurable; include `key2bin` utility as well.
`smvars.bin`	`config`	Binary environment setup file
`symbold, symbols1`	`config`	Font files for graphs, if used in application
locale\C directory contents:		
		Routines used by JetNet
`gp_cat`	`locale\C`	
`langinfo`	`locale\C`	
`libwsc_cat`	`locale\C`	
`trpc_cat`	`locale\C`	
udataobj directory contents:		
`tpadm`	`udataobj`	JetNet-specific file
`usysfl32`	`udataobj`	JetNet-specific file

Table F-1 Checklist for contents of Panther Windows applications *(Continued)*

File/Library	Found in Panther	Description
`usysflds`	`udataobj`	JetNet-specific file

Preparing a UNIX Distribution

Table F-2 Checklist for contents of Panther UNIX/Motif applications

File/Library	Found in Panther	Description
application directory contents:		
`broker.bin`		JetNet configuration file
`progserv`	`util`	Conversion server executable (rename for your application); required only for 2- to 3-tier converted applications
`progserv.env`	`config`	Conversion server environment definition file; required only for 2- to 3-tier converted applications
`Prolifics`	`config`	Resource file for Motif (installation should copy Prolifics to the home directory of each user)
`prorun`	`util`	Client executable (rename for your application); required only if supporting UNIX clients
`proserv`	`util`	Standard server executable (rename for your application)
`proserv.env`	`config`	Standard server environment definition file
`rbboot`	`util`	Utility to start JetNet and boot application servers
`rbcfinfo`	`util`	Used by `rbconfig`
`rbconfig`	`util`	Command-line utility for creating a JetNet configuration file
`rblisten`	`util`	Utility allows application servers to run on multiple ma chines.
`rbshutdown`	`util`	Utility to shutdown JetNet and application servers

Table F-2 Checklist for contents of Panther UNIX/Motif applications ***(Continued)***

File/Library	Found in Panther	Description
bin directory contents:		
		JetNet administrative executables
config directory contents:		
`client.lib` includes:		Required only if supporting UNIX clients
client screens		Panther screens that make up user interface
`smwzmenu`		Binary menu script file; include if client screens created with screen wizard use the prototype menu bar/toolbar
`smwizard.bin`		JPL module made public by client screens created by the screen wizard
JPL files		JPL files used by client screens
Graphics files		Image files (e.g., *.xbm, *.xpm, *.bmp, *.jpg) referenced on client screens and/or toolbars
`styles.sty`		Transaction manager styles file
`common.lib` includes:		
`*cmap.bin`	`config`	Binary configuration map file (maps Panther fonts to Motif-specific fonts, etc.).
`jif.bin`		Binary service and queue definition file
`*key.bin`	`config`	Binary key files for mapping physical keys to Panther logical keys. Omit this file from the library if end-users can modify key mapping on installation.
`msgfile.bin`	`config`	Contains messages and information used by Panther
`*vid.bin`	`config`	For character-mode only. Binary files that describe terminal capabilities and attributes to Panther. Omit this file from the library if end-users can modify video specifications on installation.
`*.fnt`	`config`	Graph-specific fonts referenced in graphs in your application; required if supporting UNIX clients

Table F-2 Checklist for contents of Panther UNIX/Motif applications *(Continued)*

File/Library	Found in Panther	Description
`gdsp`	`util`	Graph support utility
`grafcap`	`config`	Initialization file for graph support
`prorun5.lib`	`config`	Panther's runtime support library
`prorw5.lib`	`config`	Panther's runtime library for reports
`server.lib` includes:		Not required for two-tier applications
service components		Screens that are defined as service components for use on a server
`smwizsrv.bin`	`config`	JPL module made public by service components created by the screen wizard
JPL modules		JPL code used by service components
`smvars.bin`	`config`	Binary environment setup file. Copy and modify for your application.
`swsdrvr`	`util`	Graph support utility
`library directory contents:`		JetNet shared libraries and platform-specific shared libraries. Platform-specific libraries have a unique extension; refer to your platform documentation to determine the extension. For example, for Solaris, the file extension is .so and for HPUX on PA-RISC, it is .sl.
`libbuft.pltExt`	`lib`	
`libfml.pltExt`	`lib`	
`libfml32.pltExt`	`lib`	
`libgp.pltExt`	`lib`	
`libnwi.pltExt`	`lib`	
`libnws.pltExt`	`lib`	
`libqm.pltExt`	`lib`	
`libtmib.pltExt`	`lib`	

Table F-2 Checklist for contents of Panther UNIX/Motif applications ***(Continued)***

File/Library	Found in Panther	Description
libtux.pltExt	lib	
libtux2.pltExt	lib	
libusort.pltExt	lib	
libwsc.pltExt	lib	
libwsh.pltExt	lib	
locale/C directory contents:		
		Routines used by JetNet
CMDTUX_CAT	locale/C	
GP_CAT	locale/C	
LIBTMIB_CAT	locale/C	
LIBTUX_CAT	locale/C	
TMADMIN_CAT	locale/C	
WSNAT_CAT	locale/C	
udataobj directory contents:		
mib_views.V	udataobj	JetNet-specific file
tmib_views.V	udataobj	JetNet-specific file
tpadm	udataobj	JetNet-specific file

Index