

Building Distributed Applications Using the Progress AppServer

© 2001 Progress Software Corporation. All rights reserved.

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

Progress, Progress Results, Provision and WebSpeed are registered trademarks of Progress Software Corporation in the United States and other countries. Apptivity, AppServer, ProVision Plus, SmartObjects, IntelliStream, and other Progress product names are trademarks of Progress Software Corporation.

SonicMQ is a trademark of Sonic Software Corporation in the United States and other countries.

Progress Software Corporation acknowledges the use of Raster Imaging Technology copyrighted by Snowbound Software 1993-1997 and the IBM XML Parser for Java Edition.

© IBM Corporation 1998-1999. All rights reserved. U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Progress is a registered trademark of Progress Software Corporation and is used by IBM Corporation in the mark Progress/400 under license. Progress/400 AND 400® are trademarks of IBM Corporation and are used by Progress Software Corporation under license.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Any other trademarks and/or service marks contained herein are the property of their respective owners.

.

May 2001



Product Code: 4502
Item Number: 81071;9.1C

Contents

Preface . . . . . **xiii**
Purpose xiii
Audience xiii
Organization of This Manual xiv
How to Use This Manual xv
Typographical Conventions xvi
Syntax Notation xvii
Progress Messages xxi
Other Useful Documentation xxiii
Getting Started xxiii
Development Tools xxiv
Reporting Tools xxv
4GL xxvi
Database xxvi
DataServers xxvii
SQL-89/Open Access xxvii
SQL-92 xxviii
Deployment xxviii
WebSpeed xxix
Reference xxix

1. Distributed Application Computing **1–1**
1.1 What Is Distributed Application Computing? 1–2
1.1.1 Application Partitioning 1–2
1.1.2 The N-tier Model 1–2
1.2 Comparing Computing Models 1–3
1.2.1 The Two-tier Model 1–3
1.2.2 Progress Distributed Application Computing Models 1–4

2. Overview of the Progress AppServer 2–1
2.1 Overview of the Architecture 2–2
2.1.1 Application Partitioning 2–2
2.1.2 Reasons to Partition 2–2
2.1.3 AppServer Components 2–4
2.1.4 How Does a Remote Procedure Execute? 2–4
2.1.5 The Concept of Separate Sessions 2–5
2.1.6 The Connection-based Model 2–8
2.1.7 Configurations 2–10
2.2 Run-time Components and Operation 2–16
2.2.1 Distribution of Components 2–19
2.2.2 Fault-tolerant NameServers 2–19
2.2.3 Fault-tolerant AppServers and Load Balancing 2–20
2.2.4 Operating Modes 2–21
2.2.5 AppServer Startup and Shutdown 2–22
2.3 Connection Process 2–23
2.3.1 Application Services 2–23
2.3.2 Connection Operation 2–24
2.4 Understanding AppServer Operating Modes 2–27
2.4.1 State-reset Operating Mode 2–27
2.4.2 State-aware Operating Mode 2–28
2.4.3 Stateless Operating Mode 2–30
2.5 Understanding Synchronous and Asynchronous Requests 2–32
2.5.1 Process Control Flow 2–33
2.5.2 Specifying the Type of Remote Request 2–34
2.5.3 Handling Asynchronous Request Results 2–34
2.5.4 Asynchronous Request Queuing 2–34

3. Administration 3–1
3.1 AppServer Administration Framework 3–2
3.1.1 AppServer Clients 3–4
3.1.2 Progress Explorer and Management Utilities 3–4
3.1.3 NameServers 3–6
3.1.4 Application Broker 3–6
3.1.5 Application Server Processes 3–7
3.1.6 AppServer and NameServer Properties File (ubroker.properties) 3–7
3.1.7 Text Editor and Configuration Utilities 3–7
3.1.8 Preparing for AppServer Administration 3–8

3.2 Configuring AppServer Components 3–8
3.2.1 Using the Progress Explorer to Configure an AppServer Instance 3–9
3.2.2 Configuring an AppServer Using the Progress Explorer 3–10
3.2.3 Configuring an AppServer Using the Properties File 3–13
3.3 Setting Up the Environment for AppServer Execution 3–17
3.3.1 Environment Variable Settings. 3–17
3.3.2 AppServer and NameServer Log Files 3–18
3.3.3 Progress Remote SmartDataObject Support. 3–18
3.3.4 Customizing the Application Server Process Executable 3–18
3.4 Starting and Managing an AppServer Instance 3–19
3.4.1 Requirements for Starting an AppServer. 3–20
3.4.2 Starting and Managing an AppServer with the Progress Explorer 3–20
3.4.3 Starting and Managing an AppServer with the Management Utilities 3–22
3.4.4 Checking AppServer Status 3–26
3.5 Specifying the Server Pool Parameters 3–26
3.5.1 Tuning for Operating Modes 3–27
3.5.2 Managing Application Server Processes 3–27
3.6 Managing Code Pages 3–28
3.6.1 Code-page Settings 3–28
3.6.2 Run-time Conversions 3–29
3.7 Summary of Management Tasks 3–30

4. Programming the AppServer 4–1
4.1 4GL for Programming AppServer Procedures 4–2
4.2 Accessing the Connection ID 4–4
4.3 Using AppServer Configuration Procedures 4–5
4.3.1 Startup and Shutdown Procedures 4–5
4.3.2 Connect and Disconnect Procedures 4–7
4.3.3 Activate and Deactivate Procedures 4–9
4.4 Controlling AppServer Entry Points 4–11
4.4.1 Export List Operation 4–12
4.4.2 Setting and Resetting the Export List. 4–12
4.4.3 Calling the EXPORT( ) Method 4–12
4.4.4 Resolving Remote Procedure Names 4–14
4.5 Managing Stateless Connection Context 4–15
4.5.1 Using the SERVER-CONNECTION-CONTEXT Attribute 4–15
4.5.2 Using the SERVER-CONNECTION-ID Attribute 4–16
4.5.3 Using the SERVER-CONNECTION-BOUND-REQUEST Attribute 4–16
4.5.4 Using the SERVER-CONNECTION-BOUND Attribute 4–18

4.6 Understanding Remote Procedure Handles 4–18
4.6.1 Remote Persistent Procedure Handles 4–18
4.6.2 Deleting Remote Persistent Procedures 4–19
4.7 Handling Conditions and Returning Values 4–19
4.7.1 Raising the ERROR Condition and Returning Values 4–19
4.7.2 Raising the ENDKEY Condition 4–20
4.7.3 Handling the STOP Condition 4–20
4.7.4 Handling the QUIT Condition 4–20
4.7.5 Effects of Unhandled Conditions 4–20
4.8 Managing Transactions 4–21
4.8.1 Types of AppServer Transactions 4–21
4.8.2 Implementing Normal 4GL Transactions 4–22
4.8.3 Implementing Automatic Transactions 4–23
4.9 Programming for Open Client Applications 4–29
4.9.1 Using the ProtoGen Utility 4–29
4.9.2 Using Code Editors 4–30

5. Programming the Client Application 5–1
5.1 4GL for Programming AppServer Client Procedures 5–2
5.2 Accessing AppServer Resources from a Client 5–8
5.3 Connecting an AppServer Instance 5–9
5.3.1 Creating a Server Object Handle 5–9
5.3.2 Establishing a Connection with the CONNECT( ) Method 5–10
5.3.3 Connection Example 5–12
5.4 Accessing the Connection ID on the Client 5–13
5.5 Running and Managing Remote Procedures 5–14
5.5.1 Running Remote Procedures 5–14
5.5.2 Understanding Proxy Persistent Procedure Handles 5–19
5.5.3 Deleting Remote Persistent Procedures 5–20
5.5.4 Remote Procedure Code Examples 5–20
5.6 Disconnecting from an AppServer Instance 5–24
5.6.1 Using the DISCONNECT( ) Method 5–24
5.6.2 Deleting the Server Handle 5–25
5.7 Handling Conditions and Return Values 5–26
5.7.1 Raising the STOP Condition 5–26
5.7.2 Handling Conditions on the RUN Statement 5–26
5.7.3 Using the RETURN-VALUE Function Across Sessions 5–27

5.8 Managing Asynchronous Requests 5–27
5.8.1 Executing an Asynchronous Request 5–28
5.8.2 Handling the Response from an Asynchronous Request 5–29
5.8.3 Cancelling Asynchronous Requests 5–34
5.8.4 Asynchronous Requests and Persistent Procedures 5–35
5.8.5 Deleting Asynchronous Remote Persistent Procedures 5–35
5.8.6 Deleting Server Handles 5–35
5.8.7 Deleting Asynchronous Request Handles 5–36
5.8.8 Debugging Asynchronous Requests 5–36
5.8.9 Mixing Synchronous and Asynchronous Requests 5–36
5.8.10 Running Asynchronous Requests on the SESSION System Handle 5–37
5.8.11 Examples 5–37

6. Design and Implementation Considerations 6–1
6.1 Primary Performance Considerations 6–2
6.1.1 Minimizing Network Traffic 6–2
6.1.2 Offloading Resource-intensive Processing Tasks 6–3
6.1.3 Choosing an AppServer Operating Mode 6–3
6.1.4 Setting the Number of Application Server Processes 6–7
6.1.5 Choosing a Transaction Model 6–8
6.1.6 Using Load Balancing 6–8
6.2 Secondary Performance Considerations 6–9
6.2.1 Connect and Disconnect Performance Issues 6–9
6.2.2 Compiled Compared to Precompiled Code 6–11
6.2.3 Progress Browse Design Considerations 6–11
6.2.4 Performance Issues Related to Schema Transmission 6–12
6.3 Transaction and Record Management Considerations 6–13
6.3.1 Comparing Normal 4GL and Automatic Transactions 6–13
6.3.2 Ensuring a Small Transaction Size 6–14
6.3.3 Returning Transaction Status to the Client 6–14
6.3.4 Maintaining Transaction Consistency Between Client and AppServer 6–15
6.3.5 Managing Conflicts Between Client and AppServer 6–18
6.3.6 Buffer Currency and NO-LOCK 6–18
6.4 Deployment Considerations 6–21
6.4.1 Minimizing the Effects of Schema Changes 6–21
6.4.2 Advantages of Using Portable R-code 6–21
6.4.3 Server Configurations 6–23
6.4.4 Network Deployment Considerations 6–23

6.5 Security Considerations 6–24
6.5.1 Overview of the Progress AppServer Security Options 6–24
6.5.2 Authentication and Authorization 6–25
6.5.3 Database Access 6–27
6.5.4 AppServer Session Access 6–27
6.5.5 Generating Audit Trails 6–30
6.5.6 Run-time Compilation 6–31
6.5.7 Operating System 6–32
6.6 Multi-language Support 6–32
6.7 Distributed Application Design and Implementation 6–33
6.7.1 Understanding the Scope of Persistent Procedures 6–33
6.7.2 Understanding Procedure Handles in Distributed Progress Sessions 6–33
6.7.3 Understanding Condition Handling in Distributed Progress Sessions 6–35
6.7.4 Schema Triggers 6–37
6.7.5 Conditions That Make Migrating Your Application Model Easier 6–38

7. Debugging 7–1
7.1 AppServer Debugging 7–2
7.1.1 Distributed Debugging 7–2
7.1.2 Remote Debugging 7–3
7.2 Using Distributed Debugging 7–3
7.2.1 About the Debugger Window 7–4
7.2.2 Restrictions on Debugger Operation 7–5
7.3 Using Remote Debugging 7–6
7.3.1 Basic Tasks 7–6
7.4 General Debugging Requirements 7–7
7.5 Examining the Progress AppServer Log File 7–8

A. Running the AppServer Without a Network A–1
A.1 Preparing NT 4.0 for Standalone Operation A–2
A.2 Creating a Non-network Configuration A–2
A.3 Installing the Remote Access Service A–2

Index Index–1

Figures

Figure 1–1: Two-tier Model 1–3
Figure 1–2: Logical Three-tier Model 1–4
Figure 1–3: N-tier Model 1–5
Figure 2–1: Separate and Distinct 4GL Sessions 2–5
Figure 2–2: Separate and Distinct Application Contexts 2–7
Figure 2–3: Connection-based Model 2–8
Figure 2–4: Multiple Clients Connected to an AppServer 2–11
Figure 2–5: Client Connected to Multiple AppServers 2–13
Figure 2–6: AppServer-to-AppServer Connections 2–15
Figure 2–7: AppServer Run-time Components 2–16
Figure 2–8: Synchronous and Asynchronous Requests 2–33
Figure 2–9: Asynchronous Request Queuing 2–36
Figure 3–1: AppServer Administration Framework 3–2
Figure 5–1: Synchronous Requests 5–39
Figure 5–2: Asynchronous Requests 5–41
Figure 6–1: Transaction Data Flow 6–16

Tables

Table 2–1: Progress AppServer Run-time Components 2–17
Table 3–1: Application Server Process Status Indications 3–26
Table 3–2: AppServer Utilities 3–30
Table 3–3: Management Tasks 3–31
Table 4–1: 4GL for Programming AppServer Procedures 4–2
Table 4–2: Transaction Handle Methods and Attributes 4–24
Table 5–1: 4GL for Programming AppServer Procedures 5–2
Table 5–2: AppServer Connection Parameters 5–10
Table 6–1: Characteristic Operating Mode Trade-offs 6–4
Table 6–2: Progress AppServer Security Considerations 6–24
Table 6–3: PROXY, REMOTE, and PERSISTENT Procedure Handle Attribute Settings 6–35
Table 6–4: Progress Conditions 6–36
Table 7–1: Content of the Debugger Listing Panel Title Bar 7–4

Procedures

a-txtest.p 4–27
a-txclnt.p 4–28
client.p 5–42
server.p 5–43

Preface

Purpose

This manual provides comprehensive information on a range of topics associated with building and implementing distributed applications using the Progress AppServer. These topics include: basic product information and terminology, a discussion of issues related to designing and implementing distributed applications, details about administrative tasks such as setting up and maintaining the Progress AppServer, specific Progress 4GL programming information to build distributed applications using the Progress AppServer, and examining code associated with a distributed application that uses the Progress AppServer.

Audience

This book is designed as a guide for anyone wanting to understand and use the Progress AppServer to build distributed applications. However, network designers, system administrators, and application programmers will find the information in this guide helpful to understand and accomplish their respective tasks to design, set up, startup, program, and maintain the Progress AppServer. For more information on how to implement a Progress AppServer solution as a new or existing Progress customer, see the "How to Use This Manual" section.

Organization of This Manual

Chapter 1, “Distributed Application Computing”

Defines distributed application computing and compares the traditional client/server model with Progress distributed application computing models based on the Progress AppServer.

Chapter 2, “Overview of the Progress AppServer”

Describes the Progress AppServer architecture and how you can access it from client applications. The chapter also introduces the basic operating modes and configuration options that you can use to define an AppServer.

Chapter 3, “Administration”

Describes how the common Progress Explorer Framework manages AppServer installations and provides step-by-step instructions to set up, start up, shut down, and maintain the Progress AppServer.

Chapter 4, “Programming the AppServer”

Describes how to program the Progress AppServer, including both application procedures and the types of procedures that you can define to set up and manage the AppServer session context for your application procedures. It also describes how to manage context, including transactions, for the different AppServer operating modes and handle error information that you pass to client applications. Code examples illustrate specific programming points.

Chapter 5, “Programming the Client Application”

Describes how to program a client application, with emphasis on 4GL clients. It explains how to connect to and disconnect from an AppServer, run and mange remote procedures, and handle error information returned from the AppServer. Code examples illustrate specific programming points.

Chapter 6, “Design and Implementation Considerations”

Identifies and describes various issues related to the design and implementation of the Progress AppServer. This chapter highlights specific performance, deployment, and security issues, and introduces some programming techniques to help you write AppServer procedures.

Chapter 7, "Debugging"

Describes two enhancements to the Progress Debugger that support debugging distributed applications and explains how to use the AppServer log file to troubleshoot your application.

Appendix A, "Running the AppServer Without a Network"

Describes how to set up your UNIX or NT environment to run an AppServer on the same machine as the client.

How to Use This Manual

This section briefly describes how you might implement a Progress AppServer solution as a new or existing Progress customer.

If you are installing Progress for the first time, you must devise an overall plan to suit your distributed computing needs. This way, from the start, the Progress AppServer can play a central role in helping to fulfill your performance, deployment, and security goals.

If you already using Progress, you can implement the Progress AppServers with all the same databases that a regular Progress client accesses. You can partition your application and encapsulate the business rules for your application using the familiar Progress 4GL. However, adopting this new approach might require some analysis and redesign of your existing applications.

For information on Progress AppServer architecture and modes of operation, see Chapter 2, "Overview of the Progress AppServer." For more information on how to structure an application to take advantage of AppServer features, see Chapter 6, "Design and Implementation Considerations."

Typographical Conventions

This manual uses the following typographical conventions:

- **Bold typeface** indicates:
  - Commands or characters that the user types
  - That a word carries particular weight or emphasis
- *Italic typeface* indicates:
  - Progress variable information that the user supplies
  - New terms
  - Titles of complete publications
- `Monospaced typeface` indicates:
  - Code examples
  - System output
  - Operating system filenames and pathnames

The following typographical conventions are used to represent keystrokes:

- Small capitals are used for Progress key functions and generic keyboard keys.

  END-ERROR, GET, GO
  ALT, CTRL, SPACEBAR, TAB

- When you have to press a combination of keys, they are joined by a dash. You press and hold down the first key, then press the second key.

  CTRL-X

- When you have to press and release one key, then press another key, the key names are separated with a space.

  ESCAPE H
  ESCAPE CURSOR-LEFT

Syntax Notation

The syntax for each component follows a set of conventions:

- Uppercase words are keywords. Although they are always shown in uppercase, you can use either uppercase or lowercase when using them in a procedure.

  In this example, ACCUM is a keyword:

  SYNTAX

  ```
  ACCUM aggregate expression
  ```

- Italics identify options or arguments that you must supply. These options can be defined as part of the syntax or in a separate syntax identified by the name in italics. In the ACCUM function above, the *aggregate* and *expression* options are defined with the syntax for the ACCUM function in the *Progress Language Reference*.

- You must end all statements (except for DO, FOR, FUNCTION, PROCEDURE, and REPEAT) with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon, as in this example:

  ```
  FOR EACH Customer:
     DISPLAY Name.
  END.
  ```

- Square brackets (**[ ]**) around an item indicate that the item, or a choice of one of the enclosed items, is optional.

  In this example, STREAM *stream*, UNLESS-HIDDEN, and NO-ERROR are optional:

  SYNTAX

  ```
  DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
  ```

In some instances, square brackets are not a syntax notation, but part of the language.

For example, this syntax for the INITIAL option uses brackets to bound an initial value list for an array variable definition. In these cases, normal text brackets ([ ]) are used:

SYNTAX

```
INITIAL [ constant [ , constant ] ... ]
```

NOTE: The ellipsis (**...**) indicates repetition, as shown in a following description.

- Braces (**{ }**) around an item indicate that the item, or a choice of one of the enclosed items, is required.

  In this example, you must specify the items BY and *expression* and can optionally specify the item DESCENDING, in that order:

  SYNTAX

  ```
  { BY expression [ DESCENDING ] }
  ```

  In some cases, braces are not a syntax notation, but part of the language.

  For example, a called external procedure must use braces when referencing arguments passed by a calling procedure. In these cases, normal text braces ({ }) are used:

  SYNTAX

  ```
  { &argument-name }
  ```

- A vertical bar (**|**) indicates a choice.

  In this example, EACH, FIRST, and LAST are optional, but you can only choose one:

  SYNTAX

  ```
  PRESELECT [ EACH | FIRST | LAST ] record-phrase
  ```

In this example, you must select one of *logical-name* or *alias*:

SYNTAX

```
CONNECTED ( { logical-name | alias } )
```

- Ellipses (**. . .**) indicate that you can choose one or more of the preceding items. If a group of items is enclosed in braces and followed by ellipses, you must choose one or more of those items. If a group of items is enclosed in brackets and followed by ellipses, you can optionally choose one or more of those items.

  In this example, you must include two expressions, but you can optionally include more. Note that each subsequent expression must be preceded by a comma:

  SYNTAX

  ```
  MAXIMUM ( expression , expression [ , expression ] ... )
  ```

  In this example, you must specify MESSAGE, then at least one of *expression* or SKIP, but any additional number of *expression* or SKIP is allowed:

  SYNTAX

  ```
  MESSAGE { expression | SKIP [ (n) ] } ...
  ```

  In this example, you must specify {*include-file*, then optionally any number of *argument* or &*argument-name* = "*argument-value*", and then terminate with }:

  SYNTAX

  ```
  { include-file
      [ argument | &argument-name = "argument-value" ] ... }
  ```

- In some examples, the syntax is too long to place in one horizontal row. In such cases, **optional** items appear individually bracketed in multiple rows in order, left-to-right and top-to-bottom. This order generally applies, unless otherwise specified. **Required** items also appear on multiple rows in the required order, left-to-right and top-to-bottom. In cases where grouping and order might otherwise be ambiguous, braced (required) or bracketed (optional) groups clarify the groupings.

  In this example, WITH is followed by several optional items:

  SYNTAX

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```

  In this example, ASSIGN requires one of two choices: either one or more of *field*, or one of *record*. Other options available with either *field* or *record* are grouped with braces and brackets. The open and close braces indicate the required order of options:

  SYNTAX

```
ASSIGN {   { [ FRAME frame ]
               { field [ = expression ] }
               [ WHEN expression ]
           } ...
         | { record [ EXCEPT field ... ] }
       }
```

Progress Messages

Progress displays several types of messages to inform you of routine and unusual occurrences:

- Execution messages inform you of errors encountered while Progress is running a procedure (for example, if Progress cannot find a record with a specified index field value).
- Compile messages inform you of errors found while Progress is reading and analyzing a procedure prior to running it (for example, if a procedure references a table name that is not defined in the database).
- Startup messages inform you of unusual conditions detected while Progress is getting ready to execute (for example, if you entered an invalid startup parameter).

After displaying a message, Progress proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify, or that are assumed, as part of the procedure. This is the most common action taken following execution messages.
- Returns to the Progress Procedure Editor so that you can correct an error in a procedure. This is the usual action taken following compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

Progress messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

Use Progress online help to get more information about Progress messages. On the Windows platform, many Progress tools include the following Help menu options to provide information about messages:

- Choose Help→ Recent Messages to display detailed descriptions of the most recent Progress message and all other messages returned in the current session.
- Choose Help→ Messages, then enter the message number to display a description of any Progress message. (If you encounter an error that terminates Progress, make a note of the message number before restarting.)
- In the Procedure Editor, press the **HELP** key (**F2** or **CTRL-W**).

On the UNIX platform, you can use the Progress PRO command to start a single-user mode character Progress client session and view a brief description of a message by providing its number. Follow these steps:

1 ♦ Start the Progress Procedure Editor:

```
install-dir/dlc/bin/pro
```

2 ♦ Press **F3** to access the menu bar, then choose Help→ Messages.

3 ♦ Type the message number, and press **ENTER**. Details about that message number appear.

4 ♦ Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose File→ Exit.

Other Useful Documentation

This section lists Progress Software Corporation documentation that you might find useful. Unless otherwise specified, these manuals support both Windows **and** Character platforms and are provided in electronic documentation format on CD-ROM.

Getting Started

Progress Electronic Documentation Installation and Configuration Guide (Hard copy only)

A booklet that describes how to install the Progress EDOC viewer and collection on UNIX and Windows.

Progress Installation and Configuration Guide Version 9 for UNIX

A manual that describes how to install and set up Progress Version 9.1 for the UNIX operating system.

Progress Installation and Configuration Guide Version 9 for Windows

A manual that describes how to install and set up Progress Version 9.1 for all supported Windows and Citrix MetaFrame operating systems.

Progress Version 9 Product Update Bulletin

A guide that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Progress Application Development Environment — Getting Started (Windows only)

A practical guide to graphical application development within the Progress Application Development Environment (ADE). This guide includes an overview of the ADE and its tools, an overview of Progress SmartObject technology, and tutorials and exercises that help you better understand SmartObject technology and how to use the ADE to develop applications.

Progress Language Tutorial for Windows and *Progress Language Tutorial for Character*

Platform-specific tutorials designed for new Progress users. The tutorials use a step-by-step approach to explore the Progress application development environment using the 4GL.

Progress Master Glossary for Windows and *Progress Master Glossary for Character* (EDOC only)

Platform-specific master glossaries for the Progress documentation set. These books are in electronic format only.

Progress Master Index and Glossary for Windows and *Progress Master Index and Glossary for Character* (Hard copy only)

Platform-specific master indexes and glossaries for the Progress hard-copy documentation set.

Progress Startup Command and Parameter Reference

A reference manual that describes the Progress startup commands and parameters in alphabetical order.

Welcome to Progress (Hard copy only)

A booklet that explains how Progress software and media are packaged. An icon-based map groups the documentation by functionality, providing an overall view of the documentation set. *Welcome to Progress* also provides descriptions of the various services Progress Software Corporation offers.

Development Tools

Progress ADM 2 Guide

A guide to using the Application Development Model, Version 2 (ADM 2) application architecture to develop Progress applications. It includes instructions for building and using Progress SmartObjects.

Progress ADM 2 Reference

A reference for the Application Development Model, Version 2 (ADM 2) application. It includes descriptions of ADM 2 functions and procedures.

Progress AppBuilder Developer's Guide (Windows only)

A programmer's guide to using the Progress AppBuilder visual layout editor. AppBuilder is a Rapid Application Development (RAD) tool that can significantly reduce the time and effort required to create Progress applications.

Progress Basic Database Tools (Character only; information for Windows is in online help)

A guide for the Progress Database Administration tools, such as the Data Dictionary.

Progress Basic Development Tools (Character only; information for Windows is in online help)

A guide for the Progress development toolset, including the Progress Procedure Editor and the Application Compiler.

Progress Debugger Guide

A guide for the Progress Application Debugger. The Debugger helps you trace and correct programming errors by allowing you to monitor and modify procedure execution as it happens.

Progress Help Development Guide (Windows only)

A guide that describes how to develop and integrate an online help system for a Progress application.

Progress Translation Manager Guide (Windows only)

A guide that describes how to use the Progress Translation Manager tool to manage the entire process of translating the text phrases in Progress applications.

Progress Visual Translator Guide (Windows only)

A guide that describes how to use the Progress Visual Translator tool to translate text phrases from procedures into one or more spoken languages.

Reporting Tools

Progress Report Builder Deployment Guide (Windows only)

An administration and development guide for generating Report Builder reports using the Progress Report Engine.

Progress Report Builder Tutorial (Windows only)

A tutorial that provides step-by-step instructions for creating eight sample Report Builder reports.

Progress Report Builder User's Guide (Windows only)

A guide for generating reports with the Progress Report Builder.

Progress Results Administration and Development Guide (Windows only)

A guide for system administrators that describes how to set up and maintain the Results product in a graphical environment. This guide also describes how to program, customize, and package Results with your own products. In addition, it describes how to convert character-based Results applications to graphical Results applications.

Progress Results User's Guide for Windows and *Progress Results User's Guide for UNIX*

Platform-specific guides for users with little or no programming experience that explain how to query, report, and update information with Results. Each guide also helps advanced users and application developers customize and integrate Results into their own applications.

4GL

Progress External Program Interfaces

A guide to accessing non-Progress applications from Progress. This guide describes how to use system clipboards, UNIX named pipes, Windows dynamic link libraries, Windows dynamic data exchange, Windows ActiveX controls, and the Progress Host Language Call Interface to communicate with non-Progress applications and extend Progress functionality.

Progress Internationalization Guide

A guide to developing Progress applications for markets worldwide. The guide covers both internationalization—writing an application so that it adapts readily to different locales (languages, cultures, or regions)—and localization—adapting an application to different locales.

Progress Language Reference

A three-volume reference set that contains extensive descriptions and examples for each statement, phrase, function, operator, widget, attribute, method, and event in the Progress language.

Progress Programming Handbook

A two-volume handbook that details advanced Progress programming techniques.

Database

Progress Database Design Guide

A guide that uses a sample database and the Progress Data Dictionary to illustrate the fundamental principles of relational database design. Topics include relationships, normalization, indexing, and database triggers.

Progress Database Administration Guide and Reference

This guide describes Progress database administration concepts and procedures. The procedures allow you to create and maintain your Progress databases and manage their performance.

DataServers

Progress DataServer Guides

These guides describe how to use the DataServers to access non-Progress databases. They provide instructions for building the DataServer modules, a discussion of programming considerations, and a tutorial. Each DataServer has its own guide, for example, the *Progress DataServer for ODBC Guide*, the *Progress DataServer for ORACLE Guide*, or the *Progress/400 Product Guide*.

MERANT ODBC Branded Driver Reference

The Enterprise DataServer for ODBC includes MERANT ODBC drivers for all the supported data sources. For configuration information, see the MERANT documentation, which is available as a PDF file in *installation-path*\odbc. To read this file you must have the Adobe Acrobat Reader Version 3.1 or higher installed on your system. If you do not have the Adobe Acrobat Reader, you can download it from the Adobe Web site at: http://www.adobe.com/prodindex/acrobat/readstep.html.

SQL-89/Open Access

Progress Embedded SQL-89 Guide and Reference

A guide to Progress Embedded SQL-89 for C, including step-by-step instructions on building ESQL-89 applications and reference information on all Embedded SQL-89 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-89 and the ANSI standards upon which it is based.

Progress Open Client Developer's Guide

A guide that describes how to write and deploy Java and ActiveX applications that run as clients of the Progress AppServer. The guide includes information about how to expose the AppServer as a set of Java classes or as an ActiveX server.

Progress SQL-89 Guide and Reference

A user guide and reference for programmers who use interactive Progress/SQL-89. It includes information on all supported SQL-89 statements, SQL-89 Data Manipulation Language components, SQL-89 Data Definition Language components, and supported Progress functions.

SQL-92

Progress Embedded SQL-92 Guide and Reference

A guide to Progress Embedded SQL-92 for C, including step-by-step instructions for building ESQL-92 applications and reference information about all Embedded SQL-92 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-92 and the ANSI standards upon which it is based.

Progress JDBC Driver Guide

A guide to the Java Database Connectivity (JDBC) interface and the Progress SQL-92 JDBC driver. It describes how to set up and use the driver and details the driver's support for the JDBC interface.

Progress ODBC Driver Guide

A guide to the ODBC interface and the Progress SQL-92 ODBC driver. It describes how to set up and use the driver and details the driver's support for the ODBC interface.

Progress SQL-92 Guide and Reference

A user guide and reference for programmers who use Progress SQL-92. It includes information on all supported SQL-92 statements, SQL-92 Data Manipulation Language components, SQL-92 Data Definition Language components, and Progress functions. The guide describes how to use the Progress SQL-92 Java classes and how to create and use Java stored procedures and triggers.

Deployment

Progress Client Deployment Guide

A guide that describes the client deployment process and application administration concepts and procedures.

Progress Developer's Toolkit

A guide to using the Developer's Toolkit. This guide describes the advantages and disadvantages of different strategies for deploying Progress applications and explains how you can use the Toolkit to deploy applications with your selected strategy.

Progress Portability Guide

A guide that explains how to use the Progress toolset to build applications that are portable across all supported operating systems, user interfaces, and databases, following the Progress programming model.

WebSpeed

Getting Started with WebSpeed

Provides an introduction to the WebSpeed Workshop tools for creating Web applications. It introduces you to all the components of the WebSpeed Workshop and takes you through the process of creating your own Intranet application.

WebSpeed Developer's Guide

Provides a complete overview of WebSpeed and the guidance necessary to develop and deploy WebSpeed applications on the Web.

WebSpeed Installation and Configuration Guide

Provides instructions for installing WebSpeed on Windows and UNIX systems. It also discusses designing WebSpeed environments, configuring WebSpeed Brokers, WebSpeed Agents, and the NameServer, and connecting to a variety of data sources.

WebSpeed Version 3 Product Update Bulletin

A booklet that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Welcome to WebSpeed! (Hard copy only)

A booklet that explains how WebSpeed software and media are packaged. *Welcome to WebSpeed!* also provides descriptions of the various services Progress Software Corporation offers.

Reference

Pocket Progress (Hard copy only)

A reference that lets you quickly look up information about the Progress language or programming environment.

Pocket WebSpeed (Hard copy only)

A reference that lets you quickly look up information about the SpeedScript language or the WebSpeed programming environment.

1

Distributed Application Computing

This chapter:

- Defines distributed application computing
- Compares the traditional client/server computing model with the Progress distributed application computing models

1.1 What Is Distributed Application Computing?

Distributed application computing is an architectural approach to designing application software that runs across a network. It maximizes your processing capabilities by allowing you to distribute portions of your business application throughout your enterprise. By emphasizing a flexible, modular use of computer resources, your enterprise can potentially gain greater performance returns, realize the benefits of customized deployment strategies, and provide a security model to ensure the integrity of your application data.

Two important aspects of distributed application computing are:

- Application partitioning
- N-tier model

The Progress AppServer provides features and functionality that allow you to take full advantage of both.

The remaining text in this section details each of these aspects.

1.1.1 Application Partitioning

Application partitioning is the process of segmenting application logic from the user interface and data, and running it across multiple computers on a local or wide area network. Depending on your network configuration and performance goals, you can partition your application logic among many available resources on your network.

However, application partitioning is more than a style of computing to enhance your overall application performance and deployment options; it is a catalyst for examining current methods of running applications to determine maximum performance, deployment, and security objectives through application distribution.

1.1.2 The N-tier Model

The *n-tier model* is a computing model that supports flexible network structuring so that you can distribute application logic and the processing load among many machines across your distributed network. Because the n-tier model can support an unlimited number of clients, partitioning and deploying your application in this manner allows you to break up monolithic applications and flexibly reconfigure them to run in a network environment that better supports your computing needs.

The n-tier model supports the logical separation of user interface, application logic, and data access across three or more machines. Because of this flexibility, application partitioning is closely tied to two basic deployment approaches in a distributed environment: the logical three-tier model and the physical n-tier model.

The *logical three-tier model* supports separating the application logic from the user interface in the application implementation, and moving it to a remote server where the application data resides. A configuration based on the logical three-tier model remains a physical client/server application in that it is deployed across two machines, but the user interface, application logic, and data are logically separated from one another. In contrast, the *physical n-tier model* supports both logically and physically discrete segments. Each segment-user interface, application logic, and data-can be deployed on separate machines across the enterprise.

In a Progress distributed environment, you are not restricted to using just these models, and these models can co-exist in a distributed environment. These models are presented in this chapter as a means to understanding the various capabilities available in the Progress distributed environment.

1.2 Comparing Computing Models

This section briefly identifies the processing capabilities available with the traditional two-tier client/server model in contrast to capabilities available in the Progress distributed application computing models. This comparison helps to illustrate some of the strategic advantages that the Progress distributed application computing model offers.

1.2.1 The Two-tier Model

Traditional client/server development is based on a logical and physical *two-tier computing model*. It is deployed across two machines and connected through a network. The user interface and application logic are tightly integrated and located on a client machine, while the data resides on a separate server machine.

Figure 1–1 shows this two-tier configuration. Often, the client application runs on a limited desktop machine; the real computing power is reserved for the server.

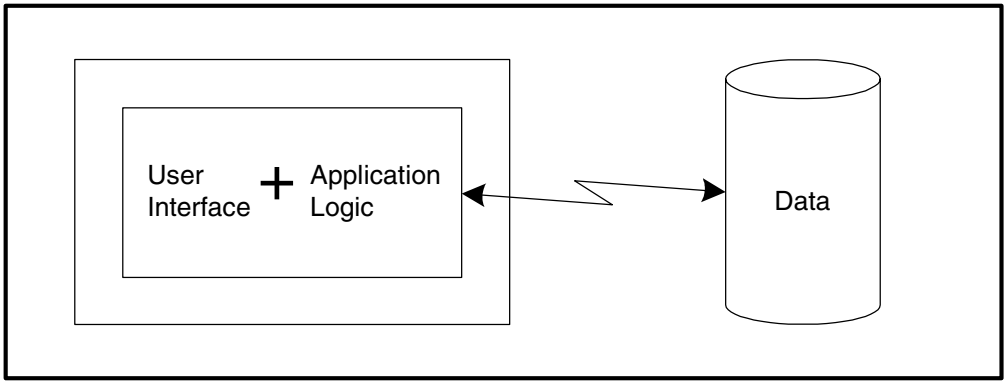


Figure 1–1: Two-tier Model

Although applications based on this model are relatively easy to build and maintain, they are generally limited in size and computing power. These limitations can restrict the possibility of increased performance capabilities.

1.2.2 Progress Distributed Application Computing Models

In contrast, Progress distributed application computing, through the introduction of the Progress AppServer, expands computing capabilities beyond the limited boundaries of the two-tier model. This section discusses two models: the logical three-tier model and the n-tier model. As previously mentioned in this chapter, these models help illustrate how you can distribute your application in a Progress distributed environment. You are not limited to using just these models.

The Logical Three-Tier Model

The logical three-tier model, as shown in Figure 1–2, has a user interface that is physically separate from another machine that contains the application logic **and** the data. Therefore, physically there are only two machines, but technically, or logically, there are three tiers: the user interface, the application logic, and the data.

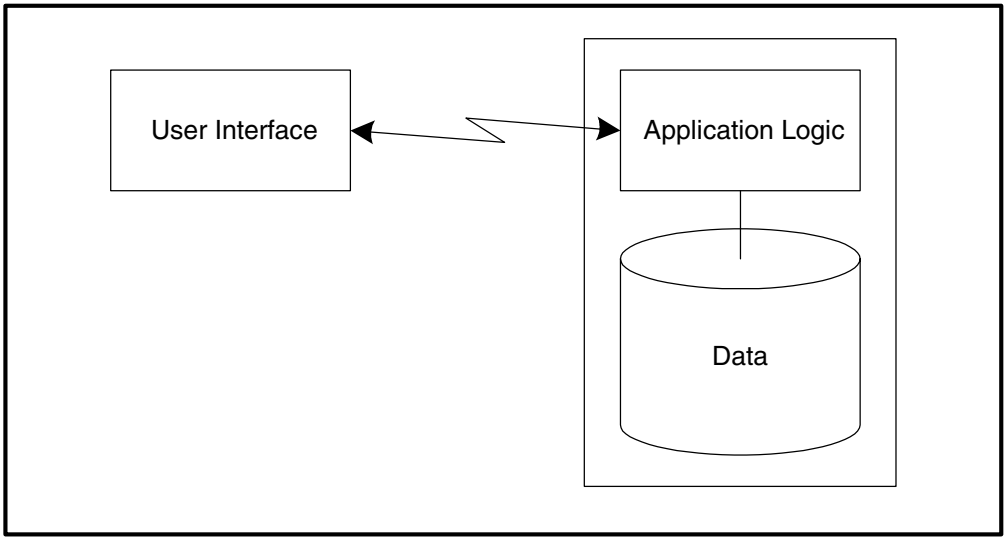


Figure 1–2: Logical Three-tier Model

The configuration of the logical three-tier model promotes building business rules and deploying them in an environment close to where the data resides. A Progress implementation of this model enables the Application Server to connect to the database server using shared memory; the need to access information over the network has been eliminated, providing faster data access.

Another important feature of this model is the fact that the Application Server remotely processes your information, and only returns the results to you; this capability minimizes the number of network messages while it delivers the specific data you need.

The N-tier Model

In addition to the logical three-tier model, the Progress distributed environment supports an n-tier configuration. Although this model does not capitalize on the use of shared memory, and introduces an additional network connection that you do not have in the logical three-tier model, the deployment flexibility of this model might have significant benefits from an overall enterprise perspective. Figure 1–3 shows how application logic can be distributed to dedicated machines without any local database connections.

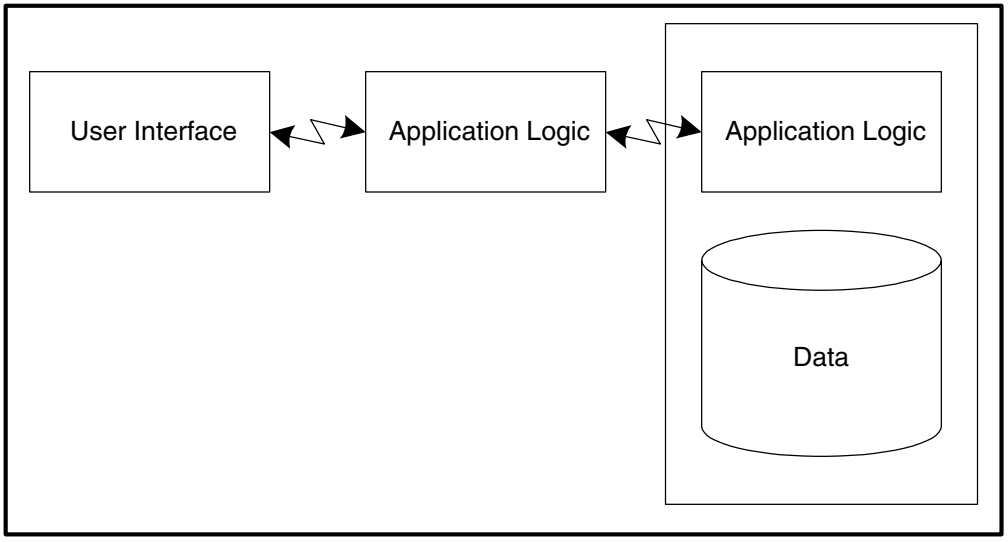


Figure 1–3: N-tier Model

For example, you might want to configure a Progress AppServer on a system that is a dedicated computation engine with no locally attached database. With the Progress AppServer, you can build arbitrary complex applications across any number of computing tiers, based on the business problem that you are trying to solve.

The remaining chapters in this manual help you understand the Progress AppServer and how you can design and implement it as an integral part of your distributed application computing environment, beginning with Chapter 2, "Overview of the Progress AppServer."

2

Overview of the Progress AppServer

This chapter introduces the Progress AppServer, the core of the Progress n-tier computing solution, and provides an overview of its central concepts. The Progress AppServer is the mechanism that allows you to distribute a 4GL application across multiple machines in a network. It does this by providing separate Progress sessions that communicate by having one session (the *client application session*) establish a connection with an AppServer. The connected AppServer then allows the client application session to execute 4GL procedures in another Progress session running on the AppServer and to return any results to the client.

The Progress AppServer also allows you to establish connections from specially prepared non-Progress client (*Open Client*) applications that can then execute 4GL procedures on the AppServer using mechanisms native to their own programming environment.

The AppServer also allows you to configure and program it for different modes of operation. The supported operating modes provide different mixes of resource consumption and performance characteristics. Thus, one or another operating mode might better meet your overall performance requirements, depending on the application.

This chapter provides the information necessary for you to begin preparing distributed applications using both Progress clients and Open Clients, and describes the following topics:

- Overview of the architecture
- Run-time components
- Connection process
- Understanding AppServer operating modes
- Understanding synchronous and asynchronous requests

2.1 Overview of the Architecture

The Progress AppServer helps you to build complex distributed applications using the 4GL by allowing you to *initiate remote procedure requests* (call remote procedures) within a client application. Each such request executes a *remote procedure*, a procedure running in a different Progress (AppServer) session that typically resides in another location on the network.

2.1.1 Application Partitioning

To take advantage of the Progress AppServer you must split your application into application partitions. The Progress AppServer provides the mechanism for these application partitions to communicate using a remote procedure calling mechanism.

Splitting your application into application partitions promotes a flexible and economical use of computing resources. For example, you can partition your application such that most compute intensive portions are co-located with the Progress Database. You set up the partition by installing the Progress AppServer at this site and access it from your client machine using the Progress AppServer remote procedure call mechanism.

Using the same machine for the application partition that accesses the database allows you to take advantage of shared memory access, eliminating the need to access the database over the network. The Progress AppServer allows you to configure these application partitions on your network to meet your organization's performance, deployment, and security objectives. For example, configuring the AppServer with sole access to the database provides a layer of security for the database. Client applications can only query and update tables indirectly through the AppServer. The 4GL code that actually performs the updates and queries is not located on the client machine. Rather, it is located on the database machine and is executed by the Progress AppServer.

Also, you can implement your client application partition in several different environments. In addition to standard 4GL clients, the AppServer can support a Java application or an ActiveX Controller as a client that you build using the Progress Open Client Toolkit.

2.1.2 Reasons to Partition

Reasons to partition your application using the Progress AppServer include:

- **Performance**—Using the AppServer, you can significantly improve the performance of your distributed application because the AppServer allows you to distribute the processing load across the machines in your network. Network traffic is minimized, which can result in a dramatic improvement in client application throughput and/or response time depending on how you program your application and configure it with the AppServer.

You further also improve the performance of your distributed application, especially one that has an event-driven 4GL client, by initiating procedure requests asynchronously. By default, a procedure executes on the AppServer as a *synchronous request*, requiring the client to wait for the result before continuing execution. If you initiate an *asynchronous request*, the 4GL client continues execution without waiting for the result. Thus, the user can continue to execute client functions after submitting one or more asynchronous requests for execution. The client then processes the results of each asynchronous request as it becomes available.

- **Security**—You can restrict access to data, as required, but at the same time take advantage of the ability to distribute and share data throughout your network. In addition to standard Progress Database security features and any other security features that your enterprise employs, the Progress AppServer allows you to protect your data at the *business logic* level (where business rules are applied in your application). The Progress AppServer accesses your database with user authorization that is independent of the client application. Using the Progress AppServer allows you to restrict database access to the AppServer user, thus forcing all client applications to go through the AppServer to gain access to your corporate data. It is the business logic that you write using the 4GL and deploy with the AppServer that provides this business logic level of security.

- **Maintenance**—Because your core business logic is deployed at your AppServer site rather than at the client application sites across the network, maintenance of your distributed application is simplified significantly. A modification to the business logic only needs to be made at the single AppServer site rather than at each of the client sites across the network.

- **Open Client Access**—Using the Progress Open Client Toolkit, you can build a client application in Java or as an ActiveX Controller, and have that client application execute your business logic on an AppServer deployed on some other machine in your network. A single AppServer instance can simultaneously support 4GL, Java, and ActiveX client applications.

- **Intelligent Data Interface**—As the interface to the Progress Database, the AppServer provides functionality similar to database triggers and stored procedures. In addition to native Progress-database triggers, you can use procedures running on the AppServer to filter or otherwise process database queries and updates.

2.1.3 AppServer Components

An AppServer relies on the following components to manage client connections and execute remote procedure requests:

- **Application Server Process**—A process running on an AppServer that executes 4GL procedures in response to remote procedure requests. An AppServer can have any number of Application Server processes to service remote procedure requests from any number of client applications.
- **Application Broker**—The process that manages client connections.
- **NameServer**—A process that accepts a client request for connection to an AppServer and directs that request to the appropriate AppServer instance.

An *AppServer instance* is one Application Broker and the pool of Application Server processes that it manages. An AppServer instance has a specified configuration and makes a well-defined set of 4GL procedures available for execution by client applications.

An *AppServer installation* is the Progress AppServer product environment installed on a particular machine. Depending on your licensing and resources, you can configure and run multiple AppServer instances using a single AppServer installation.

For more information on AppServer components and how they interact, see the "Run-time Components and Operation" section. For more information on how client applications connect to an AppServer instance, see the "Connection Process" section. The following sections provide more basic information on how clients and AppServers and the types of configurations that AppServers support.

2.1.4 How Does a Remote Procedure Execute?

Once a client has established a connection to an AppServer, the client application can initiate requests to run remote procedures. A *remote procedure* is a procedure executed by an Application Server process as a result of being directly called by a client application. A 4GL client can execute a remote procedure by invoking a RUN statement on an external procedure with the ON SERVER option or on an internal procedure within a previously executed remote persistent procedure. A Java or ActiveX Controller client can execute a remote procedure by invoking a method that maps to the remote procedure on the AppServer.

A client application can also execute a remote procedure persistently (*remote persistent procedure*). In this case, the procedure leaves its context available to the AppServer session after it returns (exactly like any persistent procedure in a 4GL client session). The persistent procedure context is not deleted until the client deletes it or until the client disconnects from the AppServer instance.

2.1.5 The Concept of Separate Sessions

An AppServer and its client applications run in separate sessions. Essentially, this means that they have no context in common. For Java and ActiveX Controller applications that run in separately built and executed processes, this is clear. For 4GL client applications, it is not so obvious because, although both the 4GL client and the AppServer run in separate processes, they both run Progress sessions in which some context is exchanged using a common 4GL mechanism.

Figure 2–1 shows where a 4GL client application and its corresponding AppServer session maintain processing capabilities that are separate and distinct from each other.

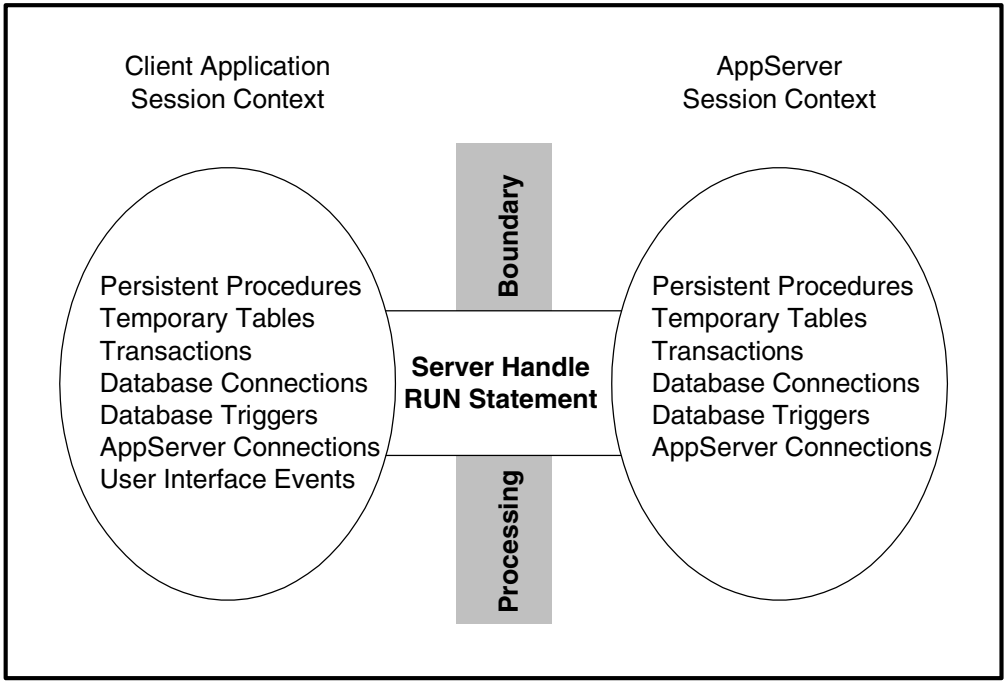


Figure 2–1: Separate and Distinct 4GL Sessions

The term *client application session* refers to all processing activities that occur within the context of the client application. Similarly, the term *AppServer session* refers to all processing activities that take place exclusively in the context of Application Server processes running on the AppServer.

NOTE: Each AppServer session exists in exactly one Application Server process. However, depending on the operating mode configured for an AppServer, the requests that a client application sends over a given connection can be processed in one or more *Application Server sessions*. For more information, see the "Understanding AppServer Operating Modes" section.

These two sessions maintain a clear *separation of context*: persistent procedures, transactions, database connections, AppServer connections, and database trigger events are separately scoped and operate only in the session in which they are invoked.

For example, even if a database is connected within an AppServer session, the client application must also connect to the database using a separate database connection if it wants direct access. For more information on managing database access in a distributed application environment, see Chapter 6, "Design and Implementation Considerations."

To further understand the concept of separate sessions, it is helpful to imagine the existence of a *processing boundary* between the two sessions. It reinforces the fact that the separate sessions exist. Only remote procedure requests can break through this processing boundary. A 4GL client invokes these requests using the RUN statement and the CONNECT( ) and DISCONNECT( ) methods on a server object handle. For more information, see Chapter 5, "Programming the Client Application."

For Java and ActiveX clients, specially prepared methods perform the same function, as shown in Figure 2–2. For more information, see the *Progress Open Client Developer's Guide*.

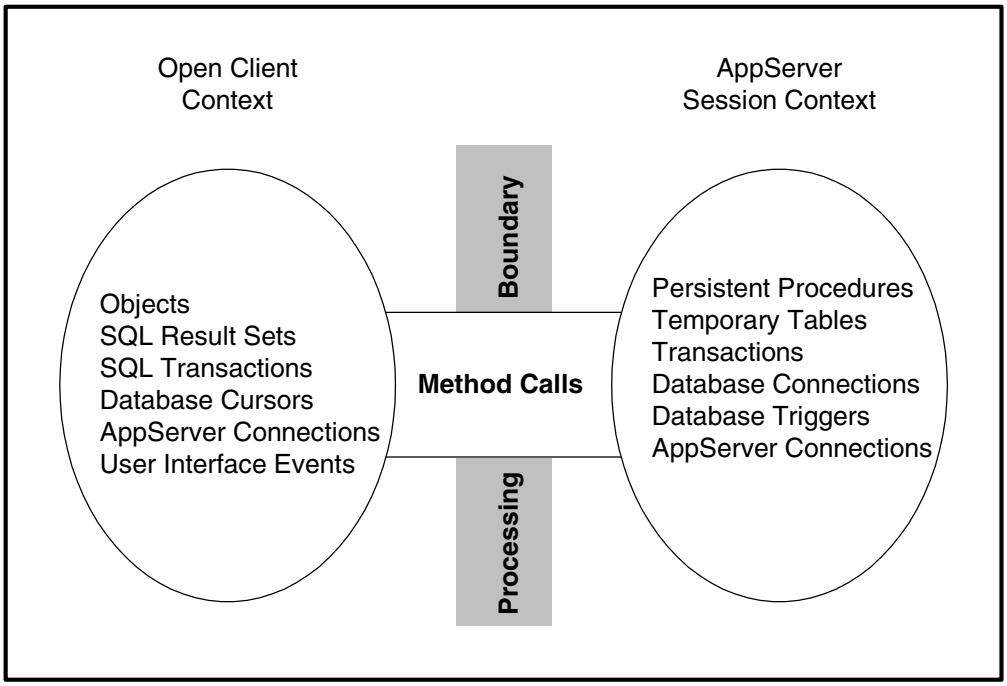


Figure 2–2: Separate and Distinct Application Contexts

2.1.6 The Connection-based Model

Communication between a client application and an AppServer is connection-based. That is, before a client can send a remote procedure request to an AppServer, it must establish a connection to that AppServer. The client can then send the remote procedure request in the context of that connection and can continue to send remote procedure requests using the same connection until the client disconnects from the AppServer.

Connection Components

A client application connects to an AppServer with the help of a NameServer, as shown in Figure 2–3. The dotted arrows indicate communications required to establish a connection between a client application and the AppServer instances.

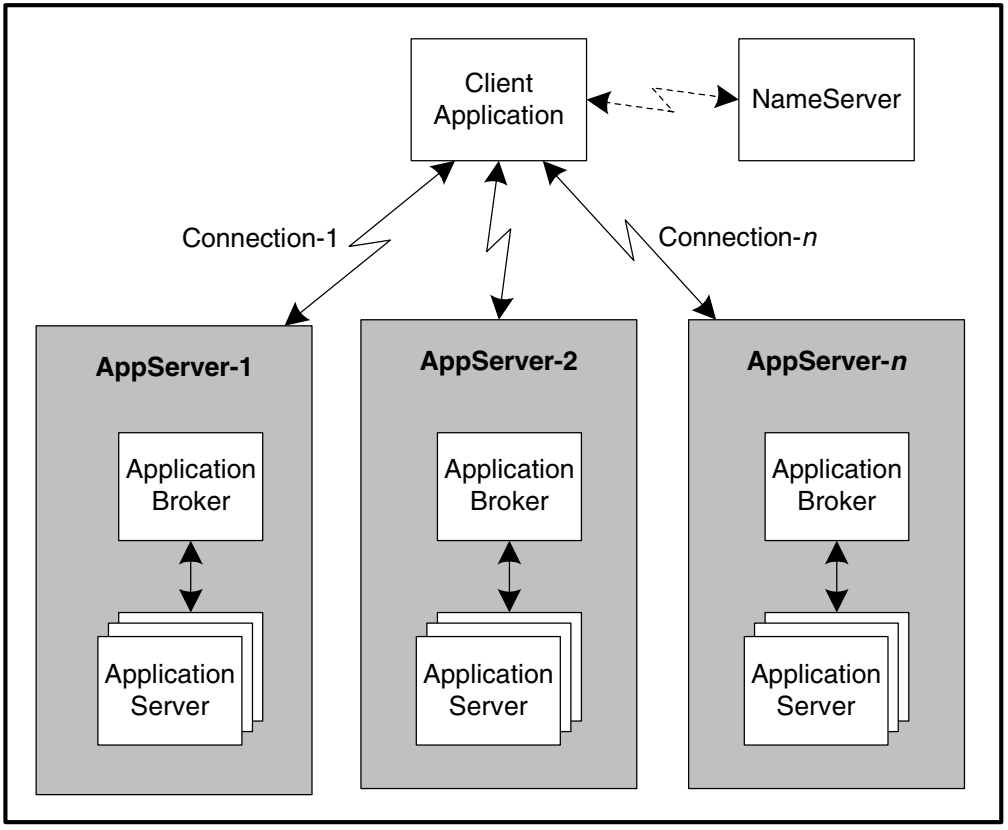


Figure 2–3: Connection-based Model

The NameServer maintains a list of the available AppServers in the network. When the client application wants to connect to an AppServer, it requests an available AppServer from the NameServer and establishes the connection with the AppServer supplied. For more information on NameServers and how they coordinate AppServer connections for client applications, see the "Run-time Components and Operation" section. For more information on how AppServers maintain connections for client applications, see the "Connection Process" section.

The Connection ID

When an AppServer accepts a connection request from a client, the Application Broker assigns a connection identifier (ID) that uniquely identifies the connection between a client and an AppServer.

The Application Broker and all Application Server processes use the connection ID as an identifier when they log any information associated with the connection. Progress also makes this connection ID available to both the client application and the AppServer application. Both the client application and AppServer session can use this connection ID, which can be used to maintain any connection-related information, such as an application log.

This connection ID has the following format:

SYNTAX

```
appserver-host::appserver-name::global-unique-id
```

appserver-host

The host name of the machine where the connected AppServer is running.

appserver-name

The name of the connected AppServer configuration. For more information on how you can specify this name, see Chapter 3, "Administration."

global-unique-id

A value guaranteed to be globally unique for all time. This value can never be generated by some other connection at the same or a different AppServer within a single computer network.

In general, you can compare connection IDs strictly for equality, but other types of comparisons are irrelevant.

Connecting an AppServer Instance

4GL clients use the CONNECT( ) method on a server object handle to establish an AppServer connection. Java and ActiveX Controller clients use similar methods to connect an AppServer. For more information on connecting from the 4GL, see Chapter 5, "Programming the Client Application." For information on connecting from a Java or ActiveX Controller application, see the *Progress Open Client Developer's Guide*.

Disconnecting an AppServer Instance

A client application can disconnect an AppServer at any time. A 4GL client disconnects by using the DISCONNECT( ) method on the server object handle to the AppServer. A Java or ActiveX Controller client uses a similar method to disconnect the AppServer.

2.1.7 Configurations

The Progress AppServer affords great flexibility in how you distribute an application. In addition to the most basic configuration, one client application connected to one AppServer, you can distribute your applications using any combination of the following basic configuration scenarios:

- Multiple client applications connected to one AppServer
- Client applications connected to multiple AppServers
- AppServers connected to AppServers

Each of these scenarios is described in the following sections.

Multiple Client Applications Connected to One AppServer

Figure 2–4 shows a connection to an AppServer from multiple client applications, with the AppServer configured with a local database. In this scenario, all running Application Server processes share self-service access to the same database. Client applications, however, have access only to data that the AppServer provides and never access the database directly. To regulate this data access, you can implement many different security arrangements on the AppServer. Each one typically restricts remote procedure access and execution based on any combination of user-based (authorization) and application-based (functional) security.

Also, each client application can be quite different and unrelated to the others, except that they all require data that the same AppServer provides. In the course of interacting with the AppServer, a client application might call a remote procedure that connects and accesses an additional database connected only to the Application Server process handling that particular request.

NOTE: Each Application Server process in any AppServer configuration has the same database access capabilities as a standard 4GL client, including access to remote database servers.

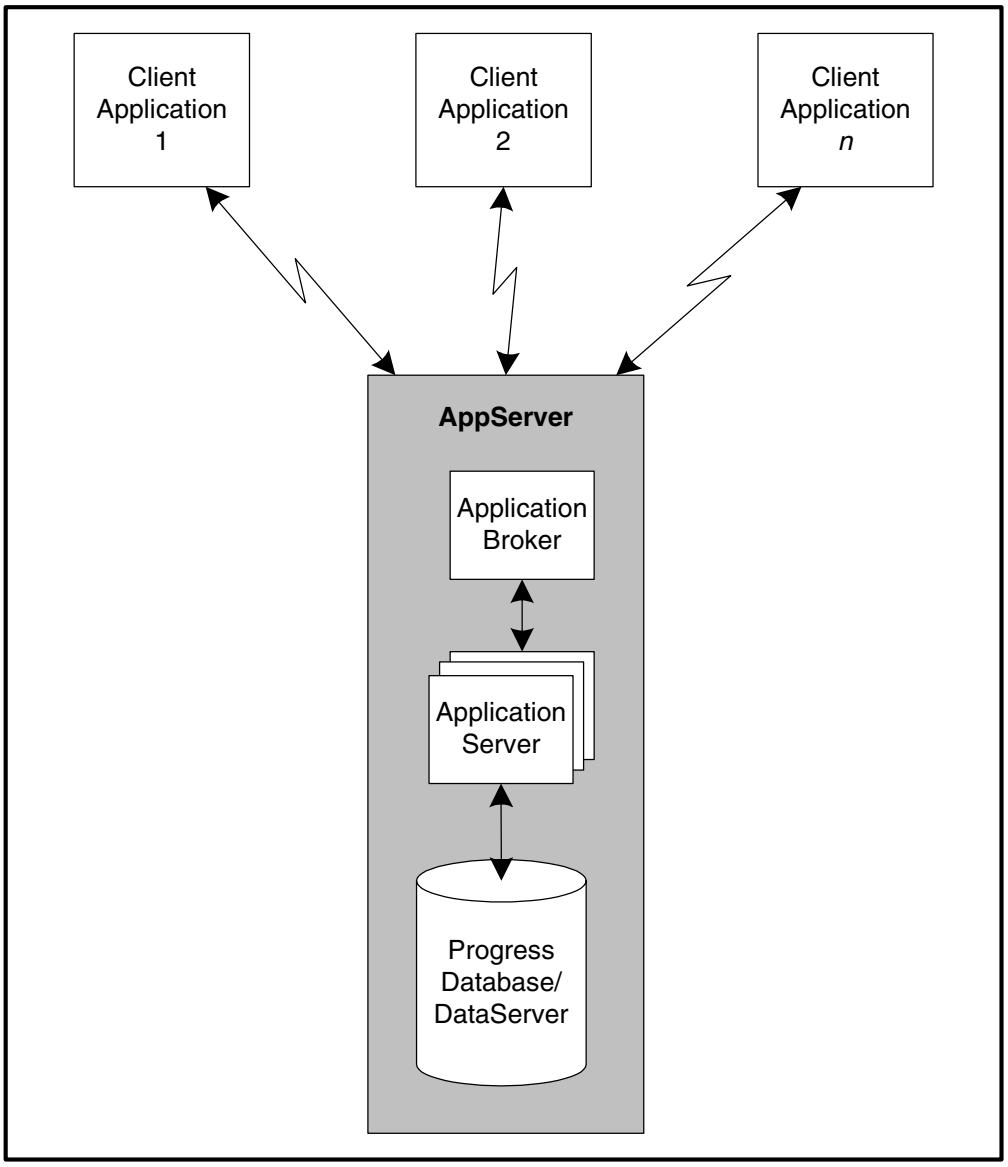


Figure 2–4: Multiple Clients Connected to an AppServer

Client Applications Connected to Multiple AppServers

A client application can connect to multiple AppServers on the same, or on different, machines. For example, a client application can be concurrently connected to an AppServer on a UNIX machine and have separate connections to two different AppServers running on Windows NT machines.

Figure 2–5 shows how this configuration might look with a client application connected to three different AppServers running (respectively) on one UNIX and two NT machines. In this scenario, the UNIX AppServer accesses a local database. Each NT AppServer accesses the same remote database server (or DataServer) running on UNIX, which thus implements a three-tier distribution of application resources.

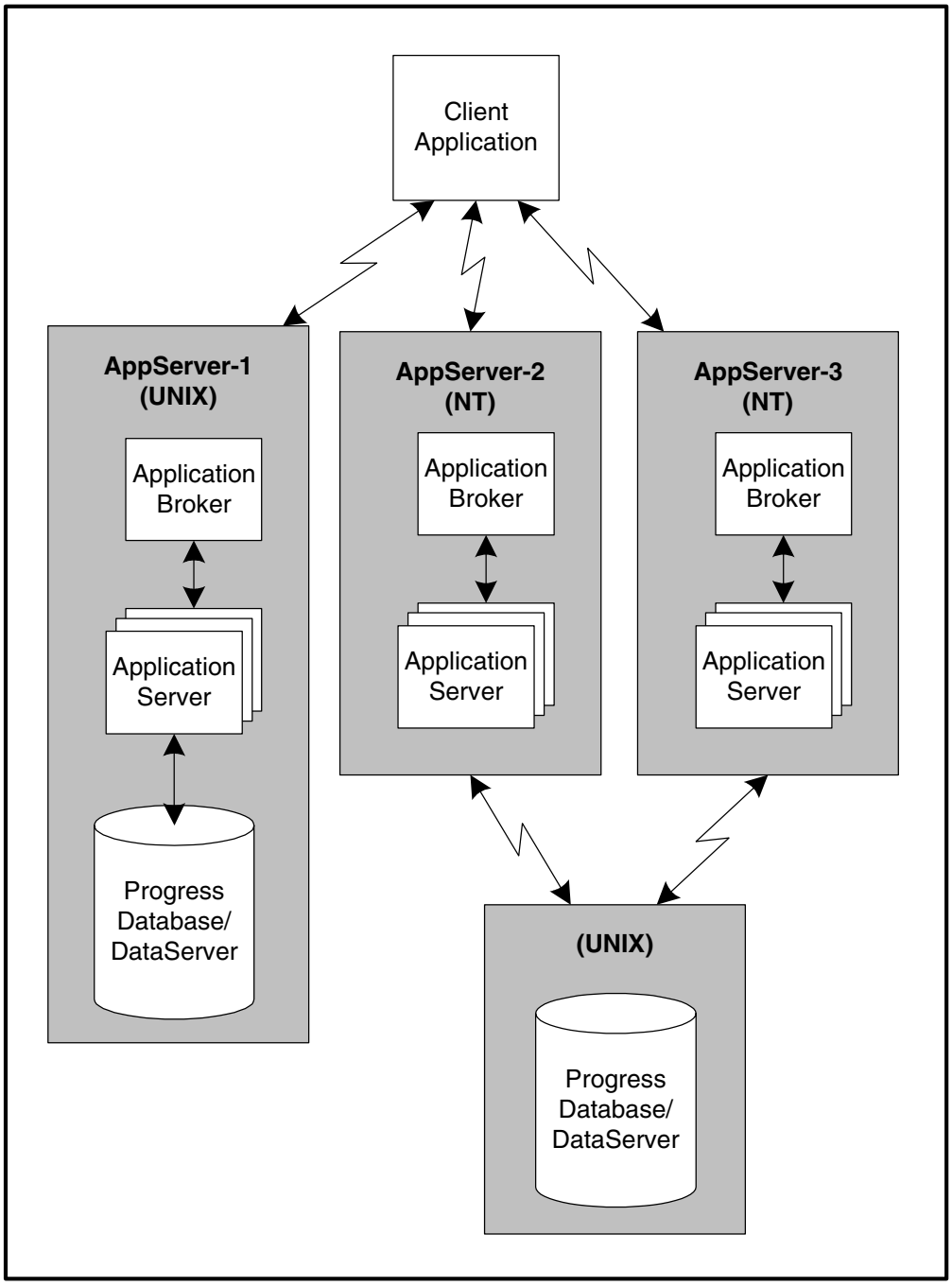


Figure 2–5: Client Connected to Multiple AppServers

AppServers Connected to One Another

Application Server processes can connect as 4GL clients to other AppServers. Figure 2–6 shows three AppServer machines (on NT and two UNIX) and one client application connected in an n-tier fashion. AppServer-1 on the NT machine accepts remote requests from the initial client application. AppServer-2 on UNIX accepts remote requests from AppServer-1, and AppServer-3 on UNIX accepts requests from AppServer-2.

Note that there is a separate AppServer session within each Application Server process and the connection that an Application Server process establishes with another AppServer is part of its AppServer session. That is, a connection that an Application Server process establishes with another AppServer is part of its session context and is not shared with any other Application Server process. For example, each Application Server process running on AppServer-2 establishes and manages, as client, its own unique connection to AppServer-3.

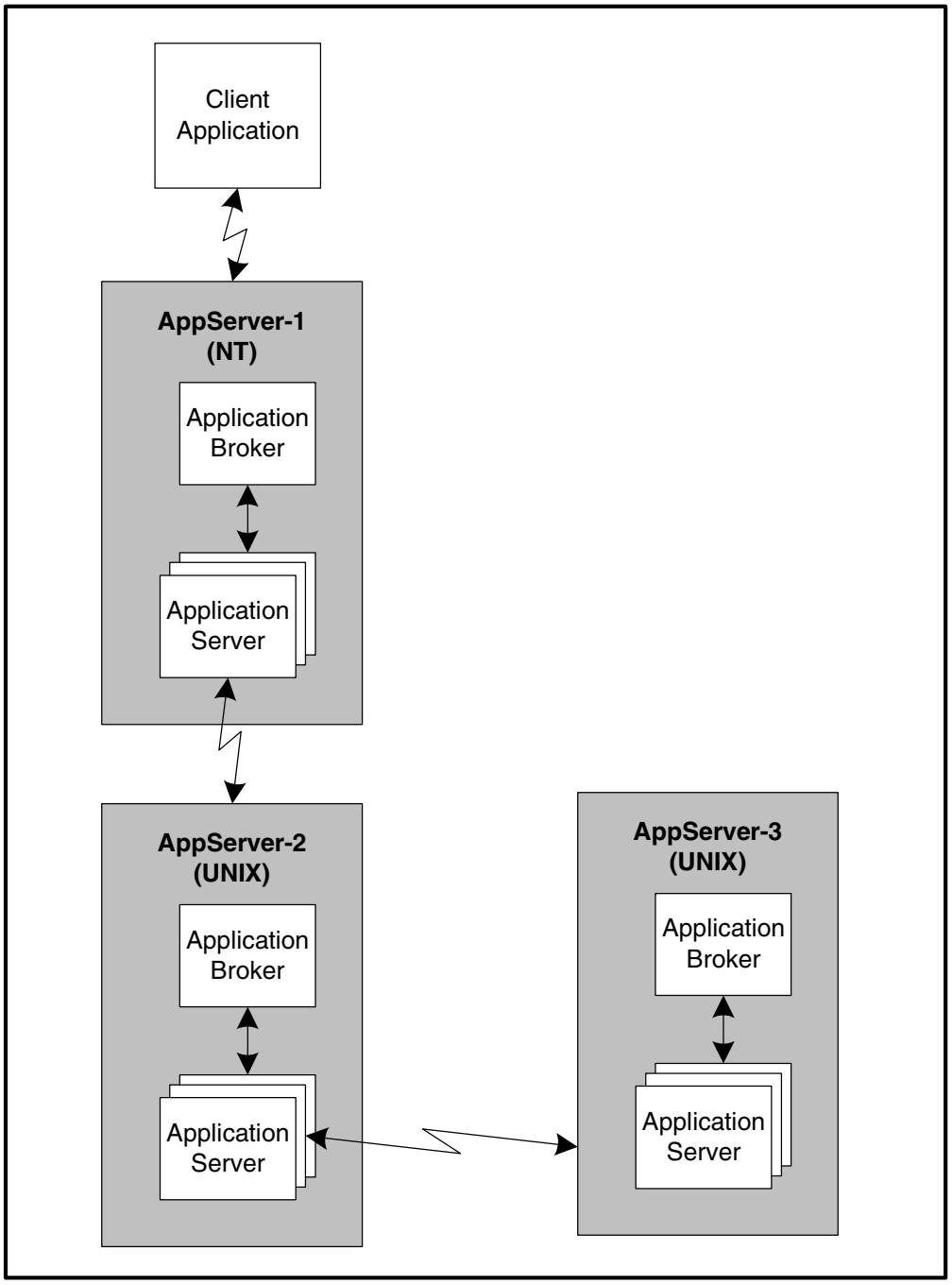


Figure 2–6: AppServer-to-AppServer Connections

2.2 Run-time Components and Operation

Figure 2–7 shows the major run-time components that comprise the AppServer architecture and their basic relationships. For more information on the complete architecture of an AppServer installation, including AppServer configuration and management, see Chapter 3, "Administration."

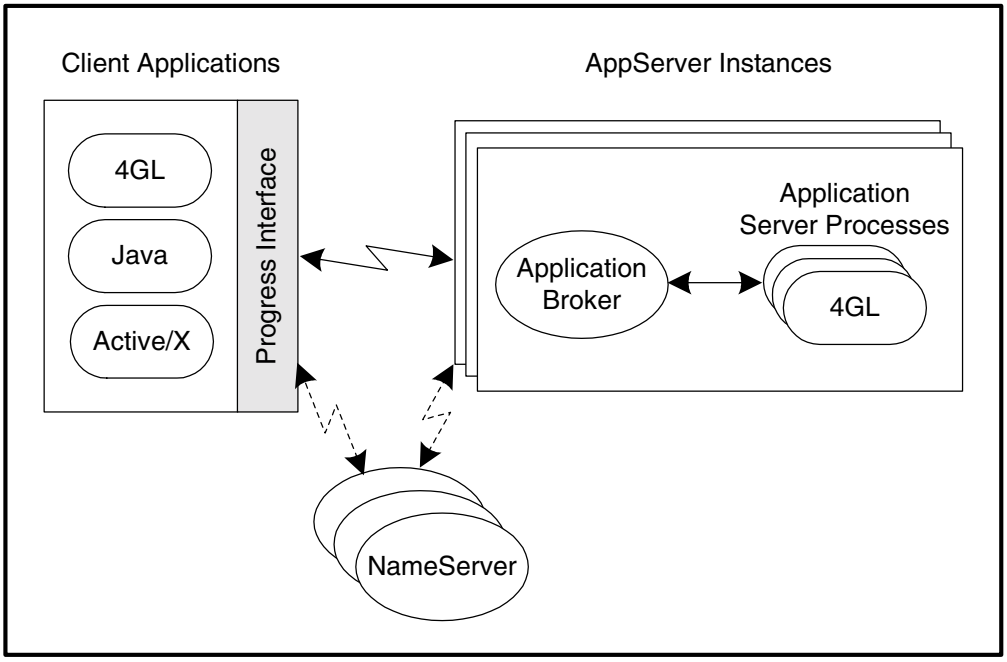


Figure 2–7: AppServer Run-time Components

The dotted arrows indicate communications required to establish a connection between client applications and the AppServer. Table 2–1 identifies and describes the components shown in Figure 2–7.

Table 2–1: Progress AppServer Run-time Components *(1 of 2)*

Component	Description
Client Application	A process that requests the execution of remote 4GL procedures in the context of an AppServer session. A client application can be: • A 4GL client session • A WebSpeed 4GL session (WebSpeed Agent), which runs 4GL procedures on behalf of Web clients[1] • Another AppServer session (Application Server process) • A Java application[2] • An ActiveX Controller[2] The Progress Interface is the code that allows a client application to access an AppServer. For 4GL clients, this interface is accessed through built-in 4GL statements and functions dedicated to AppServer access. For Java and ActiveX clients, this interface is accessed through a set of objects custom built to access AppServer procedures using the Progress Open Client Toolkit (available separately).
Application Server process	A process that executes remote procedure requests in the context of a Progress session. Much like a batch 4GL client, almost any 4GL statement that you can execute in an interactive 4GL client you can execute within an Application Server process. An AppServer instance typically contains multiple Application Server processes that start up when you start the AppServer. An Application Server process can also act as a 4GL client of another AppServer instance, by making its own remote procedure calls.
Application Broker	A process that creates, manages, and allocates Application Server processes for access by client applications. The Application Broker manages client connection requests and dispatches requests to Application Server processes. Exactly how it does this depends on the AppServer operating mode.[3] A single Application Broker supports one AppServer instance.

Table 2–1: Progress AppServer Run-time Components *(2 of 2)*

Component	Description
NameServer[4]	A process that directs client connection requests to an AppServer that supports a specified business function. A client indicates which AppServer instance it wants to connect to by specifying an *Application Service*[5] name that identifies the required business function. When an AppServer is started, it registers with a NameServer the Application Services that it supports. The NameServer with which an AppServer registers is the *controlling NameServer* for that AppServer instance. When a client requests a connection to an Application Service, the NameServer directs the client to the AppServer that has registered the specified Application Service. To provide fault-tolerant access to NameServers, you can configure multiple NameServer instances that contain identical information. Each such NameServer is a replica of the others, thus providing *connection-level fault-tolerance.* If you have installed the load-balancing option, multiple AppServers can also support the same Application Service, allowing the NameServer to distribute client connections among them, thus providing *server-level fault tolerance*.[6] Note that to connect an AppServer, the client only needs to know the host and port of the NameServer and the name of an Application Service. The Progress Interface contacts the specified NameServer on behalf of the client, processes the response, and directs the client to the appropriate AppServer.

1 For more information, see the *WebSpeed Developer's Guide*.

2 For more information, see the *Progress Open Client Developer's Guide*.

3 For more information, see the "Understanding AppServer Operating Modes" section in this manual.

4 The same NameServer process can also coordinate Web client access to WebSpeed Servers and 4GL client access to Progress DataServers. For more information on all NameServer features, see the *Progress Installation and Configuration Guide Version 9 for UNIX* or the *Progress Installation and Configuration Guide Version 9 for Windows*.

5 For more information on Application Services, see the "Connection Process" section in this manual.

6 For the NameServer to provide server-level fault tolerance, you must have the Progress AppServer Load Balancer installed.

2.2.1 Distribution of Components

The Application Broker for an AppServer instance and its associated Application Server processes must all execute on the same computer. Client applications, however, can run on any computer in the network that can access the computer where the Application Broker is running. The required NameServer can execute on the same computer as a client application, the same computer as an AppServer instance, or on any other computer in the network to which both Application Brokers and client applications have access. With these conditions satisfied, Progress can establish a connection between any client an AppServer. For more information on how you can distribute AppServer components on a network, see the information on machine distribution in the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

2.2.2 Fault-tolerant NameServers

You can configure NameServer such that a group of NameServers work together to resolve a client connection request. Having a group of NameServers working together to resolve the request provides fault-tolerant access to the NameServer function.

Progress provides two mechanisms that you can use to implement fault-tolerant NameServers, and you can use them independently or together:

- **NameServer replication**—Where you configure multiple NameServer instances within a single subnet on different machines to listen on the same User Datagram Protocol (UDP) port. Clients send connection requests and AppServers send registration requests to all NameServer instances using UDP broadcasting. Broadcasting allows a set of replicated NameServers to be set up as one controlling NameServer for each AppServer instance.

- **NameServer neighbors**—Where you configure multiple NameServers on machines located in one or more subnets so that an initial NameServer instance receives the client connection request. If this initial NameServer cannot resolve the request, it passes the request on to a specified list of *NameServer neighbors*. These NameServer neighbors then attempt to resolve the connection request. Each NameServer neighbor represents the controlling NameServer for a separate AppServer instance.

For more information on setting up fault-tolerant NameServers, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

2.2.3 Fault-tolerant AppServers and Load Balancing

When an AppServer registers with a NameServer, it specifies a list of Application Services. In the simplest configuration, only one AppServer instance can register a specific Application Service with the same controlling NameServer. Thus, the NameServer satisfies each request for the registered Application Service by directing the client connection to the same AppServer. However, if you install the load-balancing option, you can configure *fault-tolerant AppServers* by having multiple AppServer instances register the same Application Service with a single controlling NameServer.

NOTE: If you have not installed the load-balancing option and an AppServer tries to register the same Application Service on the same controlling NameServer of an already-registered AppServer, the AppServer attempting to register receives an error.

When a client requests an AppServer connection provided by fault-tolerant AppServers, the specified NameServer resolves the connection request using one of the registered AppServer instances that support the specified Application Service. By default, the NameServer chooses the AppServer instance at random from among the available instances that can satisfy the connection request, thus distributing connections equally.

The load-balancing option also allows you to have the NameServer distribute client connections across a set of fault-tolerant AppServers proportionately, according to a weight factor that you specify when you configure each AppServer instance. When each AppServer registers with the controlling NameServer, in addition to its location and supported Application Service list, the AppServer registers any weight factor that you specify. The NameServer uses the weight factors configured for all instances in a set of fault-tolerant AppServers to determine how to distribute client connections among them.

The values of these weight factors are arbitrary. It is the relative differences between these values that determines how the NameServer balances load. To achieve effective load balancing, you must coordinate weight factor assignments according to individual AppServer performance. Thus, the most correct weight factor for each AppServer instance depends on the relative performance (for your application) of the particular AppServer platform and its configuration compared to other AppServers that register support for the same Application Service.

Note that the NameServer is flexible enough that you can dynamically start an AppServer instance to register at any time among a set of running fault-tolerant AppServers. As an AppServer instance starts up and registers, the NameServer adds it to the list of available AppServers that support the same Application Services. The NameServer then apportions client connection requests according to the latest set of weight factors.

For more information on setting weight factors and the effects of these settings, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

2.2.4 Operating Modes

When you configure an AppServer instance, you must specify an operating mode for it. The operating mode determines how client requests are dispatched to individual Application Server processes running on the AppServer instance. Each operating mode features different performance and design trade-offs.

In previous versions of the AppServer, an Application Server process starts up and remains dedicated to handle requests from a single client connection, and when that connection terminates, the Application Server process terminates also. In Version 9, the AppServer never automatically terminates any Application Server processes that it starts, but reassigns them to different client connections in a manner determined by its operating mode.

The Progress AppServer supports the following three operating modes, in increasing order of complexity:

1. **State-reset**—All requests sent by a client connected to this AppServer go to the same Application Server process. This Application Server process remains dedicated to the same client for the life of the connection. When the client disconnects, the Application Server process resets its context to what it was at startup, removing all context created during the terminated client connection.

   NOTE: This operating mode is conceptually similar to the operation of AppServers prior to Progress Version 9, which run in only one operating mode.

2. **State-aware**—All requests sent by a client connected to this AppServer go to the same Application Server process. This Application Server process remains dedicated to the same client for the life of the connection. When the client disconnects, the Application Server process deletes any remote persistent procedures that are still active in its context. However, it maintains all other context created during the terminated client connection for access during future client connections. This context remains available until it is removed during a future client connection or the Application Server process terminates.

3. **Stateless**—An Application Server process is not dedicated to a specific client. The Application Server process can execute a request from any client that has an outstanding request to the AppServer. Because a client request can execute in an Application Server process that is different from any that executed previous requests for the same client, session context established for this client during a previous request might not be available. Because Progress does not automatically clean up any session context between requests, the session context in which a client request runs might have been established by a request from a different client.

For more information on the functional, performance, and design trade-offs among these operating modes, see the "Understanding AppServer Operating Modes" section.

2.2.5 AppServer Startup and Shutdown

You typically configure AppServer instances and NameServers to start up at system startup time. The Progress Explorer framework provides a graphical user interface and an equivalent set of command-line utilities that you can use to configure, startup, and manage NameServers and AppServer instances. For more information, see Chapter 3, "Administration."

Progress Startup Parameters

You can also configure the Application Broker to pass a set of Progress startup parameters to any Application Server processes that it starts. These startup parameters have the same effect on each Application Server process as the startup parameters used to start a Progress 4GL client. For example, one of the startup parameters you specify might be the Database (-db) parameter to indicate that all Application Server processes in the pool are to connect to a specific database when they start up.

AppServer Registration

When you start an AppServer instance, the first thing the Application Broker does is to register with the controlling NameServer that you specified during AppServer configuration. When you shut down an AppServer instance, the Application Broker immediately unregisters the AppServer with the controlling NameServer, ensuring that the NameServer no longer makes this AppServer available for client connection requests.

2.3 Connection Process

To establish an AppServer connection, a the client application must know both the network location (host and port number) of the controlling NameServer and the name of an Application Service registered with the NameServer by the AppServer. The client application requires no other knowledge about the AppServer that it connects to except the published API (the entry points and prototypes) required to access the AppServer's remote procedures. You provide this API to client programmers based on your AppServer application.

2.3.1 Application Services

An *Application Service* is the entire business function provided by an AppServer. An Application Service name is essentially an arbitrary alias for the AppServer that provides that specific function. An AppServer instance can register with more than one Application Service name to identify its function. Because the different Application Service names registered by an AppServer are simple aliases, a client application can call any remote procedure supported by the AppServer instance no matter what Application Service name the client uses to connect the AppServer.

For example, an AppServer instance that accesses an inventory database with certain 4GL procedures might support an Application Service named "Inventory". An AppServer instance that accesses an employees database with certain other 4GL procedures might support an Application Service named "Employees". On the other hand, the same AppServer instance might support both the "Inventory" and "Employees" Application Services.

Assigning Application Service Names

You specify the Application Service names that an AppServer supports when you configure an AppServer instance. Client applications use the Application Service names you specify to indicate an AppServer instance with which to connect. The NameServer then chooses a particular AppServer instance to connect based on whether it supports the Application Service specified by the client.

Fault-tolerant AppServer Support

When distributing connection requests for fault-tolerant AppServers, the controlling NameServer assumes that all AppServer instances that support the same Application Service also support the same API. The NameServer does not validate this assumption. You must ensure that the same published API is available on all AppServer instances that register the same Application Service name with a NameServer. For more information on fault-tolerant AppServers, see the "Fault-tolerant AppServers and Load Balancing" section.

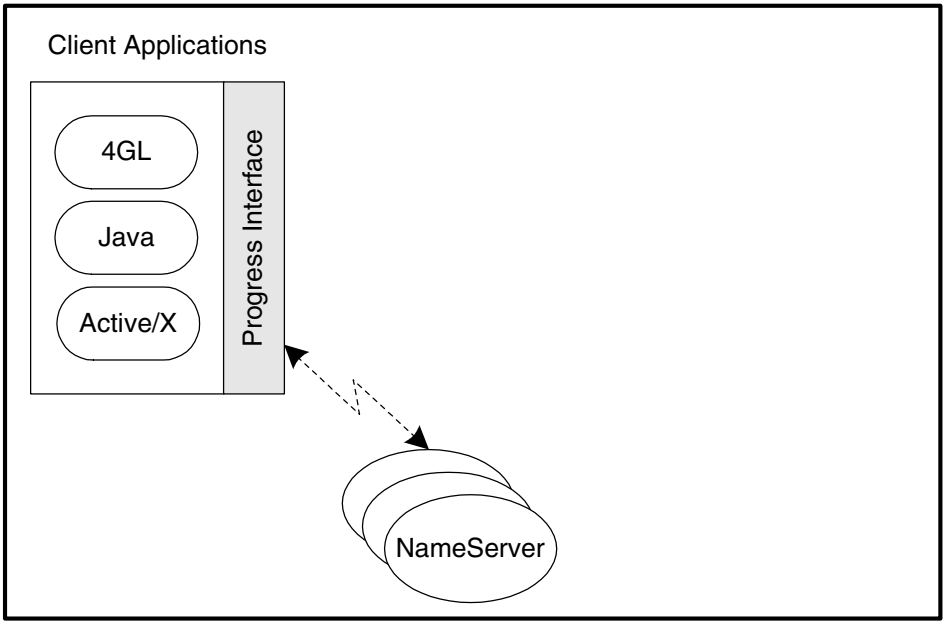
Default Service

A client application can connect to an AppServer without specifying an Application Service. In this case, the NameServer uses whatever AppServer registers itself as the *default service*. You can specify any AppServer instance as the default service during AppServer configuration. However, Progress Software Corporation recommends that you avoid relying on the default service and provide explicit Application Service names for access by all client applications. For more information on AppServer configuration, see Chapter 3, "Administration."

2.3.2 Connection Operation

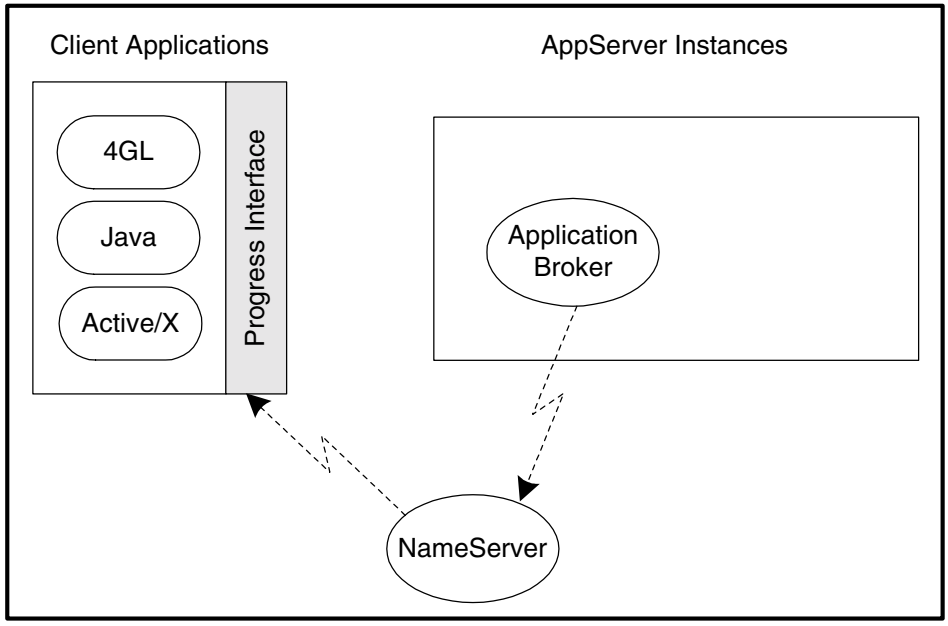
The way an AppServer connection works depends partly on the AppServer operating mode. For more information on operating modes, see the "Understanding AppServer Operating Modes" section. In general, a client application initiates the connection with an AppServer and manages the connection following these steps:

1. The client application notifies the Progress Interface to send a connection request for a specified Application Service to a specified NameServer. The Progress Interface then sends the request to the NameServer, which the Progress Interface identifies using a host and port provided by the client application:

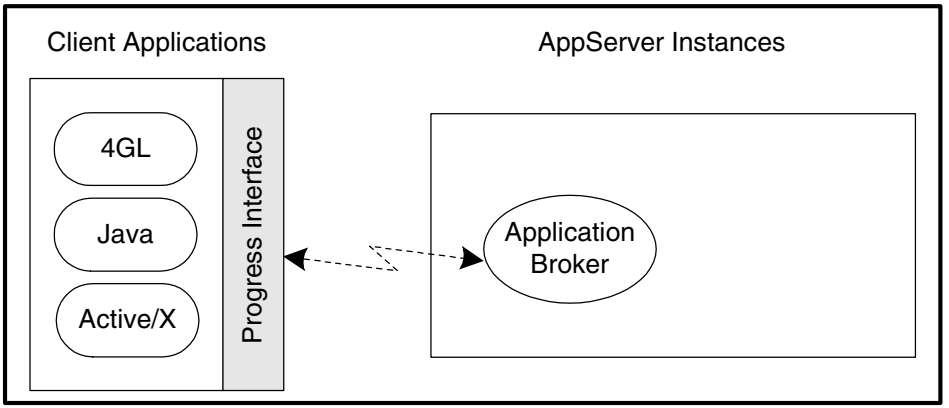


If you have configured fault-tolerant NameServers, the replicated or neighbor NameServers attempt to resolve the connection. If no such NameServer can resolve the connection, the client receives a connection error.

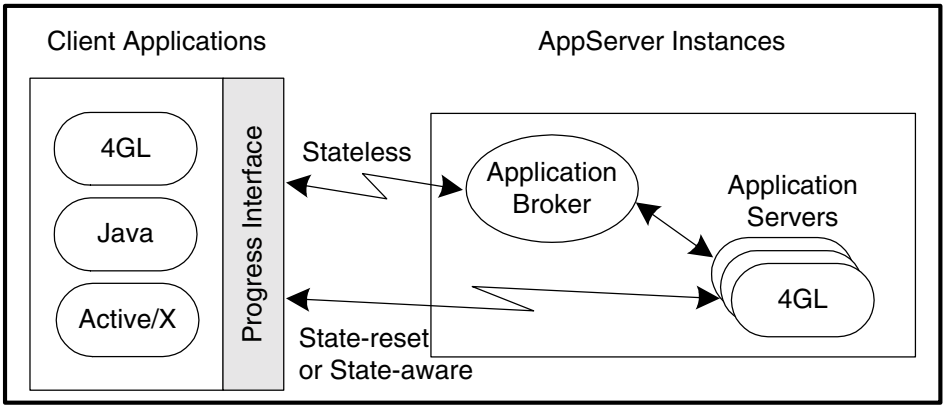
2. The resolving NameServer chooses a particular AppServer instance that supports the specified Application Service (and that balances client load, if so configured) and provides the Progress Interface with the network address of the Application Broker for that AppServer:



3. The Progress Interface then sets up a connection with the Application Broker:



4. The Application Broker either maintains the connection directly between itself and the client or provides the Progress Interface with the address of a particular Application Server process to which it can connect. Whether the Application Broker maintains the connection or hands the connection off to an Application Server process depends on the operating mode:



5. Once the connection is established, all remote procedure requests from that client are sent over this connection.

6. When the client no longer needs the connection, it disconnects from the AppServer by notifying the Progress Interface to send a disconnect request.

For more information on how to set up and work with a connection, see Chapter 5, "Programming the Client Application."

2.4 Understanding AppServer Operating Modes

When you configure an AppServer, you specify an operating mode that indicates how the Application Broker manages its Application Server process pool and assigns client requests to Application Server processes within the pool. The choice of operating mode has a major affect on the performance and design of your AppServer application (the procedures that run on the AppServer).

Any 4GL procedure can determine the operating mode of the AppServer it is running on by reading the value of the SERVER-OPERATING-MODE attribute on the SESSION system handle. This is a CHARACTER parameter that returns the name of the operating mode as a string.

The following sections describe these operating modes in detail.

2.4.1 State-reset Operating Mode

When Progress sends a client application's connection request to an AppServer that is running in state-reset operating mode, the Application Broker connects the client application directly to one of the processes in its Application Server process pool. The connected Application Server process remains dedicated to the client application for the life of the connection.

Between connections, Progress re-initializes the AppServer session to its initial state. This is the state it is in immediately after the Application Server process is started and after all startup parameters are processed.

NOTE: State-reset operating mode is conceptually similar to the operation of AppServers prior to Progress Version 9.

Connection Management

When the AppServer receives a connection request, the Application Broker chooses an Application Server process that is not currently dedicated to some other client application's connection. If all Application Server processes are unavailable, the Application Broker might create another Application Server process for the connection, or it might return an error. The result depends on how many Application Server processes are running and the values of various configuration settings.

When the client application disconnects from its Application Server process, the Application Server process notifies the Application Broker that it is now available to service another client application connection.

NOTE: In AppServers prior to Progress Version 9, the Application Broker terminates the disconnected Application Server process.

To help an AppServer session perform common connection tasks, such as user authentication, you can configure a Connect procedure that runs for each connection request and a Disconnect procedure that runs for each disconnection request. For more information on these procedures, see Chapter 4, "Programming the AppServer."

Performance Considerations

Because an AppServer running in state-reset operating mode dedicates an Application Server process to each client connection, the connected client is guaranteed of having each request handled immediately by the AppServer. Assuming that the general process load on the AppServer platform is light enough, this means that the client can expect relatively fast response time from the AppServer.

However, throughput in a state-reset connection tends to drop as client load increases, because the number of dedicated Application Server processes available to handle client connections is necessarily limited by the number of Application Server processes that can be started on the AppServer machine.

Context Management

Because each Application Server process is dedicated to a client connection, the AppServer session maintains all session context between client requests. Thus, for example, any global values or database connections that you set in remote procedures remain available for the life of the connection or until you change them during some remote procedure call.

In between connections, an AppServer session running in state-reset operating mode resets its context to the initial state. This initial state is its context after the Application Server process first starts and after all Application Server process startup parameters take effect, including databases connections.

2.4.2 State-aware Operating Mode

When Progress sends a client application's connection request to an AppServer that is running in state-aware operating mode, the Application Broker connects the client application directly to one of the processes in its Application Server process pool. The connected Application Server process remains dedicated to the client application for the life of the connection.

Unlike state-reset, Progress does not reset the AppServer session context between connections. Whatever context remains when one connection finishes is available to the next connection.

Connection Management

When the AppServer receives a connection request, the Application Broker chooses an Application Server process that is not currently dedicated to some other client application's connection. If all Application Server processes are unavailable, the Application Broker might

create another Application Server process for the connection, or it might return an error. The result depends on how many Application Server processes are running and the values of various configuration settings.

When the client application disconnects from its Application Server process, the Application Server process notifies the Application Broker that it is now available to service another client application connection.

To help an AppServer session perform common connection tasks, such as user authentication, you can configure a Connect procedure that runs for each connection request and a Disconnect procedure that runs for each disconnection request. For more information on these procedures, see Chapter 4, "Programming the AppServer."

Performance Considerations

Because an AppServer running in state-aware operating mode dedicates an Application Server process to each client connection, the connected client is guaranteed of having each request handled immediately by the AppServer. Assuming that the general process load on the AppServer platform is light enough, this means that the client can expect relatively fast response time from the AppServer.

However, throughput in a state-aware connection tends to drop as client load increases, because the number of dedicated Application Server processes available to handle client connections is necessarily limited by the number of Application Server processes that can be started on the AppServer machine.

Context Management

Unlike state-reset, an AppServer running in state-aware operating mode does not reset its session context between client connections. Instead, Progress maintains any context created in the AppServer session until you specifically delete it in some procedure that you write.

For an AppServer running in state-aware operating mode you can also configure Startup and Shutdown procedures that run in every AppServer session when it starts up and shuts down. You can use these procedures to establish and discard run-time context that the AppServer maintains in common for all connections. For more information on the Startup and Shutdown procedures, see Chapter 4, "Programming the AppServer."

Like state-reset, because each Application Server process is dedicated to a client connection, the AppServer maintains all session context between client requests. Thus, for example, any global values or database connections that you set in remote procedures remain available for the life of the connection or until you change them during some remote procedure call.

As noted in a previous section, an AppServer session running in state-aware operating mode maintains most of its context between client connections. However, it does delete any remote persistent procedures that are still active when the connection terminates.

2.4.3 Stateless Operating Mode

Unlike state-reset or state-aware, an AppServer running in stateless operating mode does not dedicate any Application Server process to a client connection. Instead, all Application Server processes remain generally available to execute remote procedure requests from all connected client applications.

When a client connects to an AppServer that is running in stateless operating mode, the Application Broker maintains the connection between itself and the client. The Application Broker can then dispatch each remote procedure request from the client to any available Application Server process to execute. Because any Application Server process that executed a previous request from this client might not be available, the Application Server process that handles subsequent requests might be different.

NOTE: The name of this operating mode ("stateless") refers to the lack of context that AppServer sessions automatically maintain between client requests. However, stateless operating mode does support additional mechanisms that you can use to maintain some context between requests, as described in the following sections.

Connection Management

By default, all connections on a stateless AppServer are *unbound*—that is, all Application Server processes are available to handle requests from any client connection. As long as an Application Server process is not currently handling a client request, it can handle requests from any connected client. However, at any time during the handling of a client request, a client connection can become *bound*—that is, the Application Server process handling the request becomes available to handle requests only from the client it is currently servicing. While a connection is bound, all requests from the connected client are handled by the same Application Server process. In the same way it became bound, at any time during the handling of a client request, a bound connection can become unbound, at which point requests from the connected client can be handled by any available Application Server process.

In general, then, a stateless Application Server process is available to handle a client request as long as it is not currently processing a client request and as long as it is not waiting on a bound connection. If a stateless AppServer has no Application Server processes available to handle requests, all requests are queued at the Application Broker until an Application Server process becomes available.

To help an AppServer session perform common connection tasks, such as user authentication, you can configure a Connect procedure that runs for each connection request and a Disconnect procedure that runs for each disconnection request. For more information on these procedures, see Chapter 4, "Programming the AppServer."

A connection in the unbound state can transition to the bound state in two ways:

- A client application instantiates a remote persistent procedure in the context of the connection. The Application Server process that handles the remote persistent procedure request thus becomes bound to the connection.
- The Application Server process runs a procedure that sets the SERVER-CONNECTION-BOUND-REQUEST attribute on the SESSION handle to TRUE. The Application Server process that sets this attributes thus becomes bound in the connection.

Note that it is the successful instantiation of a remote persistent procedure that forces the connection to transition from the unbound state to the bound state. If a client fails in its attempt to instantiate a remote persistent procedure on an unbound connection, the connection remains unbound.

For more information on managing connection states on a stateless AppServer, see Chapter 4, "Programming the AppServer."

Performance Considerations

Because a stateless AppServer passes requests from connected clients to any available Application Server process, the number of clients that a single AppServer instance can handle using this operating mode is relatively high. That is, a relatively small number of Application Server processes can support many client connections. If you never set connections to the bound state, a stateless AppServer can make maximum use of its available Application Server process resources to handle client requests. This operating mode thus maximizes throughput for large numbers of clients, especially when they maintain short connections.

Context Management

An AppServer running in stateless operating mode does not reset its session context between client connections. Instead, it maintains most all context created in an AppServer session until a particular Application Server process terminates or the context is changed by some procedure running in the AppServer session.

For an AppServer running in stateless operating mode you can also configure Startup and Shutdown procedures that run in every AppServer session when it starts up and shuts down. You can use these procedures to establish and discard run-time context that the AppServer maintains in common for all connections. For more information on the Startup and Shutdown procedures, see Chapter 4, "Programming the AppServer."

Using a bound connection, the AppServer maintains all session context created between client requests (similar to a state-aware AppServer), because the bound connection maintains one Application Server process to execute all requests for the same connected client until you reset the connection to the unbound state.

Using an unbound connection, the AppServer does not automatically guarantee that context created during a remote procedure request will be available when the same client makes the next request. As noted in a previous section, this is because a stateless AppServer cannot guarantee that the same Application Server process will handle the next request. However, you can still maintain context created between requests from a connected client, but only by explicitly saving the created context using one of several mechanisms. Two such mechanisms specially supported for stateless AppServers include:

- The SERVER-CONNECTION-CONTEXT attribute on the SESSION handle that allows you to pass CHARACTER data from one request to another on the same client connection.
- Activate and Deactivate procedures that you can configure to execute before and after each remote procedure request.

For more information on these mechanisms, see Chapter 4, "Programming the AppServer."

As noted in a previous section Progress does not automatically delete any AppServer session context between requests to a stateless AppServer. The only exception to this is if a client application disconnects from a stateless AppServer while it has remote persistent procedures active in the AppServer session. In that case, Progress does automatically delete the active remote persistent procedures before disconnecting the client application.

2.5 Understanding Synchronous and Asynchronous Requests

The 4GL client and AppServer support two types of remote procedure execution:

- **Synchronous request**—(Default) Where the client blocks and waits for the result of the remote request before continuing execution.
- **Asynchronous request**—Where the client continues execution after initiating the request and processes the result whenever the AppServer makes it available.

NOTE: Progress does not provide built-in support for asynchronous requests in Open Clients. You can implement your own asynchronous request mechanism in the Java Open Client using threads. For more information on threads in Java Open Clients, see the *Progress Open Client Developer's Guide*.

2.5.1 Process Control Flow

Figure 2–8 compares the process flow within an application using synchronous and asynchronous requests.

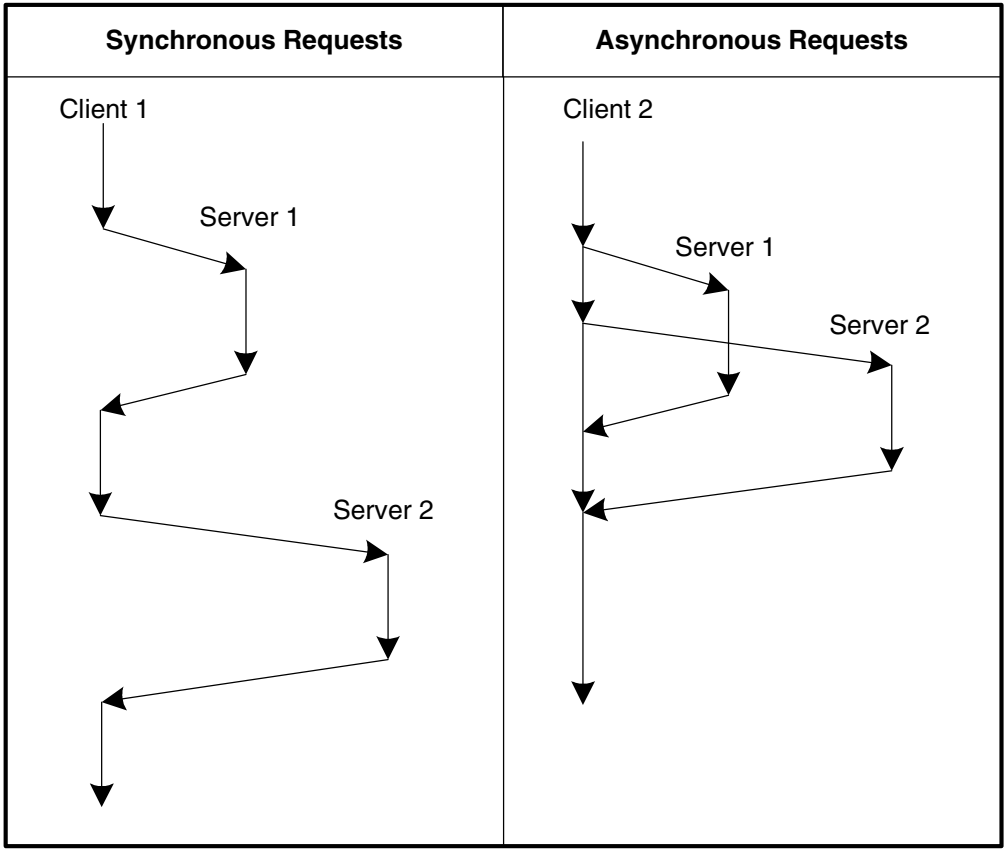


Figure 2–8: Synchronous and Asynchronous Requests

In this figure, one client (Client1) makes synchronous requests and another client (Client2) makes asynchronous requests to two AppServers (Server1 and Server2). For Client1, control flows in sequence between client and AppServer. For Client2, control flows in parallel between client and AppServer. Assuming that the clients and AppServers perform the same work, Client2 can conceivably complete the work in less time than can Client1. The advantage for Client2 comes with the parallelism that asynchronous requests can achieve. In this case, the actual advantage that Client2 affords depends on its ability to respond to the results from the AppServers at the moment they arrive, while doing its own share of the work until the results actually do arrive.

2.5.2 Specifying the Type of Remote Request

The type of remote request (synchronous or asynchronous) is determined by the 4GL procedure that initiates it, using options on the RUN statement. The AppServer can handle both types of requests in the same AppServer session and over the same client connection.

You can initiate as many asynchronous requests as you want at any point during client execution. You can also mix synchronous and asynchronous requests in the same application. The only constraint is that you can only initiate a synchronous request if there are no pending (incomplete) asynchronous requests over the same AppServer connection. Progress also provides a mechanism for you to check the status of asynchronous requests on a connection.

NOTE: As suggested earlier, the Application Server process that executes a request does not know if the request is synchronous or asynchronous. The Application Broker manages each request on behalf of the client, and dispatches the request for execution according to the AppServer operating mode (state-reset, state-aware, or stateless). Consequently, the client can execute synchronous and asynchronous requests without regard to the AppServer operating mode.

2.5.3 Handling Asynchronous Request Results

Progress supports the handling of asynchronous request results using an event procedure that executes in response to a PROCEDURE-COMPLETE event. As for other Progress events, this event is handled in the context of the PROCESS EVENTS statement or a blocking I/O statement (such as WAIT-FOR). The event procedure returns any results from the request to its own context block using appropriate input parameters that you define according to your application requirements.

2.5.4 Asynchronous Request Queuing

The AppServer can execute only one request at a time over a single connection. To allow a client to submit a sequence of asynchronous requests on the same AppServer connection, Progress maintains a queuing system to manage such requests between the client and the AppServer.

Queuing System Components

The asynchronous request queuing system includes three queues:

- One *send queue* per AppServer connection, for queuing asynchronous requests to the AppServer.
- One *response queue* per AppServer connection, for queuing the results of each asynchronous request.

- The Progress *event queue* on the client for handling all Progress events, including user-interface and other events, such as PROCEDURE-COMPLETE. This event queue handles the PROCEDURE-COMPLETE event for all AppServer connections on a client.

Understanding the Queuing Model

For a given AppServer connection, each asynchronous request is executed in the order that it is submitted, and the event procedures for those requests are executed in the same order that the requests are executed. This is order is maintained by the send and response queues associated with the connection.

The send queue queues requests that are submitted for execution by the client. The other queue is the response queue. The response queue queues responses received on a given AppServer connection for asynchronous requests that have completed execution, but whose event procedure has not yet been executed. Figure 2–9 shows how this works, where **1a** is the first asynchronous request (AsyncRequest **1a**) to AppServer 1 and **2a** is the first asynchronous request (AsyncRequest **2a**) to AppServer 2, and so forth.

In Figure 2–9, AsyncRequest refers to the execution of an asynchronous RUN statement and EventProc to the execution of an event procedure. These are further identified by 1 for the AppServer 1 connection and 2 for the AppServer 2 connection, and a letter indicating the order of the request on the client. On a given connection, if an asynchronous request is submitted for execution when there is already a request executing on the AppServer, the new asynchronous request is queued on the send queue until all previously submitted asynchronous requests for that connection have executed. Such is the case for AsyncRequests **1d** and **2e**. The new asynchronous request is only sent to the AppServer for execution once all prior asynchronous requests have completed execution on the AppServer. In Figure 2–9, AsyncRequests **1b** and **2c** are currently executing on their respective AppServers.

As each request completes execution, the response to the request is queued on the appropriate response queue and the next request in the send queue is sent to the AppServer. In Figure 2–9, AsyncRequests **1c** and **2d** are next in line for execution. Each response remains in the response queue until the corresponding event procedure can be executed in the context of a PROCESS EVENTS or blocking I/O statement. In Figure 2–9, the PROCEDURE-COMPLETE events for AsyncRequests **1a**, **2a**, and **2b** are all pending execution of their respective event procedures.

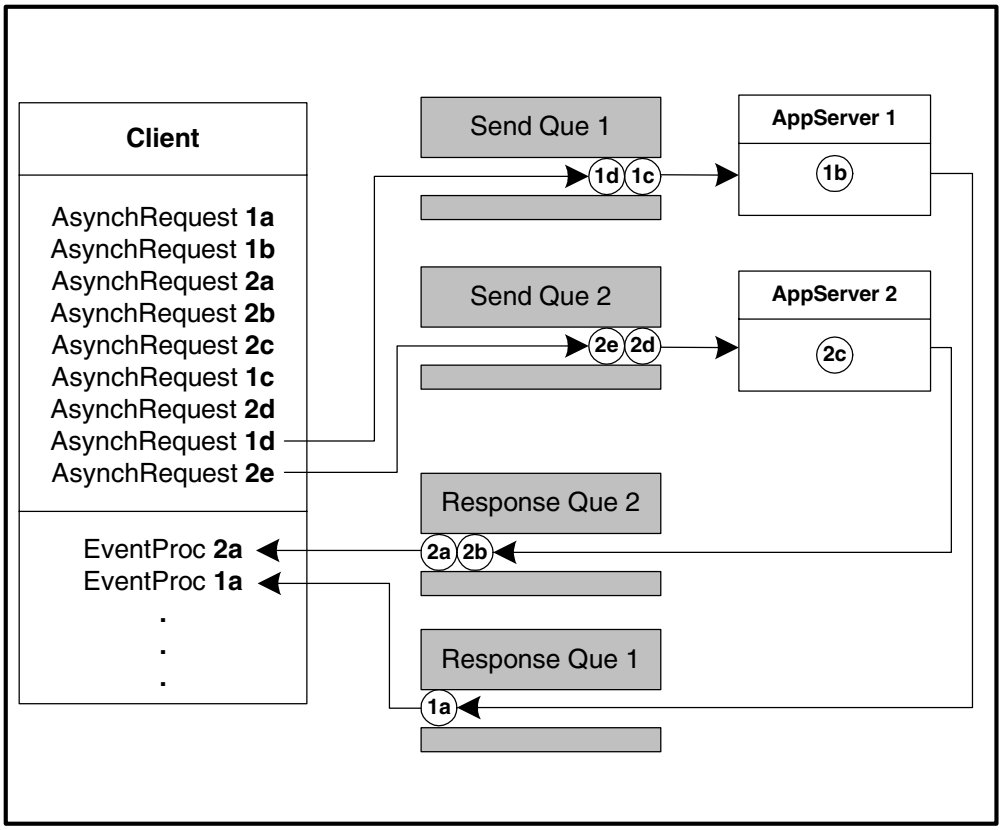


Figure 2–9: Asynchronous Request Queuing

The response to an asynchronous request can be received by the client application at any time, but the event procedure associated with that request can be run only if the application is executing a PROCESS EVENTS or blocking I/O statement. Note that the order in which the client handles pending responses for different AppServer connections is undefined. In Figure 2–9, either the event procedure for AsyncRequest **1a** or AsyncRequest **2a** can execute first.

Thus, Progress guarantees that the responses from asynchronous requests on the same AppServer connection are handled in order of their execution on the client. However, it does not guarantee the order in which responses are handled from multiple AppServer connections.

Tuning the Size of the Send and Response Queues

For each connection, Progress allocates the memory for the send and response queues with a default size. If asynchronous requests are such that they exceed the send and response queue size, Progress stores additional queued requests in a temporary table associated with the client

session. Thus, performance can vary based on whether requests are stored in memory or in the temporary table.

To help you maximize the performance of asynchronous requests, Progress provides the Async Queue Size (-asyncqueuesize) client startup parameter. This parameter allows you to specify the the number of bytes that you want Progress to allocate for the send and response queues on an AppServer connection.

The Async Queue Size parameter only applies when you create an AppServer connection on the client, and it applies equally to each such connection. Also, although Progress creates the send and response queues for each connection, they are not used unless you make asynchronous requests to the AppServer. Thus, you can maximize AppServer performance by choosing a value based on the amount of data that you expect to flow between the client and the AppServer for these asynchronous requests. Similarly, if you are making no asynchronous requests, you can choose a low value to reduce the space allocated to queues that you are not planning to use.

For more information on this startup parameter, see the Async Queue Size (-asyncqueuesize) entry in the *Progress Startup Command and Parameter Reference*.

3

Administration

This chapter describes the tasks required to configure, start up, shut down, and maintain the Progress AppServer. The Progress AppServer operates on the UNIX and NT platforms, and you can perform the required tasks from either the UNIX or NT command lines or from the Progress Explorer running on a Windows workstation.

Administration includes common tasks for configuring all Progress Server products that use the NameServer (Progress Unified Broker products), including the AppServer, WebSpeed Server, and DataServers. For an overview of these common tasks, including detailed information on NameServer configuration, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

The sections of this chapter describe:

- An overview of the AppServer administration framework
- Configuring AppServer components
- Setting up the environment for AppServer execution
- Starting and managing an AppServer instance
- Setting AppServer pool parameters
- Managing code pages for AppServers and clients
- A summary of management tasks

3.1 AppServer Administration Framework

The management functions for the Progress AppServer are an integrated part of the Progress Explorer administration framework. Figure 3–1 shows an overview of this framework as it applies to AppServer administration.

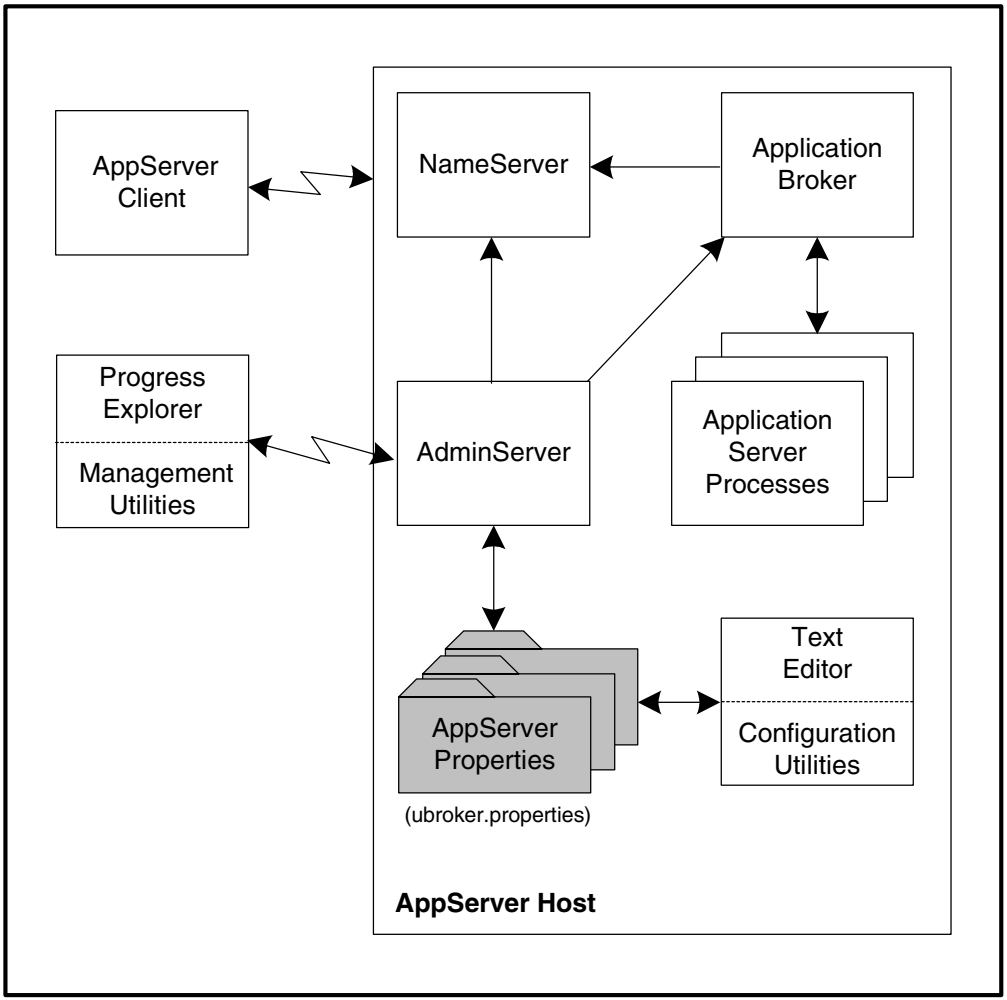


Figure 3–1: AppServer Administration Framework

Figure 3–1 shows the NameServer and AppServer components running on the same machine. You can also install the NameServer and AppServer on separate machines connected to the same network. In that case, a separate AdminServer and `ubroker.properties` file exist on each machine for access by the Progress Explorer.

For more information on distributed AppServer configurations, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

The core of the Progress Explorer administration framework is the *AdminServer* process, which resides on each machine where a Progress server product is installed. The AdminServer performs the actual configuration and management of these products within the Progress Explorer framework.

In addition to the AdminServer process, the AppServer administration framework consists of several supporting components as shown in Figure 3–1:

- AppServer clients
- Progress Explorer and management utilities
- NameServer
- Application Broker
- Application Server processes
- The AppServer and NameServer properties file (`ubroker.properties`)
- Text editor and configuration validation utilities

For more information on how AppServer clients, NameServers, Application Brokers, and Application Server processes work together, see Chapter 2, "Overview of the Progress AppServer." Administration of these components follows the general procedures outlined in this section using the Progress Explorer, the `ubroker.properties` file, and the supporting utilities. For an overview of this configuration procedure and detailed information on configuring NameServers, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

3.1.1 AppServer Clients

This chapter describes configuration and startup of the AppServer, itself. For information on configuring and starting up the various AppServer clients, see the documentation for your client product.

For Progress 4GL clients, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*, and the following manuals:

- Basic 4GL client sessions—*Progress Client Deployment Guide*
- Application Server processes as client sessions of other AppServers—This manual
- WebSpeed Agents—*WebSpeed Installation and Configuration Guide*

For information on configuring and deploying Java and ActiveX Open Clients, see the *Progress Open Client Developer's Guide*.

3.1.2 Progress Explorer and Management Utilities

Progress provides two ways to manage local or remote AppServers, using the:

- Progress Explorer tool
- Command-line management utilities

Progress Explorer Tool

Progress Explorer is a graphical user interface that provides an easy way for you to create and manage Progress AppServers. The Progress Explorer tool runs as a Windows client and works with the Progress AdminServer in a client/server framework to manage AppServers and their controlling NameServers.

For an introduction to using the Progress Explorer tool and information on creating and managing NameServers, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*. For information on configuring an AppServer using the Progress Explorer tool, see the "Configuring an AppServer Using the Progress Explorer" section in this chapter. For information on starting and managing an AppServer, see the "Starting and Managing an AppServer with the Progress Explorer" section.

Command-line Management Utilities

The command-line management utilities run on both Windows and UNIX platforms, and allow you to manage existing AppServer configurations. The management utilities for the AppServer include the:

- **NSMAN utility**—To manage NameServers
- **ASBMAN utility**—To manage AppServers

Like the Progress Explorer tool, the command-line management utilities work with the Progress AdminServer in a client/server framework to manage AppServers and their controlling NameServers. Unlike the Progress Explorer tool, these utilities do not create a new AppServer configuration. Without the Progress Explorer, you must use a text editor and the configuration utilities. For more information, see the "Text Editor and Configuration Utilities" section.

This chapter describes how to manage an AppServer using the Progress Explorer and the ASBMAN utility. For information on managing a NameServer using the NSMAN utility, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

Using these utilities, you can locally or remotely:

- Start and stop an AppServer
- Manage Application Server processes
- Monitor the status of an AppServer

For information on starting and managing an AppServer using the management utilities, see the "Starting and Managing an AppServer with the Management Utilities" section.

3.1.3 NameServers

The NameServer is an essential part of AppServer configuration and management. Its features support the following AppServer capabilities:

- *Location transparency* for client connections
- *Load balancing* for multiple simultaneous client connections (optional)
- *Server-level fault tolerance* for client requests (optional)

For an overview of these features, see Chapter 2, "Overview of the Progress AppServer." For detailed information on configuring and managing NameServers to support all of these features, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*. This chapter describes when and where to use NameServer instances as part of AppServer configuration and management.

3.1.4 Application Broker

The Application Broker:

- Registers with the controlling NameServer the Application Services that the AppServer provides. For more information on Application Services, see Chapter 2, "Overview of the Progress AppServer."
- Manages connections between clients and a pool of Application Server processes that it starts.
- Maintains the status of each Application Server process in its pool and scales the number of processes according to changing demand.
- When configured for stateless operating mode, dispatches remote requests to Application Server processes.

One Application Broker provides connection management for a single AppServer instance. However, you can configure multiple AppServer instances, possibly using different operating modes and accessing different resources, for a single AppServer installation. For information on broker configuration options, see the "Configuring AppServer Components" section.

3.1.5 Application Server Processes

An Application Server process is a Progress session within the AppServer that executes 4GL procedures on behalf of AppServer clients. The Application Broker manages a pool of Application Server processes for an AppServer instance. Each Application Server process in the pool runs an identical set of 4GL procedures that share the same Progress resources. The Application Services that an AppServer supports are really aliases for the same set of remote procedures supported by all Application Server processes in the pool.

When you configure Application Server processes, you specify a single set of properties that apply in common to all processes in the pool. For more information on configuring Application Server processes, see the "Configuring AppServer Components" section.

3.1.6 AppServer and NameServer Properties File (ubroker.properties)

The properties file (`ubroker.properties`) stores the component configurations of all Progress Unified Broker products. For AppServer configurations, these components include the AppServer and the NameServer. There is one copy of this file residing in *`ProgressInstallDir`*`/properties` on each machine where Progress is installed. So, if you install the NameServer on a separate machine, it has its own copy of the `ubroker.properties` file.

For general information on the `ubroker.properties` file and more specific information on AppServer properties, see the "Configuring an AppServer Using the Properties File" section. For information on NameServer properties, see the sections on configuring the NameServer using the properties file in the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

3.1.7 Text Editor and Configuration Utilities

You can update the `ubroker.properties` file manually by editing it using any text editor. This is the only method available to update NameServer and AppServer configurations on a UNIX platform. However, you can also manually edit this file on Windows platforms.

In general, it is recommended that you update all UNIX configurations remotely using the Progress Explorer GUI. If you must update a UNIX AppServer configuration locally, make a copy of the file, update the copy, then verify the result. Progress provides a set of configuration validation utilities that you can use to verify the correctness of any changes that you make to this file.

You can use the:

- **NSCONFIG utility**—To validate NameServer configurations. For more information on this utility, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.
- **ASCONFIG utility**—To validate AppServer configurations. For more information on the this utility and on updating the `ubroker.properties` file manually, see the "Configuring an AppServer Using the Properties File" section.

3.1.8 Preparing for AppServer Administration

There are two preliminary tasks before you can begin the configuration and operation of an AppServer installation:

- Install the necessary product components. Typically, this involves installing Progress and the AppServer on one or more network machines. If you plan to configure fault-tolerant servers or use load balancing, you must install the Load Balancing option.
- Configure and set up the machines involved in the Progress installation. Typically, you have completed any required network configuration for all machines prior to installing Progress. After installation, you must also set up each machine environment to run Progress and the AppServer.

For more information on Progress installation and setup, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*. For more information on the distribution of resources in the AppServer environment, see the information on machine distribution for Unified Broker products in these same manuals.

3.2 Configuring AppServer Components

This section describes how to configure an AppServer. It first describes AppServer configuration using the Progress Explorer, then describes how to configure an AppServer by editing the `ubroker.properties` file.

3.2.1 Using the Progress Explorer to Configure an AppServer Instance

To access the Progress Explorer tool, you can open it from the Progress program group or in the Microsoft Management Console (MMC) on your Windows system. For more information, see the *Progress Installation and Configuration Guide Version 9 for Windows*. Once you have opened the Progress Explorer, follow these basic steps to configure an AppServer instance:

1 ♦ Make sure that the AdminServer process is running on each of the following machines:

- Where the AppServer you want to configure is installed
- Where all NameServers you want to configure are installed

On NT, when you complete AppServer installation, the AdminServer automatically starts and is configured to auto-start whenever you reboot your system. You can also start the AdminServer from the Windows NT Services applet available from the Control Panel. The AdminServer is running if you see both a `jview.exe` and `admsrvc.exe` process listed in the Processes tab of the Task Manager.

On UNIX, use the PROADSV command to start the AdminServer. To check whether the AdminServer is running, run the `ps` command to show the full command line for each process on the system and locate any `jre` commands in the list. The AdminServer process is running if you see a `jre` command with the arguments that correspond to those specified for `jvmstart` in the Progress `proadsv` shell script located in *`ProgressInstallDir`*`/bin`.

For more information on starting the AdminServer, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

2 ♦ In the Progress Explorer, connect to the running AdminServer processes that you verified in Step 1. After you connect to a machine running an AdminServer process, its host name or IP address appears as a folder in the tree view.

3 ♦ Expand each machine folder to see a list of folders showing all Progress server products installed on the machine.

4 ♦ Select the NameServer product folders where you want to configure NameServers and configure one or more such instances to support all AppServer instances you want to configure. Note that for every AppServer instance that you plan to run on a separate machine from its controlling NameServer, you must configure a remote NameServer instance on the same machine as the AppServer. This remote NameServer instance must reference the host and port of the controlling NameServer for the AppServer. For more information, see the chapter on configuring Progress Unified Broker products in the *Progress Installation and Configuration Guide Version 9 for Windows*.

5 ♦ Select the AppServer product folder on the machine where you want the AppServer instance to reside. Configure the instance to register with the controlling NameServer (with or without load balancing). Specify any Application Service names required for clients to access the AppServer and any other required configuration information. For more information, see "Configuring an AppServer Using the Progress Explorer" section.

The following sections describe the basic steps for configuring AppServers. For more information on using the Progress Explorer to configure an AppServer instance, see the Progress Explorer online help.

3.2.2 Configuring an AppServer Using the Progress Explorer

Follow these steps to define and configure an AppServer instance:

1 ♦ Make sure that the Progress AdminServer is running on the host where you want to configure the AppServer (see the "Using the Progress Explorer to Configure an AppServer Instance" section).

2 ♦ Open the Progress Explorer.

3 ♦ Connect to the AdminServer on your AppServer host (see the *Progress Installation and Configuration Guide Version 9 for Windows*).

4 ♦ Do one of the following:

- To define a new AppServer, select the AppServer folder in the Progress Explorer's tree view and open the New dialog box. Enter a unique name for the AppServer. Choose OK. The Preferences Dialog box appears. Set any global preferences that you require (see the online help), and choose OK. The AppServer instance property editor appears.

- To modify an existing AppServer configuration, expand the AppServer folder in the tree view, select the AppServer instance you want to modify, and open the AppServer property editor.

The AppServer instance property editor shows a tree view of property categories on the left and the properties for the selected category on the right.

5 ♦ Select a property category and set the properties as required. You can accept the default values, if they are appropriate for your application. You probably want to specify the properties under each category, as follows:

- **Broker**—This category specifies properties of the Application Broker. Expanding this category shows the following property subcategories:

  General—You must specify an Operating mode, and you probably want to specify a non-default value for the TCP/IP port number where the Application Broker listens for requests.

  You must also select a controlling NameServer from the list of NameServer instances that you have already configured.

  If you want the AppServer to start whenever you start the AdminServer, select the Auto start check box, and if you want to use a different working directory from the one specified during AppServer product installation, you can also change it here.

  Owner Information—You can optionally provide Username and Password information for the user who owns the AppServer instance.

  AppService Name List—You can either enter any names for the Application Services supported by this AppServer, or you can select the Supports default service check box if you want this AppServer to support the default service for all client connections that do not specify an Application Service name. If you choose to use Application Service names, the default Application Service name is the name of the AppServer instance.

  Logging Setting—You can specify a different pathname from the default for the broker log file, specify the level of logging detail, and whether the logging for a session appends to or overwrites the previous broker log file.

Advanced Features—You can specify the maximum number of client connections (Maximum client instances) that the AppServer can support at one time, the AppServer weight factor (Priority weight) for load balancing, the time between retries to register the AppServer with the controlling NameServer, the timeout period for starting the AppServer, the timeout period for an AppServer to accept a client request, and the timeout period for the Application Broker to trim its quota of Application Server processes between the maximum and minimum setting (see the Server category). For more information on these options, see the Progress Explorer online help.

- **Server**—This category specifies properties of the Application Server processes. Expanding this category show the following property subcategories:

  General—You can specify a pathname of the Application Server process executable (Server executable file). You generally only need to specify a different value than the default if you build a new Application Server process executable using the PROBUILD utility. For more information, see the "Customizing the Application Server Process Executable" section.

  Specify the Progress startup parameters for the Application Server process (Server startup parameters). These are the standard Progress client startup parameters, and can include any parameters that you require for each AppServer session, including (but not limited to) all of the standard database, code-page, and process management parameters. For more information, see the *Progress Startup Command and Parameter Reference*.

  The AppServer can run with a different code page than the client application. For more information, see the "Managing Code Pages" section.

  For PROPATH, specify the semicolon-separated list of directories where the AppServer can locate 4GL procedures to execute. This setting overrides any PROPATH environment variable settings on the AppServer host when it starts up. Make sure that all of the 4GL procedures (r-code or source) that you want the AppServer to execute are located in one of these PROPATH directories. Otherwise, the procedure must be executed using its fully-qualified pathname.

  Specify the minimum and maximum TCP/IP port numbers that the Application Broker can assign to Application Server processes that it starts up. (Check with your system administrator for appropriate ranges.)

  Logging Setting—You can specify a different pathname from the default for the server log file, specify the level of logging detail, and specify whether the logging for a session appends to or overwrites the previous broker log file.

Pool Range—These settings determine the number of Application Server processes that the Application Broker can startup and maintain for the AppServer. For more information on setting these values, see the "Specifying the Server Pool Parameters" section.

Advanced Features—To allow the 4GL debugger to run in the AppServer session, select the 4GL debugger enabled check box. For more information, see Chapter 7, "Debugging." Specify the names of any AppServer configuration procedures that you want the AppServer to execute, and any parameters for the Startup procedure. For more information on AppServer configuration procedures, see Chapter 4, "Programming the AppServer."

- **Environment Variables—**If you want to specify environment variables for AppServer execution, select this category. It allows you to enter name-value pairs for environment variable settings. Any values you set here override prior values set for the same environment variables in the operating system. For more information, see the "Environment Variable Settings" section. (**NOTE:** Do not set the PROPATH variable using this category. Use the Server General category, instead.)

6 ♦ Choose the OK button to save the configuration.

3.2.3 Configuring an AppServer Using the Properties File

Progress stores the configurations for both the NameServer and AppServer products using a properties file (`ubroker.properties`). The UNIX and NT `ubroker.properties` files are the same except for platform-specific differences (for example, differences in directory path separators, and the differences between environment variable references in UNIX and registry references in NT).

The properties file stores all the configuration definitions for all instances of the NameServer, and all instances of any AppServer, WebSpeed Server, and DataServer product that run on the same machine. Each configuration definition contains environment variables, registry entries (if Windows NT), and property settings for each product instance.

There is one copy of this file for each Progress installation. Thus, if you install the NameServer on a separate machine from the AppServer product that it manages, the NameServer and AppServer product each have their own copy of the `ubroker.properties` file.

The AdminServer reads and updates this file according to your instructions using the Progress Explorer and management utilities. The `ubroker.properties` file is installed in the `properties` subdirectory of the Progress installation directory (for example, `$DLC/properties/ubroker.properties` on UNIX, or `%DLC%\properties\ubroker.properties` on Windows NT). In order for the AdminServer to access the properties file, the file must reside in this directory.

Guidelines for Editing the Properties File

When editing the `ubroker.properties` file without the Progress Explorer, note that:

- You should not directly change the values in the `ubroker.properties` file unless you have a complete understanding of how the changes affect AppServer components. When possible, always use the Progress Explorer to make all changes to this file.
- Always make a copy of this file, edit the copy, and verify the result before replacing the original with your edited copy.
- For complete definitions of all the properties and detailed information on how to set them, see the comments included in the properties file.

The file consists of a hierarchical structure of configuration entities, where parent entities provide configuration information that you can override or extend in each child entity. Each configuration entity has a name that begins the entity definition, and the definition contains configuration settings for one or more products or product instances. The AppServer configurations in `ubroker.properties` can include:

Configuration Entity Name	Configuration Entity Function
`[UBroker]`	Defines default property settings for all AppServers, DataServers, and WebSpeed Transaction Servers.
`[UBroker.AS]`	Defines default property settings for all instances of an AppServer.
`[UBroker.AS.`*`product-instance-name`*`]`	Defines property settings for this instance of an AppServer. The `ubroker.properties` file can contain several of these entities each with a unique *`product-instance-name`*. For an AppServer, *`product-instance-name`* is the name you specify for the AppServer instance.

Thus, parent entities provide default values for all of their child entities. For example, the parent [UBroker] contains a set of definitions that can be inherited by its child AppServer product [UBroker.AS] and any other product entities, and then again by its child [UBroker.AS.*product-instance-name*] and any other product instance entities. However, at any child level, a redefinition of any value supersedes the default value of its parent. All children from the redefinition level down inherit this new value.

Editing and Validating the Properties File

To edit the properties file directly, use a text editor such as vi or Notepad. Once you edit the properties file, use the following utilities to validate the AppServer configuration information in the file:

- **NSCONFIG utility**—To validate NameServer configurations. For more information on this utility, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.
- **ASCONFIG utility**—To validate AppServer configurations. This section describes how to use this utility.

If the file contains any other Progress Application Server configurations, run the configuration validation utilities for those Unified Broker products to ensure that these configurations are still valid. For more information, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

NOTE: If you always use the Progress Explorer, you never have to use these utilities.

The ASCONFIG utility displays the property settings associated with an AppServer configuration, and checks that the syntax and values are valid. You must run the ASCONFIG utility locally on the machine on which the AppServer is running. The utility does not run across the network.

This is the syntax used to invoke the ASCONFIG utility:

SYNTAX

```
asconfig
  [
     [
        [ -name AppServer-name ]
        [ -propfile path-to-properties-file ]
        [ -validate ]
     ]
   | -help
  ]
```

`-name` *AppServer-name*

Specifies which existing AppServer configuration to examine. The name must match the name of an existing AppServer configuration in the specified properties file. If you do not specify an AppServer, the ASCONFIG utility analyzes all AppServer configurations defined in the properties file specified by the `-propfile` parameter.

`-propfile` *path-to-properties-file*

Specifies a filename or pathname to a file that contains the property settings to be validated, for example `test.properties`. If a filename or pathname is not specified, it defaults to the installation version of the `ubroker.properties` file, such as:

- `%DLC%\properties\ubroker.properties` on Windows NT
- `$DLC/properties/ubroker.properties` on UNIX

`-validate`

Checks the syntax and values of property settings defined in the specified properties file.

`-help`

Displays command-line help.

For example, the following command validates the syntax and views the configurations of all AppServer instances defined within the `test.properties` file located in the current working directory:

```
asconfig -propfile test.properties -validate
```

For more information on using this utility, see the "Summary of Management Tasks" section in this chapter.

3.3 Setting Up the Environment for AppServer Execution

You might need to perform the following tasks before you start an AppServer instance:

- Set any standard environment variables (for example, DLC) on the AppServer and NameServer machines.
- Decide where to put your AppServer and NameServer log files.
- Copy r-code files to support any Progress SmartDataObjects you want to run on the AppServer.
- Customize your Application Server process executable using PROBUILD.

3.3.1 Environment Variable Settings

Environment variables required by an AppServer are set during either AppServer installation or startup. You can modify these initial settings in the registry on NT or in the shell scripts installed with Progress on UNIX. You can also override most of these settings for an AppServer or NameServer using the Progress Explorer. For more information, see the "Configuring AppServer Components" section.

For more information on setting the standard Progress environment variables, see the chapters on setting up the environment and configuring Progress Unified Broker products in the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

Working Directory Settings

The working directory settings for AppServer and NameServer each specify the directory in which the specified product starts up and, by default, where its log files and any other files output by AppServer applications are written. As with other settings, you can change or override the default settings for the working directory. You can also individually specify the locations of AppServer and NameServer log files. For more information, see the "Configuring AppServer Components" section.

3.3.2 AppServer and NameServer Log Files

The AppServer and NameServer, together, generate three log files that you can use to monitor AppServer activity. For more information on working with AppServer and NameServer log files, see the chapter on configuring Progress Unified Broker products in the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

3.3.3 Progress Remote SmartDataObject Support

To allow a remote SmartDataObject to run on the AppServer, copy the Progress-compiled ADM2 (Application Development Model, Version 2) r-code files to a directory in the AppServer PROPATH. You can find these files in the *ProgressInstallDirectory*\gui\adm2 directory, wherever you have the Progress AppBuilder installed. For example, you can copy these files to your AppServer *ProgressInstallDirectory*\tty\adm2 directory. For more information on SmartDataObjects, see the *Progress AppBuilder Developer's Guide*.

3.3.4 Customizing the Application Server Process Executable

The Progress AppServer includes a standard Application Server process executable that, in most cases, requires no customizing. However, you might have special needs that require you to add more software modules (*configurable elements*) to the executable. For example, the Application Server process might be using a Progress DataServer to connect to a non-Progress database. In this case, you must include the appropriate DataServer when building the executable.

The Application Server process executable is essentially a version of the Progress 4GL client. Like the 4GL client, you can rebuild the Application Server process executable using the PROBUILD utility.

Thus, to build a new Application Server process executable and make it available to your AppServer installation:

1 ♦ Run the Progress PROBUILD utility.

2 ♦ Select PROGRESS APPSERVER from the Product List.

3 ♦ Specify names for the executable and its link script.

4 ♦ Select the configurable elements that you want to include in the new executable.

5 ♦ Create the link script and exit the PROBUILD utility.

6 ♦ Build the executable using the link script according to your development platform (NT or UNIX) requirements.

7 ♦ Specify the pathname for your new Application Server process executable using the Progress Explorer GUI or by setting the `srvrExecFile` property in the AppServer properties file (`ubroker.properties`).

For more information on building an Application Server process executable, see the sections on PROBUILD in the *Progress Client Deployment Guide*. For more information on specifying the pathname for the Application Server process executable, see the "Configuring AppServer Components" section.

3.4 Starting and Managing an AppServer Instance

Progress provides two mechanisms to start and manage AppServers—the Progress Explorer tool and the command-line management utilities. In general, you can start AppServer instances and the NameServer that controls them in any order. However, to allow clients to connect to any AppServer, you must first start the controlling NameServer for that AppServer.

You can also shut down a NameServer and the AppServer instances that it controls in any order. However, when you shut down the NameServer, any client applications that are not already connected cannot connect to the AppServer instances that this NameServer controls. Connected clients can still continue to operate with any running AppServers. If you restart the NameServer, any running AppServers that it controls automatically register with it and again become available for client connections.

Before you begin to develop or deploy applications on Windows NT or UNIX, note the requirements for each platform.

3.4.1 Requirements for Starting an AppServer

Make sure that the AdminServer process is running on each of the following machines:

- Where the AppServer instance you want to start resides
- Where all required NameServers reside
- Where any Progress databases or other DataServers reside that your AppServer instance needs to access

For more information, see the "Using the Progress Explorer to Configure an AppServer Instance" section.

3.4.2 Starting and Managing an AppServer with the Progress Explorer

You can use the Progress Explorer on a Windows platform to start:

- A local AppServer instance on NT only.
- A remote AppServer instance running on NT or UNIX.

To access the Progress Explorer tool, you can open it from the Progress program group or in the Microsoft Management Console (MMC) on your Windows system. For more information, see the *Progress Installation and Configuration Guide Version 9 for Windows*. Once you have opened the Progress Explorer in the MMC, follow these basic steps to start an AppServer instance:

1 ♦ In the Progress Explorer, connect to each of the running AdminServer processes required by your AppServer configuration. The name or IP address of the machine running each of these processes appears as a folder in the tree view.

2 ♦ Expand each machine folder to see a list of folders showing the name of each NameServer and AppServer product installed on the machine.

3 ♦ Expand the NameServer folder under each machine to see a list of configured NameServers.

4 ♦ Start each NameServer that you need to support your AppServer instance and that is not already auto started. For more information, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

5 ♦ Start each Progress Database or DataServer that your application requires and that is not already auto started. For more information on starting databases and DataServers, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

6 ♦ Under the machine folder where it resides, expand the AppServer folder for the instance you want to start.

7 ♦ Select the AppServer instance you want to start.

8 ♦ Do one of the following:

- Right-click the instance to open a popup menu and choose Start.
- Choose the Start button.
- Choose Action→ Start.

Using the Progress Explorer you can also invoke the following management functions for the running AppServer instance:

- Stop the AppServer.

  NOTE: You cannot stop the AppServer if it is actively handling any client requests.

- Check and manage the operational status of the AppServer. Using the status dialog box, you can also reduce (Trim servers button) and increase (Add servers button) the number of running Application Server processes for this AppServer. For more information on the effects of these functions, see the "Specifying the Server Pool Parameters" section. For more information on AppServer status indicators, see the "Checking AppServer Status" section.

- View the log files for the AppServer. To view a log file, expand the AppServer instance folder in the tree view and select the log file you want to view. The contents of the log file appears in the right pane.

- Delete the AppServer instance.

  NOTE: Before you can delete an AppServer instance, you must stop the AppServer.

For more information on using the Progress Explorer to start and manage an AppServer instance, see the Progress Explorer online help.

3.4.3 Starting and Managing an AppServer with the Management Utilities

You can use the command-line management utilities on Windows or UNIX platforms to start local AppServer instances and remote AppServer instances running on any platform.

To start an AppServer instance:

1 ♦ Using the NSMAN utility, start each NameServer that you need to support your AppServer instance. For more information, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

2 ♦ Start each Progress Database or DataServer that your application requires. For more information on starting databases and DataServers, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

3 ♦ Using the ASBMAN utility, start the AppServer instance.

Using the ASBMAN Utility

The ASBMAN utility runs on both Windows and UNIX platforms. It allows you to invoke the following management functions on a local or remote AppServer instance:

- Start an AppServer.
- Check and manage the operational status of the AppServer. Management options allow you to reduce (trim) and increase (add) the number of running Application Server processes for this AppServer. For more information on the effects of these options, see the "Specifying the Server Pool Parameters" section. For more information on AppServer status indicators, see the "Checking AppServer Status" section.
- Stop the AppServer.

Unlike the Progress Explorer, the ASBMAN utility has no mechanism for viewing log files or deleting configured AppServer instances. If you want to view the AppServer log file or delete the AppServer instance, you must do it manually using operating system commands. To delete the AppServer, you must remove the entry for this AppServer instance in the AppServer properties file or use the Progress Explorer. For more information on managing log files, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*. For more information on accessing the AppServer properties file, see the "Configuring an AppServer Using the Properties File" section.

NOTE: Before you can delete an AppServer instance, you must stop the AppServer.

This is the syntax to invoke the ASBMAN utility:

SYNTAX

```
asbman
  {
      { -name AppServer-name
        {
             -kill
           | -start
           | -stop
           | -query
           | -addservers number-to-start
           | -trimservers number-to-trim
        }
        [
             -host host-name -user user-name
           | -user user-name
   ]
        [ -port port-number]
      }
    | -help
  }
```

`-name` *AppServer-name*

This parameter is required. It specifies the name of an AppServer.

`-kill`

Stops and removes the AppServer from memory, no matter what it is doing.

`-start`

Starts an AppServer.

`-stop`

Tells the NameServer to stop itself.

NOTE: The AppServer stops only after completing any active client requests.

`-query`

Queries an AppServer for its status.

`-addservers` *`number-to-start`*

Specifies the number of additional servers to start.

`-trimservers` *`number-to-trim`*

Specifies the number of additional servers to trim.

`-host` *`host-name`*

Specifies the name of the machine where the AdminServer is running. If a host name is not specified, it defaults to the local host name.

`-user` *`user-name`*

Specifies a user name and prompts for a password. A user name and password are required only when you use the `-host` parameter and specify a remote host name. If you specify a remote host name with the `-host` parameter, but do not specify a user name with the `-user` parameter, you receive a prompt for a user name and password.

Windows NT supports three different formats for *`user-name`*:

- A user name as a simple text string, such as "`mary`," implies a local user whose user account is defined on the local NT server machine, which is the same machine that runs the AdminServer.
- A user name as an explicit local user name, in which the user account is defined on the same machine that runs the AdminServer except the user name explicitly references the local machine domain, for example "`.\mary`".
- A user name as a user account on a specific NT domain. The general format is *`Domain\User`*, in which the *`User`* is a valid user account defined within the domain and the *`Domain`* is any valid NT Server, including the one where the AdminServer is running.

`-port` *`port-number`*

Specifies the port number of the machine on which the AdminServer is running. If a port number is not specified, it defaults to 20931.

`-help`

Displays command-line help.

Examples

The following commands start a local AppServer instance (AS1) after starting the local controlling NameServer (NS1):

```
nsman -name NS1 -start
asbman -name AS1 -start
```

The following commands start a remote AppServer instance (AS1) after starting a remote controlling NameServer (NS1):

```
nsman -name NS1 -host nsserve -port 20950 -user daniel -start
asbman -name AS1 -host asserve -port 20950 -user daniel -start
```

Note that the AppServer and controlling NameServer are on different hosts and happen to use the same TCP/IP port number to access the AdminServer on each host. If you specify a host, the Explorer prompts always prompts for a user name (if necessary) and password. In this example, the commands specify the user name and prompt only for the password.

The following commands stop a local AppServer instance (AS1) and its controlling NameServer instance (NS1):

```
asbman -name AS1 -stop
nsman -name NS1 -stop
```

3.4.4 Checking AppServer Status

You can check the status of a running AppServer using either the Progress Explorer or the `-query` option on the ASBMAN command. The returned status includes information on each Application Server process running on the AppServer. The information on each process includes one of the status indications listed in Table 3–1.

Table 3–1: Application Server Process Status Indications

Status	Description
AVAILABLE	The process is available to execute application logic.
BUSY	The process is actively executing application logic for an AppServer client. For a state-aware or state-reset AppServer, this status persists until the client disconnects from the AppServer.
LIMBO	The process is in a transitional state. If this status persists, it indicates an error condition.
LOCKED	On a stateless AppServer, the process is in a bound connection with an AppServer client. This status persists until the connection becomes unbound.
STARTING	The AppServer has started the process, but it has not yet completed initialization.

3.5 Specifying the Server Pool Parameters

These parameters specify how many Application Server processes an AppServer starts and maintains to service clients. The number of running processes changes depending on these settings and how you are managing AppServer execution. While there is no exact set of rules for determining the correct settings for the server pool parameters, you can start with some basic practical considerations. These parameters include:

- **Initial number of servers to start**—The number of processes started when the AppServer first starts up.
- **Minimum servers**—The minimum number of processes that the AppServer keeps running to meet client demand.
- **Maximum servers—**The maximum number of processes that the AppServer keeps running to meet client demand.

3.5.1 Tuning for Operating Modes

In general, for a state-reset or state-aware AppServer, you probably want to set the Initial Server Instances to Start parameter to the average number of AppServer clients you expect so that each connecting client has efficient access to an Application Server process. Thus, you are looking for the optimum settings for the number of concurrent connections. However, if your client load varies widely, you might set this value higher to offset the impact of client surges.

For a stateless AppServer, you need to tune these parameters to handle changing client load based on how many clients that you can expect a single Application Server process to handle in a given time period. Thus, you are looking for the optimum settings for the number of concurrent requests.

One way to assess this is capacity for all operating modes is to set a high value for the Maximum Server Instances parameter and a low value for the Minimum Server Instances parameter. Then, run live for a period of time and see how many Application Server processes the AppServer tends to start. You might want to set the Initial Server Instances to Start parameter to that value.

3.5.2 Managing Application Server Processes

In general, you can manage running Application Server processes based on the Server Pool Parameter settings as follows:

- You can trim any number of running Application Server processes below your Minimum Server Instances parameter value to zero.
- You cannot add any more Application Server processes than your Maximum Server Instances parameter setting. However, once you have found the maximum number of Application Server processes that you can run productively for the AppServer, you might even want to set that value lower if the setting negatively impacts other tasks on the system.

3.6 Managing Code Pages

The AppServer supports clients running with different code pages from the AppServer. When a client with a different code page invokes a remote procedure or user-defined function, the AppServer that receives the request converts character-string data between the client and AppServer code page as appropriate. The actual conversions are performed by the Application Server process that responds to the request.

3.6.1 Code-page Settings

You can set code pages for 4GL clients and AppServers using several code page startup parameters. Each parameter allows you to specify any one of a wide variety of supported code pages for a specified data domain in the Progress environment (for example, memory code page or input/output stream code page). For the AppServer, you specify code page startup parameters in the Progress Explorer as part of the `srvrStartupParam` property setting in the `ubroker.properties` file. For more information on these parameters, see the chapter on code pages in the *Progress Internationalization Guide*. Whatever the settings, the AppServer performs the necessary conversions to and from the client application.

NOTE: Note that the Terminal Code Page (-cpterm) startup parameter has no meaning for the AppServer.

Both Java and Active/X open clients send requests (input) to the AppServer and receive responses (output) from the AppServer using Unicode UTF-8. The AppServer automatically converts between these open client Unicode UTF-8 requests and the particular code page that the AppServer is using.

NOTE: If your AppServer application provides database services to a multi-lingual open client application, using UTF-8 for both the AppServer and the database provides the most effective data interchange across application components.

CAUTION: Unicode supports text with embedded nulls. Progress does not support this feature, no matter what character set it uses. Make sure that any open client applications filter character strings input to the AppServer for embedded nulls. Unfiltered input can result in unpredictable errors returned by the AppServer.

3.6.2 Run-time Conversions

The AppServer converts the following types of character data passed between a client and the Application Server process:

- Columns of input and output temporary tables
- Connection parameters
- Error messages
- Input and output parameters
- Return values (RETURN *string*) from 4GL procedures
- Return values from 4GL user-defined functions
- The names of executed 4GL remote procedures or functions

3.7 Summary of Management Tasks

This section summarize the basic management tasks that are available to maintain existing AppServer configurations. They are described with reference to the command-line utilities, but you can also accomplish most of these tasks, except running the AdminServer, using the Progress Explorer.

Table 3–2 describes the utilities (in addition to the Progress Explorer tool) that Progress provides to help configure and manage a Progress AppServer installation, listed generally in order by task.

Table 3–2: AppServer Utilities

Utility	Execution Location	Description
PROADSV	Local to the AppServer installation	Starts, stops, and queries status for the AdminServer on UNIX. On NT, you start the AdminServer as an NT service using the Services applet in the Control Panel.
NSCONFIG	Local to the AppServer installation files	Validates the syntax of NameServer configurations in the `ubroker.properties` file. (Never needed if you always use the Progress Explorer GUI.)
ASCONFIG	Local to the AppServer installation files	Validates the syntax of AppServer configurations in the `ubroker.properties` file. (Never needed if you always use the Progress Explorer GUI.)
NSMAN	Local or remote to the NameServer machine	Starts, stops, and queries status for a NameServer.
ASBMAN	Local or remote to the AppServer installation	Starts, stops, adds Application Server processes, trims Application Server processes, and queries status for an AppServer instance and its Application Broker.

For more information on the NameServer and NSMAN utility, and the AdminServer and PROADSV utility, see the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

Table 3–3 lists sample command-line utility syntax used to:

- Start and stop the AdminServer on the Windows NT platform.
- Start, query, and stop the AdminServer on the UNIX platform.
- Verify the configuration of, start, query, and stop a NameServer instance.
- Verify the configuration of, start, query, add Application Server processes to, trim Application Server processes from, and stop an AppServer instance.

Table 3–3: Management Tasks *(1 of 3)*

To do this task ...	Use these steps or commands ...
Start or stop the AdminServer on Windows NT	From the Windows NT desktop, choose Settings→ Control Panel→ Services. Choose the AdminService for Progress 9.1*x*, and click either Startup or Stop.
Start the AdminServer on UNIX	`proadsv -start`
Start the AdminServer on UNIX using a specified port	`proadsv -port` *`port-number`* `-start`
Query the AdminServer on UNIX	`proadsv -host` *`host-name`* `-user` *`user-name`* `-query`
Stop the AdminServer on UNIX	`proadsv -stop -port` *`port-number`*
View a NameServer configuration in the `ubroker.properties` file	`nsconfig -name` *`Name-Server`*
Validate the syntax and view the configurations of all NameServer instances defined within a specified property file	`nsconfig -propfile` *`property-file-path`* `-validate`
Start a local NameServer	`nsman -name` *`Name-Server`* `-start`
Start a remote NameServer[1, 2]	`nsman -name` *`Name-Server`* `-host` *`host-name`* `-port` *`port-number`* `-user` *`user-name`* `-start`
Query a local NameServer	`nsman -name` *`Name-Server`* `-query`

Table 3–3: Management Tasks *(2 of 3)*

To do this task ...	Use these steps or commands ...
Query a remote NameServer[1]	`nsman -name` *Name-Server* `-host` *host-name* `-port` *port-number* `-user` *user-name* `-query`
Stop a local NameServer	`nsman -name` *Name-Server* `-stop`
Stop a remote NameServer[1]	`nsman -name` *Name-Server* `-host` *host-name* `-port` *port-number* `-user` *user-name* `-stop`
View an AppServer configuration in the `ubroker.properties` file	`asconfig -name` *AppServer-name*
Validate the syntax and view the configurations of all AppServer instances defined within a specified property file	`asconfig -propfile` *property-file-path* `-validate`
Start a local AppServer	`asbman -name` *AppServer-name* `-start`
Start a remote AppServer[1]	`asbman -name` *AppServer-name* `-host` *host-name* `-port` *port-number* `-user user-name -start`
Query a local AppServer	`asbman -name` *AppServer-name* `-query`
Query a remote AppServer[1]	`asbman -name` *AppServer-name* `-host` *host-name* `-user user-name -query`
Add Application Server processes to a local AppServer	`asbman -name` *AppServer-name* `-addservers #`
Add Application Server processes to a remote AppServer[1]	`asbman -name AppServer-name -host` *host-name* `-user -addservers #`
Trim Application Server processes from a local AppServer	`asbman -name` *AppServer-name* `-trimservers #`
Trim Application Server processes from a remote AppServer[1]	`asbman -name` *AppServer-name* `-host` *host-name* `-user` *user-name* `-trimservers #`
Stop a local AppServer	`asbman -name` *AppServer-name* `-stop`

Table 3–3: Management Tasks *(3 of 3)*

To do this task ...	Use these steps or commands ...
Stop a remote AppServer[1]	`asbman -name` *`AppServer-name`* `-host` *`host-name`* `-port` *`port-number`* `-user` *`user-name`* `-stop`

[1] Prompts for a password.

[2] This is a NameServer defined on a remote machine, not a remote NameServer instance defined in the `ubroker.properties` file of the local machine.

4

Programming the AppServer

This chapter describes how to program procedures that execute on the AppServer. Specifically, the sections in this chapter describe:

- 4GL for programming AppServer procedures
- Accessing the connection ID
- Using AppServer configuration procedures
- Setting AppServer entry points
- Managing stateless connection context
- Understanding remote procedure handles
- Handling conditions and returning values
- Managing transactions
- Programming for open client applications

This chapter addresses only programming issues specific to AppServer procedures. It assumes that you are generally familiar with writing and executing Progress 4GL procedures. For more information on using the Progress 4GL, see the *Progress Programming Handbook*.

For information on programming issues specific to AppServer client applications, see Chapter 5, "Programming the Client Application." For information common to programming both 4GL client and AppServer procedures, see Chapter 6, "Design and Implementation Considerations."

4.1 4GL for Programming AppServer Procedures

Table 4–1 lists the 4GL elements that are either valid only in an AppServer session or have special application in AppServer programming. The remaining sections in this chapter explain how to use these elements.

Table 4–1: 4GL for Programming AppServer Procedures *(1 of 3)*

4GL Element	Description
DEFAULT-COMMIT	A LOGICAL attribute on a transaction object handle that specifies how the transaction object is to complete the transaction if an automatic transaction terminates with no prior SET-COMMIT( ) or SET-ROLLBACK( ) being invoked.
DELETE OBJECT *handle*	A statement that you can use to delete certain session objects, including local and remote persistent procedures.
DELETE PROCEDURE *procedure-handle*	A statement that you can use to delete both local and remote procedure objects (persistent procedures).
EXPORT ([*list*])	A method on the SESSION system handle that creates and modifies the list of remote procedures provided by the AppServer.
FUNCTION ... IN SUPER	A statement that specifies the prototype definition for a user-defined function defined within a super procedure.
IS-OPEN	A LOGICAL attribute on a transaction object handle that returns TRUE if a database transaction is active. This is identical to the 4GL TRANSACTION function.
PERSISTENT	A LOGICAL attribute on procedure handles and the THIS-PROCEDURE system handle that is TRUE if the specified procedure is running persistently as a procedure object.
PROCEDURE ... IN SUPER	A statement that begins the prototype definition for an internal procedure defined within a super procedure.

Table 4–1: 4GL for Programming AppServer Procedures *(2 of 3)*

4GL Element	Description
REMOTE	• A LOGICAL attribute on the SESSION system handle that is TRUE if the current session is running in the context of a Progress AppServer. • A LOGICAL attribute on procedure handles and the THIS-PROCEDURE system handle that is TRUE if the specified procedure is running at the top level in the context of a Progress AppServer (as the result of a remote procedure call by a client application). Always FALSE if PROXY is TRUE.
SERVER-CONNECTION-BOUND	A LOGICAL attribute on the SESSION system handle that is TRUE if the AppServer session is bound to a particular client application. Valid only if the REMOTE attribute is TRUE.
SERVER-CONNECTION-BOUND-REQUEST	A LOGICAL attribute on the SESSION system handle that, when set to TRUE, requests that the AppServer session be bound to the current client connection identified by the SERVER-CONNECTION-ID attribute. When set to FALSE, requests that the AppServer session be unbound from the currently bound client connection pending deletion of all remote persistent procedures running in the session. Valid only if the REMOTE attribute is TRUE.
SERVER-CONNECTION-CONTEXT	A CHARACTER attribute on the SESSION system handle that contains an application-determined value that you can set. Progress passes this value to each Application Server process that executes a request on behalf of the client connection identified by the SERVER-CONNECTION-ID attribute. Valid only if the REMOTE attribute is TRUE.
SERVER-CONNECTION-ID	A CHARACTER attribute on the SESSION system handle that returns the run-time connection ID of the current client connection assigned to this AppServer session. Valid only if the REMOTE attribute is TRUE.

Table 4–1: 4GL for Programming AppServer Procedures *(3 of 3)*

4GL Element	Description
SERVER-OPERATING-MODE	A CHARACTER attribute on the SESSION system handle that returns the operating mode specified by the `operatingMode` property set for this AppServer in the `ubroker.properties` file. Valid only if the REMOTE attribute is TRUE.
SET-COMMIT ( )	A method on a transaction object handle that tells the transaction object to commit any automatic transaction when the current request completes and returns execution to the client.
SET-ROLLBACK ( )	A method on a transaction object handle that tells the transaction object to roll back any automatic transaction when the current request completes and returns execution to the client.
TRANSACTION	A HANDLE attribute on procedure handles and the THIS-PROCEDURE system handle that provides a handle to the current transaction object.
TRANS-INIT-PROCEDURE	A HANDLE attribute on a transaction object handle that, if an automatic transaction is active, returns the procedure handle to the transaction initiating procedure that started the transaction.

4.2 Accessing the Connection ID

As described in Chapter 2, Progress creates a unique ID for each connection to an AppServer. In any AppServer session that executes a request for a particular connection, you can access the connection ID as the value of the SERVER-CONNECTION-ID attribute on the SESSION handle. This is a read-only attribute of type CHARACTER.

For information on how the AppServer generates the value for a connection ID, see the information on the AppServer connection-based model in Chapter 2, “Overview of the Progress AppServer.”

4.3 Using AppServer Configuration Procedures

AppServer configuration procedures are Progress procedures that you can specify when you configure an AppServer, and that run at specific times during an Application Server process's lifetime. There are three types of procedures:

- Startup and Shutdown procedures
- Connect and Disconnect procedures
- Activate and Deactivate procedures

Each of these procedures is run at specific times during the lifetime of an Application Server process with the idea of simplifying the process of deploying an application with the AppServer.

4.3.1 Startup and Shutdown Procedures

Startup and Shutdown procedures encapsulate logic that executes during the startup and shutdown of an AppServer. In effect, the logic in these procedures executes in each Application Server process as it starts up and shuts down. Functionality that might go in a Startup or Shutdown procedure includes connecting or disconnecting databases, loading or storing the contents of temporary tables, and instantiating or deleting certain local persistent procedures.

Startup Procedure

A Startup procedure executes as an Application Server process starts up. You can specify the name of an AppServer Startup procedure using the Progress Explorer GUI or by setting the `srvrStartupProc` property in the AppServer properties file (`ubroker.properties`).

The Startup procedure takes a character string as an input parameter, for example:

```
DEFINE INPUT PARAMETER startup-data AS CHARACTER NO-UNDO.
```

You can set the value for this parameter in the properties file using the `srvrStartupProcParam` property, and you can set it to any arbitrary value. If you do not specify a value for this property, the parameter is set to the unknown value (?) when the Application Server process executes the Startup procedure.

The Startup procedure is automatically executed as a persistent procedure within each Application Server process when the Application Server process first starts. The persistent procedure is automatically deleted just before the process terminates. If you do not need the context established by the Startup procedure, you can delete the persistent procedure within an Application Server process context by executing the DELETE PROCEDURE (or DELETE OBJECT) statement using the appropriate procedure handle.

If the Startup procedure completes with no error, the AppServer starts up successfully. If the Startup procedure completes with an error return value (RETURN ERROR statement), AppServer startup fails and the Application Server process is terminated.

Shutdown Procedure

A Shutdown procedure executes just before the Application Server process is shut down. A Shutdown procedure takes no parameters. You can specify the name of an AppServer Shutdown procedure using the Progress Explorer GUI or by setting the `srvrShutdownProc` property in the AppServer properties file (`ubroker.properties`).

Unlike the Startup procedure, the Shutdown procedure is run as a non-persistent procedure, and any errors propagated by the Shutdown procedure are simply ignored. The Application Server process terminates immediately after the Shutdown procedure executes.

Usage Requirements

You can specify Startup and Shutdown procedures for a state-aware or stateless AppServer. You cannot specify Startup and Shutdown procedures for a state-reset AppServer.

Note that both the Startup and Shutdown procedures are optional. There is no requirement to specify either or both of them. Additionally, the content of these procedures is completely application specific, and the AppServer architecture places no constraints on it. Any 4GL logic that is valid within an Application Server process is valid for these procedures.

NOTE: In Progress Version 8, you used the Connect procedure to program all startup activities on an Application Server process. With the introduction of reusable and stateless servers much of the logic that you might have put into the Connect procedure for the Version 8 AppServer is now more appropriate for the Startup procedure. Similarly, much of the logic in a Version 8 Disconnect procedure might be more appropriate for the Shutdown procedure.

4.3.2 Connect and Disconnect Procedures

Connect and Disconnect procedures encapsulate logic that executes whenever a client application establishes and terminates a connection with an AppServer. Functionality that might go in a Connect or Disconnect procedure includes authenticating client users, connecting or disconnecting client-specific databases, loading or storing the contents of client-specific temporary tables, or any other common actions required when each individual client connection begins or terminates.

Connect Procedure

A Connect procedure executes in whatever Application Server process handles the client connection request, before the connection request is accepted. If the Connect procedure completes with no error, the connection request from the client application is accepted. If the Connect procedure completes with an error return value (RETURN ERROR statement), the connection request is rejected.

You can specify the name of an AppServer Connect procedure using the Progress Explorer GUI or by setting the `srvrConnectProc` property in the AppServer properties file (`ubroker.properties`). If you specify this procedure, it runs as part of the connection process initiated when a 4GL client application executes the CONNECT( ) method on a server handle or when an open client executes a corresponding connection method.

You can use this procedure to authenticate a client application's request for a connection, or to perform any other initialization steps that you want executed each time a client application initiates an explicit connection request. For example, you might use it to store connection context using an application-specific context database or using the SERVER-CONNECTION-CONTEXT attribute on the SESSION handle (see the "Managing Stateless Connection Context" section). Using the SERVER-CONNECTION-ID attribute, you can uniquely identify the connection. You can then gather the context identified with the connection and store it in the context database or marshall it into the SERVER-CONNECTION-CONTEXT attribute. Any AppServer procedure that executes on behalf of the connection can restore and resave this context as appropriate. For more information, see the "Activate and Deactivate Procedures" section.

If you specify a Connect procedure, you must define it with three character input parameters. The following is an example of how you can define these input parameters.

```
DEFINE INPUT PARAMETER user-id AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER password AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER app-server-info AS CHARACTER NO-UNDO.
```

When the AppServer Connect procedure runs, the values of these three parameters are the same as the corresponding parameters specified via the CONNECT( ) method. If a value is not specified for a parameter to the CONNECT( ) method, the AppServer Connect procedure receives the unknown value (?) for that parameter.

Progress does not examine the values passed from the CONNECT() method to an AppServer Connect procedure. The name of each character input parameter might suggest the content you define for it. For example, the user-id can contain user name information, the password can contain password information, and the app-server-info can contain other user-defined information needed by the AppServer Connect procedure to complete a connection. However, the requirement to specify a particular value for the CONNECT( ) method is based only on the implementation of your AppServer Connect procedure. After examining the values provided to the AppServer Connect procedure, you can reject the connection request from the client application by terminating the procedure with the RETURN ERROR statement.

Disconnect Procedure

A Disconnect procedure executes in whatever Application Server process handles a client disconnection request, but before the connection is terminated. You can use the AppServer Disconnect procedure to specify logic that you want executed at this time. For example, you might want to use it to disconnect from a client-specific database or from other AppServers that were connected on behalf of the client.

You can specify the name of an AppServer Disconnect procedure using the Progress Explorer GUI or by setting the `srvrDisconnProc` property in the AppServer properties file (`ubroker.properties`). This procedure takes no parameters.

For an AppServer running in state-reset or state-aware operating mode, the Disconnect procedure runs after the AppServer returns the response to the client disconnection request to the client application. For a stateless AppServer, the procedure runs just before the AppServer returns this disconnection response, where it is typically used to delete connection-specific context. Any raised conditions that you do not handle in this procedure are ignored by Progress.

Usage Requirements

You can specify Connect and Disconnect procedures for all three AppServer operating modes (state-reset, state-aware, and stateless).

If the AppServer operating mode is state-reset or state-aware, the Connect procedure runs as a persistent procedure. If the AppServer operating mode is stateless, the Connect procedure runs as a non-persistent procedure. The Disconnect procedure runs as a non-persistent procedure in all operating modes.

If the Connect procedure is executed as a persistent procedure, and you do not need its context, you can delete it by executing the DELETE PROCEDURE (or DELETE OBJECT) statement on the procedure handle associated with the Connect procedure. If the AppServer operating mode is state-reset, the persistent Connect procedure is automatically deleted when a new connection is assigned to the Application Server process.

Note that if the AppServer is running in stateless operating mode, there is no guarantee that the Application Server process that runs the Connect procedure for the client application will be the same one that actually processes requests for the client application. Thus, any context that a Connect procedure establishes for a stateless connection must be stored in a file, database, SERVER-CONNECTION-CONTEXT attribute of the SESSION handle (see the "Managing Stateless Connection Context" section), or some other resource that is accessible through the 4GL.

Note that both the Connect and Disconnect procedures are optional. There is no requirement to specify either or both of them. Additionally, the content of these procedures is completely application specific, and the AppServer architecture places no constraints on it. Any 4GL logic that is valid within an Application Server process is valid for these procedures.

4.3.3 Activate and Deactivate Procedures

Activate and Deactivate procedures encapsulate logic that executes automatically for certain client requests on an AppServer running in stateless operating mode. (For information on the stateless operating mode, see the sections on operating modes in Chapter 2, "Overview of the Progress AppServer.") When a client application sends remote procedure requests to a stateless AppServer, each request might execute in a different Application Server process instance. To maintain application continuity between Application Server process s, you might need to establish some application-specific resources or context before and discard the resources and context after each request. The Activate and Deactivate procedures help to manage these resources and context more easily.

Activate Procedure

The Activate procedure executes immediately before a remote procedure request when the connection is in the unbound state. A typical use of the Activate procedures is to retrieve connection context using an application-specific context database or the SERVER-CONNECTION-CONTEXT attribute on the SESSION handle (see the "Managing Stateless Connection Context" section). Using the SERVER-CONNECTION-ID attribute, you can uniquely identify (key) the context in the context database. You can then retrieve the connection-identified context from the context database or unmarshall it from the SERVER-CONNECTION-CONTEXT attribute value. You can create the initial context for the connection using the Connect procedure. For more information, see the "Connect and Disconnect Procedures" section.

You can specify the name of an AppServer Activate procedure using the Progress Explorer GUI or by setting the `srvrActivateProc` property in the AppServer properties file (`ubroker.properties`). If you specify this procedure, it accepts no parameters and runs as a non-persistent procedure on any Application Server process that executes a remote procedure request for an unbound connection.

NOTE: The Activate procedure only runs prior to remote procedure requests. It does not run in response to connection or disconnection requests.

For the purposes of error handling, the Activate procedure is considered part of the remote procedure request. If the Activate procedure completes with no 4GL termination condition (ERROR, QUIT, or STOP), the remote procedure executes immediately afterward. If the Activate procedure completes with a termination condition, it returns to the client application as if the remote procedure had generated the termination condition. Any return values (using the RETURN [ERROR] *string* statement) are passed to the client and the original remote procedure request is not executed.

Deactivate Procedure

The Deactivate procedure executes immediately after remote procedure and delete procedure requests when the connection is in the unbound state. A typical use of the Deactivate procedures is to store connection context using an application-specific context database or using the SERVER-CONNECTION-CONTEXT attribute on the SESSION handle (see the "Managing Stateless Connection Context" section). Using the SERVER-CONNECTION-ID attribute, you can uniquely identify (key) the context in the context database. You can then gather the context identified with the connection and store it in the context database or marshall it into the SERVER-CONNECTION-CONTEXT attribute.

You can specify the name of an AppServer Deactivate procedure using the Progress Explorer GUI or by setting the `srvrDeactivateProc` property in the AppServer properties file (`ubroker.properties`). If you specify this procedure, it accepts no parameters and runs as a non-persistent procedure after the request executes, but before any response is returned to the client. Also, the Deactivate procedure runs only if the connection is unbound when the request completes execution.

A stateless connection can be in the unbound or the bound state when a request is initiated and can change to the opposite state before the request completes execution. If the connection ends up in the bound state during execution of the request, the Deactivate procedure does not run when the request completes. However, if the connection ends up in the unbound state, the Deactivate procedure runs even if the original request completes with some 4GL termination condition (ERROR, QUIT, or STOP).

After a delete procedure request, the Deactivate procedure runs only if the connection is unbound. The connection can become unbound only if both of the following are true:

- The deleted procedure was the last instantiated remote persistent procedure for the connection.
- The SERVER-CONNECTION-BOUND-REQUEST attribute has a value of FALSE.

For more information on how stateless connections transition between bound and unbound states, see the "Managing Stateless Connection Context" section.

NOTE: The Deactivate procedure only runs following remote procedure and delete procedure requests. It does not run in response to new connection or disconnection requests.

For the purposes of error handling, only return values from the request (or any executed Activate procedure) are passed to the client. Any return values generated by the Deactivate procedure, itself, are ignored.

Usage Requirements

As described in the previous sections, Activate and Deactivate procedures run only on behalf of connections in the unbound state. If a remote procedure request executes over a bound connection and the connection becomes unbound during execution of this procedure, the Deactivate procedure runs when the request completes with or without error.

Note that both the Activate and Deactivate procedures are optional. There is no requirement to specify either or both of them. Additionally, the content of these procedures is completely application specific, and the AppServer architecture places no constraints on it. Any 4GL logic that is valid within an Application Server process is valid for these procedures.

4.4 Controlling AppServer Entry Points

You can control AppServer entry points for an Application Server process at run time using the EXPORT( ) method on the SESSION handle. AppServer *entry points* are the pathnames of procedures in the AppServer PROPATH that a client application can execute as remote procedures (persistently or non-persistent). The EXPORT( ) method establishes entry points by allowing you to set and maintain an *export list* that contains the pathnames of authorized remote procedures. As such, this is a mechanism that you can use to enforce secure access to the AppServer.

The scope of the export list set by the EXPORT( ) method is the Application Server process in which the method executes. As result, when and where you need to set the export list depends both on your AppServer operating mode and the requirements of your application. If you never set an export list using the EXPORT( ) method, client applications can call any 4GL procedure on the AppServer PROPATH as a remote procedure.

This section describes how the AppServer uses an export list to limit client access to the AppServer and how you can use the EXPORT( ) method to set them. For more information on how the AppServer operating mode can affect export list settings, see the information on AppServer security in Chapter 6, "Design and Implementation Considerations."

4.4.1 Export List Operation

When an Application Server process receives a remote procedure request from a client application, the requested procedure is first filtered by the export list. Only if the procedure is declared within this list can it be run as a remote procedure request. Otherwise, the remote procedure request is rejected and an error is returned to the client application.

4.4.2 Setting and Resetting the Export List

You can invoke the EXPORT( ) method at any time, as many times as you need to, within an AppServer session. When you invoke the method, you have the option to pass it a comma-separated list of procedure names. Any format for a procedure name that is valid for the 4GL RUN statement is also valid when you call the EXPORT( ) method. Each time you invoke the method, any procedure names that you specify are added to the current export list. If you invoke the method without a parameter (with no list of procedure names), the export list is reset to empty, as if the EXPORT( ) method were never called.

NOTE: If you do specify a procedure name list, the EXPORT( ) method does not validate the procedure names that you provide. Validation of a procedure name occurs only during the remote procedure request, after the name has been filtered by the export list.

To set export lists consistently for all client connections requires a different sequence of EXPORT( ) method settings depending on the operating mode of your AppServer. For more information on setting export lists for different AppServer operating modes, see Chapter 6, "Design and Implementation Considerations."

4.4.3 Calling the EXPORT( ) Method

There is no requirement to execute the EXPORT( ) method within an AppServer session. If the EXPORT( ) method is not called, or the export list is at any point reset to empty, then all remote procedure requests are accepted for execution.

Setting an Initial Export List

The following code example shows how you might invoke the EXPORT( ) method from within an AppServer Connect procedure. In this example, two procedures, `order.p` and `stock.p`, are added to the export list.

```
DEFINE INPUT PARAMETER user-id AS CHARACTER.
DEFINE INPUT PARAMETER password AS CHARACTER.
DEFINE INPUT PARAMETER app_server_info AS CHARACTER.

DEFINE VARIABLE ret AS LOGICAL.

/* authenticate user and call EXPORT appropriately for this user. */
.
.
.

ASSIGN ret = SESSION:EXPORT("order.p,stock.p").
```

When this code executes, it adds the two procedures to the export list. A remote procedure request to an Application Server process that executes this AppServer Connect procedure executes itself only if the request is for `order.p` or `stock.p`. Otherwise, the remote procedure request returns an error.

Export list entries can also include names with an asterisk (*). The asterisk is a wildcard character to indicate that any group of characters is acceptable as a replacement for the asterisk.

For example, you can reference all procedures in a directory such as `stocksys/*.p`, some procedures in a directory such as `acctpay/qry*.p`, or all procedures in a library such as `order.pl<<*>>`.

```
ret = SESSION:EXPORT("stocksys/*.p,acctpay/qry*.p,orders.pl<<*>>").
```

After this code executes, all procedures located in the `stocksys` directory, some procedures located in the `acctpay` directory, and all procedures located in the orders procedure library are added to the export list.

If you create an export list, keep in mind that a client application is unaware that the procedures that it is submitting are being validated against this list. When the Application Server process rejects a procedure for not being on this list, it returns an error to the client application.

Resetting the Export List

At any time while servicing a connection, you can modify any existing export list by adding to the list or resetting the list. Thus, for example, if the list contains "order.p,stock.p", you can add "payment.p":

```
ret = SESSION:EXPORT("payment.p").
```

This, then, expands the list to "order.p,stock.p,payment.p".

If at any point, you want to completely reset the list, you can do this:

```
ret = SESSION:EXPORT(). /* Reset to empty */
ret = SESSION:EXPORT("special-order.p,special-stock.p").
```

At this point, the Application Server process only accepts remote procedure calls to `special-order.p` and `special-stock.p`.

4.4.4 Resolving Remote Procedure Names

If a procedure is on the export list, or if no export list is set, the Application Server process determines which procedure to run using the PROPATH setting and standard procedure name resolution rules that apply to the RUN statement.

Also, the following issues can affect how procedure names are resolved:

- The directory in which an Application Server process starts might affect how a procedure name is resolved because remote procedure requests and the PROPATH setting might contain unqualified (relative) pathnames.
- An Application Server process runs in the same directory as the Application Broker that starts it. Therefore, carefully consider what directory the Application Broker starts in because procedure names might be resolved relative to this directory. See Chapter 3, "Administration." for more information on setting up Application Brokers.

For more information on the PROPATH environment variable, see Chapter 3, "Administration." For more information on the standard name resolution rules, see the RUN statement entry in the *Progress Language Reference*.

4.5 Managing Stateless Connection Context

On a state-reset or state-aware AppServer, all remote procedures executed over the same connection run in the same Application Server process. The state-aware AppServer session is dedicated to the client connection and its state at the initiation of a client request is exactly as it was when the previous request on the client connection finished.

However, stateless AppServer sessions generally require more complex management than sessions on a state-reset or state-aware AppServer. The trade-off is potentially higher throughput for the effort. Because any Application Server process in a stateless AppServer pool can handle any remote procedure request for any unbound connection, the context created by one remote procedure request is generally unavailable to the next request in the connection sequence. To help you manage context in a stateless environment, the Progress AppServer provides the following mechanisms:

- SERVER-CONNECTION-CONTEXT attribute
- SERVER-CONNECTION-ID attribute
- SERVER-CONNECTION-BOUND-REQUEST and SERVER-CONNECTION-BOUND attributes

4.5.1 Using the SERVER-CONNECTION-CONTEXT Attribute

When a client application establishes a connection with an AppServer, the Application Broker creates a connection context area in memory for the connection. An AppServer procedure can access this context area using the SERVER-CONNECTION-CONTEXT attribute of the AppServer SESSION handle. This is a readable and writable attribute of type CHARACTER. The initial value of SERVER-CONNECTION-CONTEXT is the unknown value (?).

When any remote procedure executes, it can set the SERVER-CONNECTION-CONTEXT attribute to any application-specific value. Each time that Progress assigns a client request to an Application Server process, Progress ensures that the SERVER-CONNECTION-CONTEXT attribute is set to the last value assigned in a previous Application Server process that serviced the same client connection. This value is also available to any Connect procedure, Activate procedure, Deactivate procedure, or Disconnect procedure that you configure for the AppServer. Thus, each Application Server process that services a client connection can pass context information to the next.

For more information on the Connect, Activate, Deactivate, and Disconnect procedures, see the "Using AppServer Configuration Procedures" section.

4.5.2 Using the SERVER-CONNECTION-ID Attribute

For more complicated context than you might want to store in the SERVER-CONNECTION-CONTEXT attribute, you might choose to use a context database. In this case, you can use the SERVER-CONNECTION-ID attribute as a primary key to help you store and retrieve the context for each client connection. A context database is an application database that is connected to and accessible from each Application Server process on the AppServer.

Because the SERVER-CONNECTION-ID is a globally unique ID, Progress ensures that no two connections on a computer network have the same value. Before a request is executed within an AppServer session, Progress also ensures that the SERVER-CONNECTION-ID is set to the ID for the connection on which the request was received. This value is also available to any Connect procedure, Activate procedure, Deactivate procedure, or Disconnect procedure that you configure for the AppServer.

For more information on the value that Progress sets for the SERVER-CONNECTION-ID attribute, see Chapter 2, "Overview of the Progress AppServer."

4.5.3 Using the SERVER-CONNECTION-BOUND-REQUEST Attribute

As explained earlier (see Chapter 2, "Overview of the Progress AppServer."), a connection from client to a stateless AppServer is in one of two states:

- **Unbound**—Where remote procedure requests over the connection can be handled by any available Application Server process running on the AppServer.
- **Bound**—Where remote procedure requests over the connection are handled by a single Application Server process dedicated to the connection.

By default, a connection on a stateless AppServer is unbound.

Making a Connection Bound

An unbound connection can become bound in two ways:

- A client application can call a remote persistent procedure (see Chapter 5, "Programming the Client Application"). This call runs and instantiates a persistent procedure in the Application Server process that handles the remote procedure request. The connection thus becomes bound, dedicating this Application Server process to handle all future client requests over the bound connection.
- An Application Server process handling a request from a client can run a procedure that sets the SERVER-CONNECTION-BOUND-REQUEST attribute on the SESSION handle to TRUE. The connection thus becomes bound, dedicating the Application Server process that sets this attribute to handle all future client requests over the bound connection.

Note that it is the successful instantiation of a remote persistent procedure that forces the connection to transition from the unbound state to the bound state. If a client fails in its attempt to instantiate a remote persistent procedure on an unbound connection, the connection remains unbound.

Making a Connection Unbound

You can request that a connection become unbound by setting the SERVER-CONNECTION-BOUND-REQUEST attribute on the SESSION handle to FALSE. However, the connection only becomes unbound as long as there are no remote persistent procedures still active in the Application Server process.

Once a connection transitions from the bound to the unbound state, the Application Server process can handle the next request sent by any connected client. Conversely, the next request sent by the client on the previously bound connection can be handled by any Application Server process.

If remote persistent procedures are active on a bound connection, setting the SERVER-CONNECTION-BOUND-REQUEST attribute to FALSE has no effect until all remote persistent procedures in the Application Server process are explicitly deleted. When all remote persistent procedures are deleted, the connection either remains in the bound state or becomes unbound depending on the last setting of the SERVER-CONNECTION-BOUND-REQUEST attribute.

4.5.4 Using the SERVER-CONNECTION-BOUND Attribute

A stateless connection can be in a bound state because of active remote persistent procedures even if the SERVER-CONNECTION-BOUND-REQUEST attribute is set to FALSE. To easily determine if a connection is bound, even if the SERVER-CONNECTION-BOUND-REQUEST attribute is set to FALSE, check the value of the SERVER-CONNECTION-BOUND attribute. When set to TRUE for an Application Server process running on a stateless AppServer, the read-only SERVER-CONNECTION-BOUND attribute indicates that the connection is currently bound, either from having active remote persistent procedures or from having the SERVER-CONNECTION-BOUND-REQUEST attribute set to TRUE.

NOTE: If the AppServer is running in state-aware or state-reset operating mode, the SERVER-CONNECTION-BOUND attribute for each of its Application Server processes is always set to TRUE.

4.6 Understanding Remote Procedure Handles

The THIS-PROCEDURE system handle returns the current procedure handle value from within any executing block of an external procedure. The THIS-PROCEDURE handle created within an AppServer session for a remote procedure request is a *remote procedure handle*. Its REMOTE attribute is, therefore, set to TRUE. If the REMOTE attribute is FALSE, then the current procedure exists as the result of a *local procedure* call (that is, a procedure call initiated and executed within the current Progress session context).

4.6.1 Remote Persistent Procedure Handles

When a 4GL client application executes a remote persistent procedure, two persistent procedure handles are created: one within the client application session and another separate handle within the AppServer session where the persistent procedure is created. Progress internally maintains a mapping between the two handles. The handle within the client application is a *proxy persistent procedure handle*, and its PROXY attribute and PERSISTENT attribute are set to TRUE. The corresponding handle within the Application Server process is a *remote persistent procedure handle*, and its REMOTE attribute and PERSISTENT attribute are set to TRUE.

Unlike the values of persistent procedure handles and the THIS-PROCEDURE handle that reference the same local procedure context, the proxy persistent procedure handle and the remote persistent procedure handle are truly separate handles. For example, setting the PRIVATE-DATA attribute on a remote persistent procedure handle has no effect on the PRIVATE-DATA attribute of the corresponding proxy persistent procedure handle in the client application.

For more information on the relationship between remote and proxy persistent procedure handles, see the information on procedure handles in Chapter 6, "Design and Implementation Considerations."

4.6.2 Deleting Remote Persistent Procedures

As with local persistent procedures, a remote persistent procedure context remains active within an AppServer session until it is deleted using the DELETE OBJECT or DELETE PROCEDURE statement. You can thus delete a remote persistent procedure in the AppServer session by deleting its remote procedure handle. If the delete occurs in the context of a remote procedure request, the deletion is pending until the request completes and returns to the client. When the remote persistent procedure is finally deleted, both its proxy procedure handle on the client and its remote procedure handle in the Application Server process are deleted together.

4.7 Handling Conditions and Returning Values

Conditions raised in an AppServer session have varied effects, depending on the condition. You can also return values to the client application from a remote procedure, with or without raising an ERROR condition, similar to returning values from a local procedure. The following sections describe how each of these standard Progress mechanisms can affect an AppServer session and any client remote procedure request.

4.7.1 Raising the ERROR Condition and Returning Values

You can return application exceptions, with or without associated values, to client applications by executing the RETURN ERROR statement. The effect is similar to a local procedure return. When you execute this statement in any remote procedure (persistent or non-persistent), the procedure:

1. Terminates.
2. Raises the ERROR condition on the RUN statement in the client (or an equivalent exception in an Open Client).
3. Returns any specified value to the client application.

An ERROR condition raised in an Activate procedure for a stateless AppServer session has the same effect as if it were raised in the remote procedure for the request.

To return a value without raising the ERROR condition, use the RETURN statement without the ERROR option.

4.7.2 Raising the ENDKEY Condition

Like a 4GL client, an AppServer session raises the ENDKEY condition if an attempt to find a database record fails or an INPUT FROM statement reaches the end of an input stream. Like a batch client, there are no user-interface events that can raise the ENDKEY condition.

4.7.3 Handling the STOP Condition

If a client application raises the STOP condition while a remote procedure is active on a connected AppServer, Progress also raises the STOP condition in the context of the Application Server process handling the remote procedure request.

NOTE: A client can raise the STOP condition in the context of an executing remote procedure by invoking the STOP( ) method on the server object handle.

A STOP condition raised within a remote procedure does not necessarily terminate procedure execution, and it never causes the AppServer session to restart (as in a 4GL client session). You can use the ON STOP statement within the remote procedure to intercept any STOP condition raised during execution and handle it as your application requires. Otherwise, the STOP condition propagates to the client application. For information on unhandled STOP conditions, see the "Effects of Unhandled Conditions" section.

A STOP condition raised in an Activate procedure for a stateless AppServer session has the same effect as if it were raised in the remote procedure for the request.

4.7.4 Handling the QUIT Condition

You can use the ON QUIT statement within a remote procedure to intercept any QUIT condition raised during execution and handle it as your application requires. Otherwise, the QUIT condition propagates to the client application. For information on unhandled QUIT conditions, see the "Effects of Unhandled Conditions" section.

A QUIT condition raised in an Activate procedure for a stateless AppServer session has the same effect as if it were raised in the remote procedure for the request.

4.7.5 Effects of Unhandled Conditions

An unhandled ERROR or ENDKEY condition raised in an Application Server process has no effect on the client application to which it is connected. The handling of each condition conforms to standard Progress rules. For more information about these rules, see the *Progress Programming Handbook*.

Two Progress conditions (STOP and QUIT), if not handled using the ON STOP and ON QUIT statements in an AppServer session, have definite effects on any client application for which the Application Server process is handling a request.

Any unhandled STOP causes the remote procedure request (persistent or non-persistent) to terminate and propagate the STOP condition to the client application.

Any unhandled QUIT condition causes the remote procedure request to terminate and return successfully without any propagated condition to the client. However, Progress also terminates the client connection to the AppServer.

4.8 Managing Transactions

You can manage transactions in an AppServer session much the same way that you can manage them in a 4GL client session, using transaction blocks and statements that update tables in a database. However, because client and AppServer sessions are entirely separate, including their transaction contexts, the client normally has no direct ability to influence the course of an AppServer transaction, which begins and ends within the scope of a single remote procedure request. To allow finer-grained control of transactions running on the AppServer, Progress provides an additional transaction type that allows a transaction on the AppServer to span multiple client requests.

4.8.1 Types of AppServer Transactions

An AppServer can allow a client to invoke two different types of transactions in an Application Server process. The two types of transaction are distinguished by scope (duration) and include the:

- Normal 4GL transaction
- Automatic transaction

Normal 4GL Transactions

A normal 4GL transaction has the scope of part or all of a single remote procedure call. That is, the transaction begins and ends during execution of a single remote procedure request. Thus, all control of the transaction is handled by the Application Server process within a single request.

Normal 4GL transactions take full advantage of Progress built-in transaction management features, such as 4GL-integrated retry and undo (rollback) mechanisms. Normal 4GL transactions yield the smallest possible transaction unit, providing the most efficient handling of record conflicts and transaction resolution. For most applications, this is the only type of transaction that you need to use. It is also the only type of transaction available to previous versions of the Progress AppServer. For more information on managing normal 4GL transactions on an AppServer, see the "Implementing Normal 4GL Transactions" section in this chapter.

Automatic Transactions

An *automatic transaction* allows the transaction to span more than one remote procedure call. That is, once an automatic transaction is started through some remote procedure call, you can explicitly terminate it in the same procedure call or in a subsequent procedure call running in the same AppServer session. To control this type of transaction, the AppServer provides attributes and methods that you can use to specify how and when you want the transaction to end.

With automatic transactions, transactions tend to grow large, with greater potential for lock conflicts and other concurrency problems. (For more information on small and large transactions, see the *Progress Programming Handbook*.) Even so, automatic transactions can be extremely useful where the application requires the transaction to span multiple AppServer requests. This is especially true where client response is entirely programmatic and does not depend on user interaction for a transaction to complete.

An example of a useful application might be where a transaction commit depends on a real-time status from a process control or financial trading system. Depending on this status, the AppServer might require additional information from the client to resolve the transaction. As long as the client does not have to wait to make this information available to the AppServer, the impact of the automatic transaction on system performance can be minimal. For more information on managing automatic transactions, see the "Implementing Automatic Transactions" section in this chapter.

4.8.2 Implementing Normal 4GL Transactions

You can implement a normal 4GL transaction in any AppServer procedure that the client runs either directly or indirectly. The 4GL for starting a normal 4GL transaction in an AppServer session is identical to the 4GL required to start transactions in a Progress client session. In any case, a normal 4GL transaction starts and ends entirely within the context of the AppServer procedure that implements it. Because this type of transaction completes within the scope of a single procedure call, you can implement a normal 4GL transaction on an AppServer running in any operating mode (state-reset, state-aware, or stateless).

4.8.3 Implementing Automatic Transactions

To implement an automatic transaction, you provide a specially-coded remote procedure (*transaction initiating procedure*) that initializes a transaction object in the Application Server process where it executes. This *transaction object* is a permanent object in the session context that provides methods and attributes on a transaction object handle that allow the AppServer session to commit, rollback, or set and check the status of an automatic transaction. Because an automatic transaction terminates when the context of the transaction initiating procedure is deleted, you must execute the transaction initiating procedure persistently to make practical use of the transaction that it initiates.

Initializing the Automatic Transaction Object

To initialize the transaction object, you must create a transaction initiating procedure by specifying the TRANSACTION-MODE statement as the first executable statement in the procedure file:

SYNTAX

```
TRANSACTION-MODE AUTOMATIC [ CHAINED ]
```

When the client calls this transaction initiating procedure as a remote persistent procedure, an automatic transaction starts in the context of the AppServer session and can remain open after the remote procedure call returns to the client. The context for this open automatic transaction is the entire AppServer session, not (as you might think) the context internal to the transaction initiating procedure instantiated in the AppServer session. Thus, any remote procedure (persistent or non-persistent) that is subsequently executed participates in this open automatic transaction.

Controlling the Transaction

You can access the transaction object handle for the initialized transaction object using the TRANSACTION attribute on the procedure of any procedure executing in the AppServer session by using the THIS-PROCEDURE system handle:

```
DEFINE VARIABLE hTran AS HANDLE NO-UNDO.
hTran = THIS-PROCEDURE:TRANSACTION.
```

You can then access the transaction object methods and attributes to control the transaction, as described in Table 4–2.

Table 4–2: Transaction Handle Methods and Attributes *(1 of 2)*

Method/Attribute	Return/Data Type	Description
DEFAULT-COMMIT	LOGICAL	A writable attribute whose setting tells the transaction object how to complete any open automatic transaction if the transaction terminates with no prior invocation of the SET-COMMIT( ) or SET-ROLLBACK( ) method (for example, if the transaction initiating procedure is deleted). Setting it to TRUE ensures that the transaction is committed. Setting it to FALSE ensures that the transaction is rolled back. The default value is FALSE.
IS-OPEN	LOGICAL	A read-only attribute that returns TRUE if a transaction is active. The function of this attribute is identical to the 4GL TRANSACTION function. See also the TRANS-INIT-PROCEDURE attribute.
SET-COMMIT( )	LOGICAL	Tells the transaction object to commit any active automatic transaction after the current request returns execution to the client. If no automatic transaction is active, the method returns FALSE. Note that you **cannot** invoke this method after invoking a prior SET-ROLLBACK( ) within the same client request.

Table 4–2: Transaction Handle Methods and Attributes *(2 of 2)*

Method/Attribute	Return/Data Type	Description
SET-ROLLBACK( )	LOGICAL	Tells the transaction object to roll back any active automatic transaction after the current request returns execution to the client. If no automatic transaction is active, the method returns FALSE. Note that you **can** invoke this method after invoking a prior SET-COMMIT( ) within the same client request.
TRANS-INIT-PROCEDURE	HANDLE	A read-only attribute that returns the procedure handle to the transaction initiating procedure for any active automatic transaction. If no automatic transaction is active this attribute returns an invalid handle. (Validate with the VALID-HANDLE function.)

Terminating Automatic Transactions

An automatic transaction remains active as long as the context of the transaction initiating procedure remains active and you do not otherwise terminate the transaction using a transaction handle method. Thus, you can terminate an automatic transaction using two techniques:

- **Explicit termination**—You can explicitly terminate an automatic transaction on the AppServer with a reference to the SET-COMMIT( ) method or the SET-ROLLBACK( ) method:

```
hTran = THIS-PROCEDURE:TRANSACTION.
hTran:SET-COMMIT().
```

```
hTran = THIS-PROCEDURE:TRANSACTION.
hTran:SET-ROLLBACK().
```

As long as an automatic transaction is open, you can execute any internal procedure of the transaction initiating procedure from any other procedure running in the AppServer session. However, if no transaction is open, any such attempt to call these internal procedures from within the Application Server process context returns an error. Without an open transaction, only a client application can call a remote internal procedure of the transaction initiating procedure (see the "Restarting Automatic Transactions" section).

NOTE: You can determine whether an automatic transaction is open in an AppServer session by either checking for an error after executing an internal procedure of the transaction initiating procedure or by checking the value of the TRANS-INIT-PROCEDURE attribute.

- **Implicit termination**—You can also terminate the automatic transaction from either the client or the AppServer session by deleting the proxy persistent procedure handle (on a 4GL client) or remote persistent procedure handle (on the AppServer) of the transaction initiating procedure. If you delete the transaction initiating procedure, the transaction commits or rolls back depending on the value of the transaction handle DEFAULT-COMMIT attribute.

Restarting Automatic Transactions

After an automatic transaction terminates, how an automatic transaction can restart depends on how it terminates. If the transaction terminates by deleting the transaction initiating procedure, only a client application can restart an automatic transaction by again remotely instantiating a transaction initiating procedure.

If a transaction initiating procedure is still active in the AppServer session, you can support the restarting of an automatic transaction using one of the following techniques, depending on how you code the transaction initiating procedure:

- **Automatic (chained) restarting**—If you specify the CHAINED option for the TRANSACTION-MODE AUTOMATIC statement, a new automatic transaction starts immediately after the previous automatic transaction commits or rolls back. Thus, with the CHAINED option an automatic transaction is always open in the AppServer session until the transaction initiating procedure, itself, is made inactive (is deleted).

- **Manual restarting**—If you do not specify the CHAINED option for the TRANSACTION-MODE AUTOMATIC statement, the client application can cause another automatic transaction to restart by calling any internal procedure of the active transaction initiating procedure. Thus, a remote call to an empty internal procedure can start a new automatic transaction.

CAUTION: In general, try to keep automatic transactions on an AppServer as short as possible. The multi-request feature of automatic transactions tend to encourage long transactions, and long transactions can make large or important segments of a database inaccessible to other Progress sessions for an indefinite period. This can create database deadlocks and traffic jams that can seriously retard the performance of multiple sessions. For more information on techniques to shorten automatic transactions, see the information on transaction management considerations in Chapter 6, "Design and Implementation Considerations."

Automatic Transaction Example

By including appropriate calls to internal procedures of the transaction initiating procedure in your remote procedures, you can encapsulate the management of a multi-request automatic transaction in a manner completely hidden from the client. However, the following example is relatively simple in that the client calls all of the internal procedures of the transaction initiating procedure directly, thus more openly exposing the transaction mechanism to the client.

The following two procedures include a transaction initiating procedure and a 4GL client procedure that accesses it.

The transaction initiating procedure (`a-txtest.p`) implements an unchained automatic transaction. This procedure provides three internal procedures that allow the client to explicitly start (startme), commit (stopme), and roll back (abortme) transactions.

a-txtest.p

```
TRANSACTION-MODE AUTOMATIC.  /*this statement makes this procedure a */
                             /*transaction initiating procedure          */

PROCEDURE startme:  /*this empty internal proc starts a transaction*/
END.

PROCEDURE stopme: /*this internal proc arranges for a commit*/
  DEFINE VARIABLE x AS HANDLE.
  x = THIS-PROCEDURE:TRANSACTION.
  x:SET-COMMIT().
END.

PROCEDURE abortme: /*this internal proc arranges for a rollback*/
  DEFINE VARIABLE x AS HANDLE.
  x = THIS-PROCEDURE:TRANSACTION.
  x:SET-ROLLBACK().
END.
```

The client procedure (a-txclnt.p) establishes the initial transaction object by instantiating a-txtest.p on the AppServer. After calling two database update procedures, the client commits the transaction with a call to stopme. It then begins a new transaction by calling startme, and after calling the same two database update procedures, rolls back the transaction by calling abortme.

As noted in the comments a-txtest.p establishes an unchained transaction object. If the procedure instead establishes a chained transaction object, each time the current transaction terminates (stopme, abortme), the Application Server process automatically starts a new transaction with no need for the client to call a procedure (startme) to initiate it.

a-txclnt.p[1]

```
DEFINE VARIABLE ph AS HANDLE.
/*create a transaction initiating procedure and start the transaction*/
RUN a-txtest.p ON SERVER h TRANSACTION DISTINCT PERSISTENT SET ph.

RUN custupdate.p ON SERVER h TRANSACTION DISTINCT.
RUN orderupdate.p ON SERVER h TRANSACTION DISTINCT.

RUN stopme IN ph.  /*causes the transaction to commit including both
                     custupdate and orderupdate*/

/*start another transaction by running an internal proc of a-txtest.p*/
RUN startme IN ph.  /*starts a new transaction -- this is
                      unnecessary if the TRANSACTION-MODE statement
                      in a-txtest.p specifies the CHAINED option*/

RUN custupdate.p ON SERVER h TRANSACTION DISTINCT.
RUN orderupdate.p ON SERVER h TRANSACTION DISTINCT.

RUN abortme IN ph.  /*causes the transaction to roll back including
                       both custupdate and orderupdate*/

DELETE PROCEDURE ph.
```

[1] The RUN statements in this example use the 4GL syntax for executing remote procedures of various types. For more information, see Chapter 5, "Programming the Client Application."

Note that while these examples show a close correlation between transaction directives and remote procedure requests, there is no requirement that this be so. You can completely hide any sense that the client is ultimately initiating and managing a transaction by hiding all calls to the internal procedures of the transaction initiating procedure within the logic of your application API. For example, you can encapsulate the remote procedure calls in a-txclnt.p in higher-level procedures, such as start-updates.p, rerun-updates.p, and end-updates.p.

4.9 Programming for Open Client Applications

In general, programming AppServer procedures for open client (Java and ActiveX Controller) applications is not much different than programming for 4GL clients. However, for each AppServer persistent procedure that you expect to be instantiated as an open client object, you must also specify the prototypes defined by all super procedures, including user-defined functions, that your persistent procedure uses. The reason for this is that the Progress Open Client Toolkit, which generates proxy objects for use by open client applications, analyzes the r-code to help determine class definitions for proxy objects. This r-code analysis includes any super procedure references required to define equivalent method overrides for each class.

You can specify super procedure prototypes in one of two ways, depending on how you write your procedures:

- Using the ProtoGen utility available from the Progress Application Development Environment (ADE).
- Using a text editor or the Procedure Editor

4.9.1 Using the ProtoGen Utility

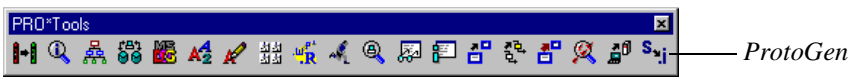
The ProtoGen utility analyzes the source code of any super procedure that you specify and generates an include file containing the prototypes for all of its internal procedures and user-defined functions. You can access this utility from the ProTools palette, which is available from the Tools menu in the ADE.

One of the more common applications of this utility is to generate prototype include files for persistent procedures that you build in the AppBuilder, especially if you build them using SmartObject technology.

Thus, to include super procedure prototypes in AppBuilder procedures:

1 ♦ Code each super procedure sufficiently to specify all PROCEDURE and FUNCTION statements that it requires.

2 ♦ From the ADE, choose ProTools from the Tools menu, and click the ProtoGen button:



3 ♦ In the Prototype Generator dialog box, enter the filename of the super procedure and the name of the include file you want to generate from it.

4 ♦ Generate the include file and, using the AppBuilder, insert a reference to the include file within every persistent procedure and SmartObject that relies on the specified super procedure prototypes.

5 ♦ Compile and generate the r-code for each persistent procedure and SmartObject.

Once you compile and generate the r-code, the open client developer can use the Open Client Toolkit to generate the corresponding proxy object. For more information on using ProtoGen, see the ADE online help and for using the AppBuilder, see the *Progress AppBuilder Developer's Guide*.

4.9.2 Using Code Editors

For any persistent procedures that you write by hand, you can also use the ProtoGen utility to generate include files from any super procedure prototypes you might need. However, you might prefer or need to code these prototypes by hand, especially in a character environment.

To code the necessary prototypes, edit a persistent procedure that requires them and include a PROCEDURE statement for each internal procedure prototype and a FUNCTION statement for each user-defined function prototype using the IN SUPER option in each case. For internal procedures, you must also follow the PROCEDURE statement with any DEFINE PARAMETER statements to specify the parameters for the internal procedure and terminate the procedure block with an END statement, like a standard procedure block. For user-defined functions, you can include the parameter and return value prototypes within the FUNCTION statement itself:

```
PROCEDURE addMessage IN SUPER:
  DEFINE INPUT PARAMETER pcText AS CHARACTER.
  DEFINE INPUT PARAMETER pcField AS CHARACTER.
  DEFINE INPUT PARAMETER pcTable AS CHARACTER.
END PROCEDURE.

FUNCTION setPosition RETURNS LOGICAL
  (INPUT pdRow AS DECIMAL,
   INPUT pdCol AS DECIMAL) IN SUPER.
```

This example defines prototypes for an internal procedure, addMessage, and a user-defined function, setPosition, that are defined in some super procedure. Note that there is no indication of what super procedure these prototypes refer to. For the Open Client Toolkit, only the prototypes are required; the IN SUPER option only tells Progress that the procedure or function definition lies elsewhere. As long as the toolkit knows the prototypes, it can generate the correct proxy object definitions. Progress then locates the appropriate code to execute at run time for each super procedure prototype that the open client application references.

Again, remember to compile and generate the r-code for each persistent procedure before making it available for access by the Open Client Toolkit.

5

Programming the Client Application

This chapter describes how to program 4GL procedures that execute in a Progress session that runs as a client to an AppServer. Specifically, the sections in this chapter describe:

- 4GL for programming AppServer client procedures
- Accessing AppServer resources from a client
- Connecting an AppServer instance
- Accessing the connection ID on the client
- Running and managing remote procedures
- Disconnecting an AppServer instance
- Managing Asynchronous Requests
- Handling conditions and return values

This chapter addresses programming issues specific to AppServer 4GL client procedures. It assumes that you are generally familiar with writing and executing Progress 4GL procedures. For more information on using the Progress 4GL, see the *Progress Programming Handbook*.

While there is some reference to Open Clients, for complete information on programming Java and ActiveX Controller Open Clients, see the *Progress Open Client Developer's Guide*.

For information on programming issues specific to AppServer applications, see Chapter 4, "Programming the AppServer." For information common to programming both 4GL client and AppServer procedures, see Chapter 6, "Design and Implementation Considerations."

5.1 4GL for Programming AppServer Client Procedures

Table 5–1 lists the 4GL elements that are either valid only for accessing an AppServer connection or have special application in AppServer client programming. The remaining sections in this chapter explain how to use these elements.

Table 5–1: 4GL for Programming AppServer Procedures *(1 of 7)*

4GL Element	Description
ASYNC-REQUEST-COUNT	1) An INTEGER attribute on the server object handle that returns the number of active asynchronous requests submitted to this server. 2) An INTEGER attribute on the procedure handle that returns the number of currently outstanding asynchronous requests for this procedure. Can be non-zero only if the PROXY and PERSISTENT attributes are both set to TRUE.
Asynchronous Request Object Handle	A type of handle that maintains the status of an asynchronous request in a 4GL client application. This handle provides methods and attributes that allow you to check the status of a remote procedure (internal or external) that is called asynchronously.
CANCELLED	A LOGICAL attribute on the asynchronous request object handle that indicates if the asynchronous request was cancelled using either the CANCEL-REQUESTS( ) method or the DISCONNECT( ) method on the associated server handle.
CANCEL-REQUESTS( )	A method on the server object handle that raises a STOP condition in the context of the currently running asynchronous request and purges the send queue of any asynchronous requests that have not been executed.
CLIENT-CONNECTION-ID	A CHARACTER attribute on the server object handle that returns the connection ID for the AppServer connection associated with this server handle.

Table 5–1: 4GL for Programming AppServer Procedures *(2 of 7)*

4GL Element	Description
COMPLETE	A LOGICAL attribute on the asynchronous request object handle that indicates if the asynchronous request is completed and its result is processed on the client.
CONNECT([*connection-parameters*] [, *userid*] [, *password*] [, *appserver-info*])	A method on the server object handle that connects and associates an AppServer instance with the server handle.
CONNECTED( )	A method on the server object handle that indicates if an AppServer is currently connected and associated with the server handle.
CREATE SERVER *server-handle*	A statement that creates a server object handle and stores it in a HANDLE variable.
DEFINE INPUT PARAMETER *parameter*	A statement for defining INPUT parameters within an event procedure. Each INPUT parameter must correspond to an OUTPUT or INPUT-OUTPUT parameter of the associated asynchronous remote procedure.
DELETE OBJECT *handle*	A statement that you can use to delete certain objects, including server objects, and persistent procedures (local and remote), and asynchronous request objects.
DELETE PROCEDURE *procedure-handle*	A statement that you can use to delete both local and remote procedure objects (persistent procedures).
DISCONNECT( )	A method on the server object handle that disconnects from and removes all reference to the Progress AppServer currently associated with the server handle. Any running or pending asynchronous requests submitted by this client are also cancelled.

Table 5–1: 4GL for Programming AppServer Procedures *(3 of 7)*

4GL Element	Description
ERROR	A LOGICAL attribute on the asynchronous request object handle that indicates that an ERROR condition was returned from the AppServer as a result of processing the request.
EVENT-PROCEDURE	A CHARACTER attribute on the asynchronous request object handle that contains the name of the internal procedure to be run as the event procedure for this asynchronous request.
EVENT-PROCEDURE-CONTEXT	A HANDLE attribute on the asynchronous request object handle that contains the procedure handle of the active procedure context where the event procedure for this asynchronous request is defined.
FIRST-ASYNC-REQUEST( )	A method on the server object handle that returns the first entry in the list of all current asynchronous request handles for the specified AppServer.
FIRST-PROCEDURE	A HANDLE attribute on the server object handle that returns the first entry in the list of remote persistent procedures running on the connected AppServer.
FIRST-SERVER	A HANDLE attribute on the SESSION system handle that returns the handle to the first entry in the chain of server handles for the session.
FUNCTION ... IN *procedure-handle*	A statement that declares a user-defined function. When *procedure-handle* is a proxy persistent procedure handle, the function is defined on the connected AppServer in the specified procedure object.
LAST-ASYNC-REQUEST( )	A method on the server object handle that returns the last entry in the list of all current asynchronous request handles for the specified AppServer.
LAST-PROCEDURE	A HANDLE attribute on the server object handle that returns the last entry in the list of remote persistent procedures running on the connected AppServer.

Table 5–1: 4GL for Programming AppServer Procedures *(4 of 7)*

4GL Element	Description
LAST-SERVER	A HANDLE attribute on the SESSION system handle that returns the handle to the last entry in the chain of server handles for the session.
NEXT-SIBLING	1) A HANDLE attribute on the server object handle that returns the next entry in the list of server handles created for the current Progress session. 2) A HANDLE attribute on the proxy persistent procedure handle that returns the next entry in the list of proxy persistent procedure handles. 3) A HANDLE attribute on the asynchronous request object handle that returns the next entry in the list of asynchronous request handles for asynchronous remote procedures submitted for execution on the same AppServer.
PERSISTENT	A LOGICAL attribute on procedure handles and the THIS-PROCEDURE system handle that is TRUE if the specified procedure is running persistently as a procedure object.
PERSISTENT-PROCEDURE	A HANDLE attribute on the asynchronous request object handle that returns the proxy persistent procedure handle to the remote persistent procedure that contains the internal procedure executed for the specified asynchronous request. If the request is for an external, rather than an internal, procedure, this attribute returns an invalid handle.
PREV-SIBLING	1) A HANDLE attribute on the server object handle that returns the previous entry in the list of server handles created for the current Progress session. 2) A HANDLE attribute on the proxy persistent procedure handle that returns the previous entry in the list of proxy persistent procedure handles. 3) A HANDLE attribute on the asynchronous request object handle that returns the previous entry in the list of asynchronous request handles for asynchronous remote procedures submitted for execution on the same AppServer.

Table 5–1: 4GL for Programming AppServer Procedures *(5 of 7)*

4GL Element	Description
PROCEDURE-COMPLETE	The event returned for an asynchronous request object handle that indicates the associated remote procedure has completed execution and, as a result, causes execution of the specified event procedure as specified by the EVENT-PROCEDURE and EVENT-PROCEDURE-CONTEXT attributes.
PROCEDURE-NAME	A CHARACTER attribute on the asynchronous request object handle that provides the name of the remote procedure executed to instantiate this asynchronous request handle.
PROCESS EVENTS	A statement that you can use to handle any pending PROCEDURE-COMPLETE events for asynchronous requests. You can also use any blocking I/O statement, such as the WAIT-FOR statement.
PROXY	A LOGICAL attribute on procedure handles that is TRUE if the procedure handle is a proxy handle for a persistent procedure running remotely in the context of an AppServer.
QUIT	A LOGICAL attribute on the asynchronous request object handle that indicates a QUIT condition was returned from the AppServer as a result of processing the request.
RUN ... [PERSISTENT ...] ON [SERVER] *server-handle* [TRANSACTION DISTINCT] [ASYNCHRONOUS ...] ...	The statement that executes an external (remote) procedure on a connected AppServer specified by *server-handle*, in a transaction that is distinct from the client. With the PERSISTENT option, it instantiates the procedure as a persistent procedure object. With the ASYNCHRONOUS option, the procedure executes asynchronously with respect to the client. This option also optionally specifies the associated asynchronous request handle and event procedure to handle results of the request. If *server-handle* is the SESSION system handle, the external procedure executes synchronously in the local session.

Table 5–1: 4GL for Programming AppServer Procedures *(6 of 7)*

4GL Element	Description
RUN ... IN *procedure-handle* [ASYNCHRONOUS ...] ...	The statement that executes an internal (remote) procedure on a connected AppServer, where *procedure-handle* specifies a remote persistent procedure that is instantiated on the AppServer and that defines the internal procedure. With the ASYNCHRONOUS option, the internal procedure executes asynchronously with respect to the client. This option also optionally specifies the associated asynchronous request handle and event procedure to handle results of the request. If *procedure-handle* is the handle to a local persistent procedure, the internal procedure executes synchronously in the local session.
SELF	A system handle that, in the context of an event procedure, returns the asynchronous request object handle of the completed request for which the event procedure is executing.
SERVER	1) A HANDLE attribute on a remote persistent procedure handle that returns the server object handle to the AppServer on which the specified remote persistent procedure runs. Valid only if the PROXY and PERSISTENT attributes are both TRUE. 2) A HANDLE attribute on the asynchronous request object handle that returns the server handle of the AppServer where this asynchronous request was submitted for execution. If the request is run in the local session (using the SESSION system handle), the attribute is set to SESSION.
Server Object Handle	A type of handle that provides a connection to an AppServer in a 4GL client application. This handle provides server object methods and attributes that allow you to connect and manage an AppServer.
STOP	A LOGICAL attribute on the asynchronous request object handle that indicates a STOP condition was returned from the AppServer as a result of processing the request.

Table 5–1: 4GL for Programming AppServer Procedures *(7 of 7)*

4GL Element	Description
TYPE	1) A CHARACTER attribute on the server object handle that returns the handle type, which is "SERVER" for a server object handle. 2) A CHARACTER attribute on the asynchronous request object handle that provides the handle type, which is "ASYNC-REQUEST" for an asynchronous request object handle
WAIT-FOR ...	A statement that you can use to handle any pending PROCEDURE-COMPLETE events for asynchronous requests. You can also use PROCESS EVENTS or any other blocking I/O statement, such as the PROMPT-FOR statement.

5.2 Accessing AppServer Resources from a Client

In order to run remote procedures on an AppServer, a client application must first establish a connection with the required AppServer. When done with the connection, the client application can then disconnect from the AppServer.

There are five steps that a 4GL client application follows to connect and then run remote procedures on an AppServer:

1. Create a server object handle.
2. Use the CONNECT( ) method on the server object handle to connect an AppServer instance.
3. Invoke remote procedure requests using the RUN statement.
4. Use the DISCONNECT( ) method on the server object handle to disconnect a client application from the AppServer.
5. Delete the server object handle.

The following sections provide more information and code examples for each step in this process.

Following is a list of general points that describe the relationship between a client application and the AppServer environment.

- A client application can have connections to multiple AppServers simultaneously.
- The choice of AppServer for each connection is determined by the NameServer based on the Application Service name provided by the client. The actual location of the AppServer is completely transparent to the client.
- How the AppServer allocates Application Server processes to service requests received on the connection depends on the AppServer operating mode, which is completely transparent to the client.
- An AppServer can manage transactions in a number of ways. The transaction management mechanism used by the AppServer is completely transparent to the client. The client has no direct control over the transaction mechanics used by the AppServer.

5.3 Connecting an AppServer Instance

To connect a 4GL client to an AppServer, you must:

1. Create a server object handle
2. Execute the CONNECT( ) method on the server object handle.

5.3.1 Creating a Server Object Handle

To create a server object handle, you must define a HANDLE variable and execute the CREATE SERVER statement to store the new server object handle:

```
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
CREATE SERVER hAppSrv.
```

You can then use this handle to connect to an AppServer instance.

5.3.2 Establishing a Connection with the CONNECT( ) Method

The CONNECT( ) method takes up to four CHARACTER arguments in the following order, all of which are optional, depending on the configuration of your AppServer environment:

1. *connection-parameters*
2. *userid*
3. *password*
4. *appserver-info*

The *connection-parameters* argument is used to locate an AppServer that supports the required business function.

The *userid*, *password*, and *appserver-info* arguments are passed from the client application to the AppServer, which passes them to the Connect procedure (if defined). If a Connect procedure is not defined for the connected AppServer, these arguments are discarded. The requirement to specify these arguments depends entirely on the AppServer application.

An AppServer can reject a connection request either because the Connect procedure failed or because there are not enough system resources (for example, Application Server processes) available to service the connection.

The rest of this section describes these parameters in greater detail.

Connection Parameters Argument

The *connection-parameters* argument specifies a comma-separated list of parameters necessary to establish the AppServer connection. Table 5–2 describes the AppServer connection parameters that you can include in this list.

Table 5–2: AppServer Connection Parameters *(1 of 2)*

Connection Parameter[1]	Description
-AppService *application-service*	The name of an Application Service supported by the specified NameServer. (Defaults to the default service for the specified NameServer.)
-H [*host_name* | *IP-address*]	The network address of the NameServer machine. You can specify either the TCP/IP host name or the Internet protocol address of the machine. (Defaults to `localhost`.)

Table 5–2: AppServer Connection Parameters *(2 of 2)*

Connection Parameter[1]	Description
-S [*udp-service-name* | *port-number*]	The port number for the NameServer connection. You can specify either an explicit UDP port number or a UDP service name. If you use a UDP service name, the method uses the port number associated with that name in the TCP/IP services file. (Defaults to 5162)
-pf *filename*	A text file containing any of the other AppServer connection parameters described in this table. If this file contains any other Progress startup parameters, the method ignores them.

[1] Previous versions of the AppServer allow you to include a network protocol using the -N parameter, which must always specify TCP. While still allowed, it is optional and always defaults to TCP/IP.

Note that the actual AppServer that the client connects to is controlled by the NameServer based on the Application Service (-AppService) name specified by the client. The Progress interface in co-operation with the NameServer connect the client application to one of the AppServer instances that supports the specified Application Service. If you do not specify an Application Service, the NameServer uses whatever AppServer registers itself as the default service, if any. For more information on load balancing, see the information on NameServers and load balancing in Chapter 2, "Overview of the Progress AppServer," and Chapter 3, "Administration."

If the Application Service is unknown to the NameServer, the client application receives an error. Otherwise, the connection proceeds and any configured Connect procedure executes for the connected AppServer.

For more information on Application Services and NameServers, see Chapter 2, "Overview of the Progress AppServer."

Application Arguments

The *userid*, *password*, and *appserver-info* arguments are optional. However, the requirement to specify these arguments, and to provide specific values for any of them, depends on whether the AppServer Connect procedure is being used on the AppServer you are connecting.

If the AppServer Connect procedure is defined for the AppServer, then you must know how the AppServer Connect procedure is implemented to determine how to specify these values. If you do not specify a value for an argument, the unknown value (?) is passed to the AppServer Connect procedure for that argument. Progress only passes these values from the CONNECT( ) method to the AppServer Connect procedure. It is the procedure that actually evaluates the values that are passed.

If the AppServer Connect procedure is not defined for the AppServer, you do not need to pass the three application arguments to the CONNECT( ) method. If you happen to provide them, they are ignored.

For more information on Connect procedures, see Chapter 4, "Programming the AppServer."

NOTE: When determining what arguments to pass to the CONNECT( ) method, understand that an AppServer accepts a connection request only if any configured Connect procedure executes successfully.

5.3.3 Connection Example

The following code example shows how a client application specifies the connection and application arguments to connect an AppServer. The -AppService, -H, and -S parameters are passed as the *connection-parameters* argument to the CONNECT( ) method. Specific *userid* and *password* values are also passed as connection arguments. A value is not supplied for the *appserver-info* argument.

```
DEFINE VARIABLE hAppSrv AS HANDLE  NO-UNDO.
DEFINE VARIABLE ret     AS LOGICAL NO-UNDO.

CREATE SERVER hAppSrv.
ret = hAppSrv:CONNECT
  ("-AppService inventory -H zeus -S 5162","SMITH","STARSHIP").
IF NOT ret THEN
  RETURN ERROR "Failed to connect to AppServer".

.
.
.

ret = hAppSrv:DISCONNECT().
DELETE OBJECT hAppSrv.
```

This code tries to connect an AppServer that supports the Application Service, `inventory`. It sends its connection request to a NameServer that runs on a machine with the host name, `zeus` and is listening on UDP port 5162. When the CONNECT( ) method executes, the last two arguments are passed as the first two parameters to the AppServer Connect procedure. A value of "SMITH" is passed for *userid* and "STARSHIP" is passed for *password*. The unknown value (?) is passed for the *appserver-info* because a value is not supplied for it.

NOTE: The previous code example, and all remaining code examples in this chapter, are presented in simplified form for discussion purposes. In actual practice, you might use the Parameter File (-pf) parameter as the *connection-parameters* argument and variables to store the *userid* and *password* (probably encrypted).

5.4 Accessing the Connection ID on the Client

On a 4GL client, the connection ID is available as the value of the CLIENT-CONNECTION-ID attribute on the server object handle for the connection. This is a read-only attribute of type CHARACTER. For an Open Client application, Progress provides appropriate methods to access the connection ID (see the *Progress Open Client Developer's Guide*).

For a client, the connection ID of a given AppServer connection remains the same until the client disconnects from the AppServer. If the client reconnects to the same AppServer, the connection ID of the new connection (and thus the value of the CLIENT-CONNECTION-ID attribute for that connection) is different from the connection ID of any previous connection.

For information on how the AppServer generates the value for a connection ID and how to use it, see the information on audit trails and the connection ID in Chapter 6, "Design and Implementation Considerations."

5.5 Running and Managing Remote Procedures

Once a client application connects to an AppServer, the client application can invoke requests to:

- Run remote procedures persistently or non-persistently on the AppServer using the RUN statement.
- Run internal procedures and user-defined functions in remote persistent procedures.
- Run both remote external and internal procedures asynchronously.

  NOTE: A user-defined function cannot run asynchronously.

- Delete remote persistent procedures using the DELETE PROCEDURE or DELETE OBJECT statement.

5.5.1 Running Remote Procedures

A remote procedure, whether it runs persistently or non-persistently, is scoped only to the AppServer session in which it runs. The order of execution for a remote procedure, relative to the client, depends on whether you execute it synchronously or asynchronously.

Synchronous Remote Procedures

If you run a remote procedure synchronously, the client application blocks until the remote procedure request completes and returns to the client. Execution then resumes in the client after the RUN statement that invoked the remote procedure.

Asynchronous Remote Procedures

If you run a remote procedure asynchronously, the client application continues execution immediately after the RUN statement that invokes the remote procedure completes. The remote procedure executes on the AppServer whenever the AppServer is available to execute it. You can then access the results of the remote request in an event procedure that executes in response to a PROCEDURE-COMPLETE event. The client handles the event in the context of a PROCESS EVENTS or other blocking I/O statement, similar to a user-interface event, and executes the event procedure like a user-interface trigger.

RUN Statement Options for Remote Procedures

You can use various options on the RUN statement to execute remote procedures:

- Run a synchronous external procedure using the ON [SERVER] *handle-variable* [TRANSACTION DISTINCT] option:

```
DEFINE VARIABLE hAppSrv AS HANDLE  NO-UNDO.
CREATE SERVER hAppSrv.
hAppSrv:CONNECT(...).
                    .
                    .
                    .
RUN order.p ON SERVER hAppSrv TRANSACTION DISTINCT.
```

  With a small change, this example can execute a remote procedure or a local procedure depending on whether hAppSrv contains a server handle value or the value of the SESSION system handle. Thus, with the same statement, you can run `order.p` remotely (with a server handle) or locally (with the SESSION handle), and determine the choice at run time:

```
DEFINE VARIABLE hAppSrv AS HANDLE  NO-UNDO.
CREATE SERVER hAppSrv.
IF NOT hAppSrv:CONNECT(...) THEN
  hAppSrv = SESSION.
                    .
                    .
                    .
RUN order.p ON SERVER hAppSrv.
```

- Run a remote persistent procedure using the ON SERVER and PERSISTENT SET *handle-variable* options:

```
DEFINE VARIABLE hAppSrv AS HANDLE  NO-UNDO.
DEFINE VARIABLE hOrder  AS HANDLE  NO-UNDO.
CREATE SERVER hAppSrv.
hAppSrv:CONNECT(...).
                    .
                    .
                    .
RUN order.p PERSISTENT SET hOrder
  ON SERVER hAppSrv TRANSACTION DISTINCT
```

A synchronous remote persistent procedure executes similarly to the same persistent procedure run locally. The AppServer session creates the context for the persistent procedure as it starts to execute, and that context persists after it returns until the end of the AppServer connection or until the persistent procedure is explicitly deleted (see the "Deleting Remote Persistent Procedures" section). Once a remote persistent procedure context is instantiated, you can execute any remote internal procedure or user-defined function that is defined within that remote persistent procedure.

The execution context for a remote persistent procedure is managed almost exclusively within the AppServer session where the persistent procedure is instantiated. However, the persistent procedure handle (hOrder in the example), as a proxy procedure handle, provides some additional access to the remote context from the client, allowing the client to delete the persistent procedure remotely. For more information on proxy persistent procedure handles, see the "Understanding Proxy Persistent Procedure Handles" section.

- Run a synchronous internal procedure or user-defined function using the IN *handle-variable* option:

```
DEFINE VARIABLE hAppSrv AS HANDLE  NO-UNDO.
DEFINE VARIABLE hProc AS HANDLE  NO-UNDO.

FUNCTION fnOrders RETURNS LOGICAL IN hProc.

CREATE SERVER hAppSrv.
hAppSrv:CONNECT(...).

RUN order.p PERSISTENT SET hProc
  ON SERVER hAppSrv TRANSACTION DISTINCT.
                    .
                    .
                    .
RUN GetOrders IN hProc.
IF fnOrders THEN DISPLAY "Orders Found!"
```

In this example, both the GetOrders procedure and fnOrders user-defined function are defined in the remote persistent procedure specified by the proxy persistent procedure handle, hProc.

Note that running a synchronous remote internal procedure or invoking a remote user-defined function is syntactically identical to running it locally. Progress automatically determines the right AppServer connection and remote procedure or function to execute from the specified proxy persistent procedure handle.

- Pass INPUT, OUTPUT, and INPUT-OUTPUT variable or TEMP-TABLE parameters, but **not** BUFFER parameters.
- Run a procedure (internal or external) asynchronously using the ASYNCHRONOUS option:

```
DEFINE VARIABLE hAppSrv AS HANDLE  NO-UNDO.
DEFINE VARIABLE hProc AS HANDLE  NO-UNDO.
DEFINE VARIABLE hAsync AS HANDLE NO-UNDO.
DEFINE VARIABLE hAsync2 AS HANDLE NO-UNDO.
DEFINE TEMP-TABLE order-table ...

FUNCTION fnHighVolume RETURNS LOGICAL IN hProc.
CREATE SERVER hAppSrv.
hAppSrv:CONNECT(...).

RUN order.p PERSISTENT SET hProc
  ON SERVER hAppSrv TRANSACTION DISTINCT
  ASYNCHRONOUS SET hAsync.

DO WHILE NOT hAsync:COMPLETE:
  PROCESS EVENTS.
  IF hAsync:COMPLETE THEN
    RUN GetOrders IN hProc ASYNCHRONOUS SET hAsync2
      EVENT-PROCEDURE GetOrderRecords IN THIS-PROCEDURE
      (INPUT-OUTPUT TABLE order-table).
  ELSE /* Order Module Not Ready! */
    /* Work on other stuff while waiting. */
                    .
                    .
                    .
END.

WAIT-FOR CLOSE OF THIS-PROCEDURE.

PROCEDURE GetOrderRecords:
  DEFINE INPUT PARAMETER TABLE FOR order-table.
  IF fnHighVolume THEN DISPLAY "Orders > 100: Call in extra help!".
  MESSAGE "Do you want to see the orders?" UPDATE lSee.
  IF lSee THEN ... /* Display order-table. */
  RETURN.
END.
```

This example asynchronously instantiates a remote persistent procedure, `order.p`, and when available, asynchronously calls the GetOrders remote internal procedure defined in `order.p` to return a TEMP-TABLE parameter with order information. While waiting for the persistent procedure, `order.p`, to become available, the procedure does an unspecified amount of work. This part of the example is procedural, using the PROCESS EVENTS statement to handle the PROCEDURE-COMPLETE event each time through the loop.

The status of the request is checked by testing the COMPLETE attribute of the asynchronous request handle (hAsync).

After the example runs GetOrders asynchronously, the example blocks using the WAIT-FOR statement to handle the PROCEDURE-COMPLETE event for the GetOrders request. When GetOrders completes, the associated event procedure, GetOrderRecords, executes. GetOrderRecords returns the TEMP-TABLE parameter and executes the remote user-defined function fnHighVolume (also defined in `order.p`) to provide warning of heavy order traffic. Note that the remote user-defined function is (and can only be) called synchronously.

Clearly, the handling of asynchronous requests is more complex than for synchronous requests. While this example is partly procedural, you typically handle all asynchronous requests in an event-driven context using a blocking I/O statement, such as the WAIT-FOR statement. For more information on how to manage asynchronous requests, see the "Managing Asynchronous Requests" section.

Remote Procedure Behavior

In addition to the behavior specified by the RUN statement options, a remote procedure:

- Executes in the AppServer session as if it was the first procedure executed in the session (*top level*), using the normal Progress 4GL rules for starting and terminating transactions. For more information about the normal Progress 4GL rules for procedure execution, see the *Progress Programming Handbook*.

  NOTE: Remote procedures can also participate in automatic transactions, which are unique to AppServer sessions. For information, see the sections on transaction management in Chapter 4, "Programming the AppServer."

- Can create other objects in the AppServer session, such as other persistent procedures. These objects are only available locally within the Application Server process in which they are created. They cannot be directly shared between a 4GL client application and the AppServer session.

5.5.2 Understanding Proxy Persistent Procedure Handles

When you execute a remote persistent procedure in a 4GL client application, two persistent procedure handles are created: one within the client application session and another separate handle within the AppServer session where the persistent procedure is created. Progress internally maintains a mapping between the two handles. The handle within the client application is a *proxy persistent procedure handle*, and its PROXY attribute and PERSISTENT attribute are set to TRUE. The corresponding handle within the Application Server process is a *remote persistent procedure handle*, and its REMOTE attribute and PERSISTENT attribute are set to TRUE.

For more information on the relationship between proxy and remote persistent procedure handles, see the information on procedure handles in Chapter 6, "Design and Implementation Considerations."

Accessing Proxy Persistent Procedure Handles

You can obtain access to the proxy persistent procedure handle in the client using the SET option of the RUN statement, which returns the proxy persistent procedure handle value to the HANDLE variable you specify.

You can also access all proxy persistent procedure handles that are currently active for an AppServer connection by returning the value of the FIRST-PROCEDURE attribute or the LAST-PROCEDURE attribute on the server handle for the connection. You can then use the NEXT-SIBLING or PREV-SIBLING attributes on the resulting procedure handle to navigate the list of active proxy persistent procedure handles, for example:

```
DEFINE VARIABLE hProxy AS HANDLE NO-UNDO.
DEFINE VARIABLE hServer AS HANDLE NO-UNDO

/* Create server handle for hServer and connect to an AppServer. */
/* Call remote persistent procedures on the AppServer.           */

hProxy = hServer:FIRST-PROCEDURE. /* First proxy handle */

/* Display the pathname of the file that contains the remote procedure */
/* active on the AppServer for each proxy persistent procedure handle. */

DO WHILE NOT hProxy = hServer:LAST-PROCEDURE:
   DISPLAY hProxy:FILE-NAME.
   hProxy = hProxy:NEXT-SIBLING.
END.
DISPLAY hProxy:FILE-NAME. /* Procedure pathname for last proxy handle */
```

Comparing Proxy and Local Persistent Procedure Handles

Unlike the values of a procedure handle variable and the THIS-PROCEDURE system handle that reference the same local persistent procedure context, the proxy persistent procedure handle in a client session and the corresponding remote persistent procedure handle in the Application Server process are truly separate handles. For example, setting the PRIVATE-DATA attribute on a proxy persistent procedure handle has no effect on the PRIVATE-DATA attribute of the corresponding remote persistent procedure handle.

However, note that the Progress mapping between a proxy persistent procedure handle and its corresponding remote persistent procedure handle allows you to execute remote internal procedures and user-defined functions using the proxy handle. For more information, see the "Running and Managing Remote Procedures" section.

5.5.3 Deleting Remote Persistent Procedures

As with local persistent procedures, a remote persistent procedure context remains active within an AppServer session until it is deleted using the DELETE OBJECT or DELETE PROCEDURE statement. All remote persistent procedures are also deleted when the client application disconnects from the AppServer where the procedure is active (see the "Disconnecting from an AppServer Instance" section). You can thus delete a remote persistent procedure from a 4GL client session by:

- Deleting its proxy procedure handle using the DELETE OBJECT or DELETE PROCEDURE statement.
- Disconnecting the AppServer by using the DISCONNECT( ) method on the corresponding server handle. This deletes all remote persistent procedures in the AppServer session.

If the delete occurs in the context of another remote procedure request to the AppServer, the deletion is pending until the request completes and returns to the client. When the remote persistent procedure is finally deleted, both its proxy procedure handle on the client and its remote procedure handle in the Application Server process are deleted together.

5.5.4 Remote Procedure Code Examples

This section presents Progress AppServer-related code examples. The first example presents the code for a non-persistent procedure. The remaining two examples show two different ways to write code for the same remote persistent procedure.

Example 1: Remote Non-Persistent Procedure Example

This example shows how to run a remote, non-persistent procedure, `checkstk.p`, on an AppServer identified as hAppSrv. The TRANSACTION DISTINCT option is used on the RUN statement to specify that the client application's transaction is not propagated to hAppSrv. The two temporary table parameters that are defined, ttOrderLine and ttItem, are passed as INPUT and OUTPUT parameters, respectively, to the `checkstk.p` procedure.

```
DEFINE VARIABLE hAppSrv AS HANDLE  NO-UNDO.
DEFINE VARIABLE ret     AS LOGICAL NO-UNDO.

DEFINE TEMP-TABLE ttOrderLine LIKE Order-Line.
DEFINE TEMP-TABLE ttItem      LIKE Item.

CREATE SERVER hAppSrv.
ret = hAppSrv:CONNECT("-AppService inventory -S 5162 -H zeus",
                      "SMITH", "STARSHIP").
IF NOT ret THEN
  RETURN ERROR "Failed to connect to AppServer".
IF ERROR-STATUS:ERROR THEN
  RETURN ERROR RETURN-VALUE.

.
.

RUN checkstk.p ON hAppSrv TRANSACTION DISTINCT
  (INPUT Customer.Cust-Num,
   INPUT TABLE ttOrderLine,
   OUTPUT TABLE ttItem) NO-ERROR.
IF ERROR-STATUS:ERROR THEN DO:
  ret = hAppSrv:DISCONNECT().
  RETURN ERROR RETURN-VALUE.
END.

.
.

ret = hAppSrv:DISCONNECT().
DELETE OBJECT hAppSrv.
```

When the RUN statement is executed, a remote procedure request to run `checkstk.p` is sent to the AppServer instance that is connected via the hAppSrv server handle. Once the RUN statement is completed, this code checks to see if `checkstk.p` completed with an ERROR condition as would occur by executing the RETURN ERROR statement. If `checkstk.p` did execute the RETURN ERROR statement, then this code also returns with an ERROR condition. By using the RETURN-VALUE function, the code also returns the value returned by `checkstk.p`.

Example 2: Remote Persistent Procedure Example

This code example shows how to instantiate a remote persistent procedure, `order.p`, on an AppServer.

```
DEFINE VARIABLE hAppSrv AS HANDLE  NO-UNDO.
DEFINE VARIABLE hOrder  AS HANDLE  NO-UNDO.
DEFINE VARIABLE iOrderNum AS LIKE Order.Order-Num NO-UNDO.
DEFINE VARIABLE ret     AS LOGICAL NO-UNDO.

DEFINE TEMP-TABLE ttOrder     LIKE Order.
DEFINE TEMP-TABLE ttOrderLine LIKE Order-Line.

CREATE SERVER hAppSrv.
ret = hAppSrv:CONNECT("-AppService inventory -S 5162 -H zeus",
                      "SMITH", "STARSHIP").
IF NOT ret THEN
  RETURN ERROR "Failed to connect to AppServer".
IF ERROR-STATUS:ERROR THEN
  RETURN ERROR RETURN-VALUE.

.

.
RUN order.p ON hAppSrv TRANSACTION DISTINCT
  PERSISTENT SET hOrder.

RUN GetExistingOrder IN hOrder
  (INPUT iOrderNum,
   OUTPUT TABLE ttOrder,
   OUTPUT TABLE ttOrderLine) NO-ERROR.
IF ERROR-STATUS:ERROR THEN DO:
  ret = hAppSrv:DISCONNECT().
  RETURN ERROR RETURN-VALUE.
END.

.
.

ret = hAppSrv:DISCONNECT().
DELETE OBJECT hAppSrv.
```

When the RUN statement for `order.p` completes, a reference to the persistent procedure context is saved in handle hOrder. The SERVER attribute for hOrder contains a reference to the server handle hAppSrv. When the internal procedure GetExistingOrder is run for persistent procedure hOrder, Progress uses the value of hAppSrv to determine the AppServer instance where the persistent procedure is located.

Example 3: Remote Persistent Procedure Example Using the FUNCTION Statement

This example shows how a client application invokes a user-defined function, GetExistingOrder, that is implemented remotely on an AppServer. It is presented as an alternative way to code example 2. The FUNCTION GetFirstOrder statement in this example replaces the internal procedure GetExistingOrder referenced in example 2.

```
DEFINE VARIABLE hAppSrv AS HANDLE  NO-UNDO.
DEFINE VARIABLE hOrder  AS HANDLE  NO-UNDO.
DEFINE VARIABLE iCustNum AS LIKE Order.Order-Num NO-UNDO.
DEFINE VARIABLE ret     AS LOGICAL NO-UNDO.

DEFINE TEMP-TABLE ttOrder     LIKE Order.
DEFINE TEMP-TABLE ttOrderLine LIKE Order-Line.
/* user-defined function GetFirstOrder */
FUNCTION GetFirtOrder RETURNS LOGICAL
  (INPUT inpcustnum AS INTEGER,
   OUTPUT TABLE ttorder,
   OUTPUT TABLE ttorderLine) in hOrder.


CREATE SERVER hAppSrv.
ret = hAppSrv:CONNECT("-AppService inventory -S 5162 -H zeus",
                      "SMITH", "STARSHIP").
IF NOT ret THEN
  RETURN ERROR "Failed to connect to AppServer".
IF ERROR-STATUS:ERROR THEN
  RETURN ERROR RETURN-VALUE.

.
.

RUN order.p ON hAppSrv TRANSACTION DISTINCT
  PERSISTENT SET hOrder.
IF NOT GetFirstOrder (INPUT iCustNum,OUTPUT TABLE ttorder, OUTPUT TABLE
ttorderline) THEN DO:
  ret = hAppSrv:DISCONNECT().
  RETURN ERROR.
END.
.
.

ret = hAppSrv:DISCONNECT().
DELETE OBJECT hAppSrv.
```

Like example 2, when the RUN statement for `order.p` completes, a reference to the persistent procedure context is saved in handle hOrder, and the SERVER attribute for hOrder contains a reference to the server handle hAppSrv. However, instead of running an internal procedure, it runs the user-defined function, GetFirstOrder.

For more information about creating user-defined functions, see the FUNCTION Statement section in the *Progress Language Reference*.

NOTE: If a client application exits without executing the DISCONNECT( ) method on the appropriate handles, Progress automatically disconnects the client from the AppServer. This automatic disconnection can also occur if the AppServer machine looses network connectivity with the client application. This can happen, for example, if the is a network failure.

5.6 Disconnecting from an AppServer Instance

You can disconnect an AppServer instance by executing the DISCONNECT( ) method on the server object handle for the connection.

5.6.1 Using the DISCONNECT( ) Method

Using the DISCONNECT( ) method terminates the connection with the specified AppServer by sending a disconnect request to the connected AppServer. When the AppServer receives the disconnect request:

- **For a state-reset or state-aware AppServer**—Control returns immediately to the client application, and any configured Disconnect procedure executes in the connected Application Server process.
- **For a stateless AppServer—**Any configured Disconnect procedure executes in an available Application Server process, then control returns to the client application.
- **For any AppServer with pending asynchronous requests**—All running or pending asynchronous requests are cancelled and the corresponding event procedure is called for each request. The CANCELLED attribute on the asynchronous request handle for all such cancelled requests is set to TRUE. For more information on asynchronous requests, see the "Managing Asynchronous Requests" section.

When the disconnection request completes, you can send no more remote requests using the disconnected server handle.

The following example shows how a client application can disconnect from an AppServer instance:

```
DEFINE VARIABLE hAppSrv AS HANDLE  NO-UNDO.
DEFINE VARIABLE ret     AS LOGICAL NO-UNDO.

CREATE SERVER hAppSrv.
ret = hAppSrv:CONNECT("-AppService inventory -S 5162 -H zeus",
                      "SMITH", "STARSHIP").
IF NOT ret THEN
  RETURN ERROR "Failed to connect to AppServer".
IF ERROR-STATUS:ERROR THEN DO:
  ret = hAppSrv:DISCONNECT().
  RETURN ERROR RETURN-VALUE.
END.
.
.
.

ret = hAppSrv:DISCONNECT().
DELETE OBJECT hAppSrv.
```

When the DISCONNECT( ) method executes, the client application disconnects from the AppServer referenced by the hAppSrv handle.

5.6.2 Deleting the Server Handle

To delete a server object handle, you must first disconnect from the AppServer. Once you have disconnected an AppServer, you can release the associated server object handle by executing the DELETE OBJECT statement, as shown in the previous code example.

NOTE: If you attempt to delete the server handle before disconnecting the AppServer, Progress returns an error to the client.

5.7 Handling Conditions and Return Values

Typically, you pass context across the process barrier between client and AppServer sessions using parameters on remote procedures. Mechanisms for handling conditions and return values provide another way to exchange context between the two process spaces.

5.7.1 Raising the STOP Condition

A 4GL client can raise a STOP condition within a remote procedure when a user presses the **STOP** key. Raising the STOP condition while a remote procedure is executing causes the STOP condition to be propagated to the appropriate AppServer session where a STOP condition is then raised.

A STOP condition sent to the remote procedure does not necessarily mean that the remote procedure terminates immediately. ON STOP handlers within the remote procedure might prevent the procedure from actually being terminated. For information on how a remote procedure handles a STOP condition received from the client, see the information on condition handling in Chapter 4, "Programming the AppServer."

5.7.2 Handling Conditions on the RUN Statement

When you execute a synchronous remote procedure, all of the following conditions are handled by Progress exactly as if they were raised within a local procedure call:

- Any condition raised during execution of the synchronous RUN statement that invokes the remote procedure
- Any ERROR condition raised by executing the RETURN ERROR statement in the remote procedure itself

If you use the NO-ERROR option on the RUN statement, you can obtain any information about errors that result from executing the RUN statement using the ERROR-STATUS system handle.

A remote procedure executed persistently returns as a non-persistent procedure when the remote procedure does one of the following:

- Completes with an unhandled STOP or QUIT condition.
- Executes a RETURN ERROR statement.

For an asynchronous remote procedure, conditions that occur on the RUN statement itself are handled exactly as for a synchronous remote procedure. However, unhandled conditions that occur during execution of the asynchronous remote procedure are propagate back to the client in the context of the event procedure. The result can be referenced within the event procedure using the RETURN-VALUE function, the ERROR-STATUS system handle, and the associated asynchronous request handle (SELF). Any conditions that result from problems with the event procedure itself are handled in the context of the PROCESS EVENTS or blocking I/O statement that executes the event procedure. For more information on handling conditions for asynchronous remote procedures, see the "Handling the Response from an Asynchronous Request" section.

5.7.3 Using the RETURN-VALUE Function Across Sessions

The value of the RETURN-VALUE function in a 4GL client session is affected by remote procedure execution. If a remote procedure returns a value by executing the RETURN statement, this value is returned to the client as the value of the RETURN-VALUE function.

5.8 Managing Asynchronous Requests

Asynchronous requests allow a 4GL client to execute a remote procedure and continue execution before the AppServer has completed (or even begun) executing the remote request. This mechanism enhances client performance, especially in an event-driven application where the user can continue interacting with the client application while the AppServer processes the remote request.

In the following sections, when executing an asynchronous request, Progress places the request on a send queue for the connection until the AppServer can execute the specified remote procedure. Similarly, when the procedure completes execution, Progress places the results on a response queue, where the results wait until the client handles the PROCEDURE-COMPLETE event associated with the request. As a result, all asynchronous requests on a connection are guaranteed to be executed, with results returned, in the order you submit them. For more information on the send and response queues, see Chapter 2, "Overview of the Progress AppServer."

5.8.1 Executing an Asynchronous Request

You execute an asynchronous request using the RUN statement with the ASYNCHRONOUS option (the RUN...ASYNCHRONOUS statement). With this option, you can specify a variable to return an asynchronous request handle used to uniquely identify and maintain the status of the asynchronous request. You can also specify an internal procedure to use as the event procedure to handle the response from the asynchronous request. For more information on the RUN...ASYNCHRONOUS statement, see the RUN statement entry in the *Progress Language Reference*.

When specifying the RUN...ASYNCHRONOUS statement, you can also include INPUT, OUTPUT, and INPUT-OUTPUT parameters similar to the parameters on a synchronous RUN statement. As with the synchronous RUN statement, the input parameters represent values passed to the remote procedure. However, unlike the synchronous RUN statement, the output parameters do not return results from the remote procedure directly on the RUN...ASYNCHRONOUS statement. The values for these output parameters are not available until the asynchronous request completes and the client handles the associated PROCEDURE-COMPLETE event. On detecting this event, the client executes the specified event procedure, which returns the values as input parameters to the event procedure. Thus, the output parameters on the RUN...ASYNCHRONOUS statement serve only as placeholders to map the signature of the remote procedure on the AppServer. For more information, see the "Handling the Response from an Asynchronous Request" section.

Asynchronous Request Handles

Associated with each asynchronous request is an *asynchronous request handle*. An instance of this handle is created each time you execute the RUN...ASYNCHRONOUS statement. Thus, you can use this handle to obtain information on a specific asynchronous request.

The 4GL client allows you to access this handle in three ways:

- Using the SET *async-request-handle* option on the RUN...ASYNCHRONOUS statement
- Using the FIRST-ASYNC-REQUEST and LAST-ASYNC-REQUEST attributes of the server handle or the NEXT-SIBLING and PREV-SIBLING attributes of the asynchronous request handle to walk through the valid asynchronous request handles for the specified AppServer
- Accessing the SELF system handle within an event procedure in response to the asynchronous request

For more information on using the asynchronous request handle, see "Handling the Response from an Asynchronous Request."

5.8.2 Handling the Response from an Asynchronous Request

When an asynchronous request completes execution, it sends a response to the AppServer client, which places it on the response queue for the appropriate server handle. To signify that the response is ready to be processed, the PROCEDURE-COMPLETE event is placed on the event queue where it can be processed in the context of blocking I/O statements, such as WAIT-FOR or the PROCESS EVENTS statement.

PROCEDURE-COMPLETE Events

The PROCEDURE-COMPLETE event is an event on the asynchronous request handle. It indicates that the asynchronous request associated with the handle has completed execution, and that the corresponding event procedure can be executed. Progress executes the event procedure in the context of a blocking I/O statement much like it executes the trigger for a user-interface event.

Note that you do not have to specify the PROCEDURE-COMPLETE event explicitly in your client code. Progress processes the event like any other 4GL event. However, you do not have to define a trigger for the event (for example, using an ON statement) because you specify an event procedure in the RUN...ASYNCHRONOUS statement that defines the “trigger code” for the event.

To process a PROCEDURE-COMPLETE event for a particular asynchronous request handle, Progress:

- Decrements the ASYNC-REQUEST-COUNT attribute on the following handles:
  - The server handle contained in the SERVER attribute of the asynchronous request handle
  - The procedure handle contained in the PERSISTENT-PROCEDURE attribute of the asynchronous request handle, if the PERSISTENT-PROCEDURE attribute refers to a valid persistent procedure
- Sets the COMPLETE attribute for the asynchronous request handle to TRUE.
- Sets the STOP, QUIT, or ERROR attribute for the asynchronous request handle appropriately as indicated by the response message from the AppServer, and stores any error information returned from the AppServer for the request in the ERROR-STATUS system handle.
- Sets the return value for the RETURN-VALUE function, if a return value was returned by the AppServer.

- Attempts to execute the event procedure specified by the EVENT-PROCEDURE and the EVENT-PROCEDURE-CONTEXT attributes for the asynchronous request handle, if EVENT-PROCEDURE is not the empty string (""), and:
  - Sets each INPUT parameter for the event procedure to the unknown value (?) or, if the parameter is a TEMP-TABLE, the TEMP-TABLE remains unchanged, if the response message indicates that the remote request finished with a STOP, ERROR, or QUIT condition.
  - Sets the INPUT parameter values for the event procedure to the OUTPUT and INPUT-OUTPUT parameter values returned by the remote procedure, if the response message indicates that the remote request completed successfully.
  - Displays an error message, if a specified event procedure fails to execute for any reason.

Event Procedures

To handle the response from an asynchronous request, you must define an event procedure. In this context, an *event procedure* is an internal procedure that executes in response to a PROCEDURE-COMPLETE event. To define an event procedure you must:

- Specify the name of the event procedure using the EVENT-PROCEDURE option on the RUN...ASYNCHRONOUS statement that executes the request.
- Define an internal procedure with the same name as the event procedure specified in the RUN...ASYNCHRONOUS statement.

Typical places where you might define the specified internal procedure include:

- The same procedure context as the RUN...ASYNCHRONOUS statement that specifies the event procedure.
- An external procedure context that you specify using the EVENT-PROCEDURE option on the RUN...ASYNCHRONOUS statement using its procedure handle. In this case, the external procedure context must be active in the client session before you execute the RUN...ASYNCHRONOUS statement.

You must define the specified internal procedure with INPUT parameters that correspond in type and order to the OUTPUT and INPUT-OUTPUT parameters that you specify on the RUN...ASYNCHRONOUS statement.

To receive results from the asynchronous request in an event procedure, you can access:

- The asynchronous request handle for the request using the SELF system handle

- The attributes and methods of the ERROR-STATUS system handle to obtain error information from the request
- Values passed as output parameters from the remote procedure to corresponding input parameters in the event procedure
- The RETURN-VALUE function to get the return value from the request

Obtaining Error Information

With an asynchronous remote procedure request, the RUN...ASYNCHRONOUS statement returns once the client request has been validated. Any errors that occur while processing the RUN...ASYNCHRONOUS statement are returned as a STOP or ERROR condition in the context of the RUN...ASYNCHRONOUS statement. Information on errors that occur once the asynchronous request has been submitted can be obtained by using the ERROR-STATUS system handle within the event procedure for the asynchronous request.

If the asynchronous request completes with a condition, you can use attributes on the asynchronous request handle to determine the condition that the request completed with, as follows:

- ERROR attribute indicates if the request completed with an ERROR condition
- STOP attribute indicates if the request completed with a STOP condition
- QUIT attribute indicates if the request completed with a QUIT condition.

Note that any asynchronous request that raises the STOP condition (by executing the STOP statement), and does not handle the condition itself, also sets the STOP attribute of its corresponding asynchronous request handle to TRUE.

If any errors occur when the event procedure attempts to execute, Progress displays an error message in the context of the blocking I/O statement, and no information is stored in the asynchronous request handle. Errors where this might occur include, for example, an event procedure whose INPUT parameters do not match in order and type with the output parameters from the remote request.

Obtaining Parameter Values

As explained earlier, the values returned for OUTPUT or INPUT-OUTPUT parameters on an asynchronous remote procedure are not returned to the RUN statement as for a synchronous remote procedure. Instead, they are returned in the corresponding INPUT parameters of the specified event procedure.

Note that INPUT parameters in an event procedure observe the same rules of definition and scoping as any internal procedure:

```
DEFINE TEMP-TABLE order-table ...
DEFINE VARIABLE icount AS INTEGER INITIAL 0.
DEFINE VARIABLE hAsync AS HANDLE.
DEFINE BUTTON bCheckOrders. ...

CREATE SERVER hAppSrv.
hAppSrv:CONNECT().

RUN order.p
  ON SERVER hAppSrv ASYNCHRONOUS SET hAsync
  EVENT-PROCEDURE GetOrderRecords IN THIS-PROCEDURE
  (INPUT-OUTPUT TABLE order-table, OUTPUT icount).
DISPLAY icount. /* Displays 0 */
ON CHOOSE OF bCheckOrders
  IF icount > 0
  THEN MESSAGE "You have" icount "orders." VIEW-AS ALERT-BOX
  ELSE MESSAGE "Orders are pending ..." VIEW-AS ALERT-BOX.
                    .
                    .
                    .
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.

PROCEDURE GetOrderRecords:
  DEFINE INPUT PARAMETER TABLE FOR order-table.
  DEFINE INPUT PARAMETER order-count AS INTEGER.
  IF SELF:ERROR OR SELF:STOP THEN ... /* Handle condition */
  ELSE icount = order-count.
  RETURN.
END.
```

Thus, in this example, the order-table TEMP-TABLE (defined in the outer block) receives its content directly from the order-table INPUT parameter of the GetOrderRecords event procedure. However, you must explicitly assign the icount variable (defined in the outer block) to the order-count parameter in order to make the parameter value available to the outer block. (That is, if you define the INPUT parameter as icount instead of order-count, Progress does not automatically make the parameter value available as icount in the outer block. Only the TABLE parameter is scoped to the outer block and thus receives its value from the parameter.)

Note also that the event procedure checks for and handles any unhandled conditions generated from the execution of `order.p` using AppServer error-handling mechanisms.

Obtaining the Return Value

You can obtain the return value for a remote procedure that you call asynchronously by invoking the RETURN-VALUE function within the event procedure that handles the asynchronous request. This function thus returns the value specified by a corresponding RETURN statement executed in the remote procedure. As for a synchronous request, if the remote procedure returns no value in the RETURN statement, the RETURN-VALUE function returns the empty string ("").

5.8.3 Cancelling Asynchronous Requests

You can cancel all asynchronous requests that are executing or pending on behalf of the client on a particular AppServer by executing the CANCEL-REQUESTS( ) method on the server object handle. This method raises the STOP condition in the context of the asynchronous procedure currently executing and causes results to be returned for any pending asynchronous requests, as described in the following paragraph.

The associated event procedures execute for all cancelled requests the next time the client blocks for I/O or executes the PROCESS EVENTS statement. Each event procedure receives the following results, depending on the final state of its corresponding asynchronous request:

- **Any request that is completed at the time of cancellation but whose event procedure has not yet run**—The event procedure receives all input parameters passed from the request, and the COMPLETE attribute on the asynchronous request handle (SELF) is set to TRUE. This is the same result as a normal asynchronous request completion.

- **The request that is executing and that is stopped in response to the CANCEL-REQUESTS( ) method**—The event procedure input parameters are set to the corresponding values that were set in the remote procedure when execution stopped, and the COMPLETE attribute on the asynchronous request handle (SELF) is set to TRUE. If the stopped request does not handle the STOP condition raised by the CANCEL-REQUESTS( ) method, the STOP attribute on the asynchronous request handle (SELF) is also set to TRUE. This is the same result as if the STOP condition were raised on the Application Server running the request.

- **Any request that is removed from the send queue prior to its execution**—The event procedure input parameters are set to the unknown value (?) (or unchanged for TEMP-TABLE parameters), and the CANCELLED attribute on the asynchronous request handle (SELF) is set to TRUE. This result can only occur for a request that is cancelled while waiting for execution.

NOTE: If you disconnect an AppServer using the DISCONNECT( ) method, this also cancels all asynchronous requests still running or pending on the AppServer. For more information, see the "Disconnecting from an AppServer Instance" section.

5.8.4 Asynchronous Requests and Persistent Procedures

Client applications can asynchronously instantiate a persistent procedure and then execute an internal procedure on the persistent procedure before the remote persistent procedure has actually been created on the AppServer. When the RUN...ASYNCHRONOUS statement is executed for the purposes of creating the remote persistent procedure, a proxy persistent procedure handle is returned to the client application.

The client application can then immediately use the proxy persistent procedure handle to run an internal procedure in the remote persistent procedure. If the asynchronous request to instantiate the remote persistent procedure has completed, you can run the internal procedure synchronously or asynchronously. If it has not completed, you can only run the internal procedure asynchronously.

Because both the request to instantiate the persistent procedure and the asynchronous request to execute the internal procedure are executing on the same connection, Progress guarantees that the request to instantiate the persistent procedure executes before the request to run the internal procedure. If the request to instantiate the persistent procedure ultimately fails, the request to run the internal procedure will ultimately fail as well, because the proxy persistent procedure handle used to run the internal procedure is invalid. When the internal procedure request fails, Progress sets the ERROR attribute to TRUE on both the asynchronous request handle and the ERROR-STATUS system handle, indicating that an invalid proxy persistent procedure handle was used to execute the internal procedure.

5.8.5 Deleting Asynchronous Remote Persistent Procedures

You can delete a remote persistent procedure that you have executed asynchronously just like one you have executed synchronously, using the DELETE OBJECT or DELETE PROCEDURE statement. However, for an asynchronous procedure, the delete request raises an ERROR condition if the ASYNC-REQUEST-COUNT attribute on the proxy persistent procedure handle is greater than zero (0). If ASYNC-REQUEST-COUNT is zero (0), the local proxy procedure handle is deleted immediately.

If an asynchronous request for another procedure is executing in the Application Server process at the time of the delete request, the delete request is queued on the AppServer until the Application Server process becomes available to delete the remote persistent procedure.

5.8.6 Deleting Server Handles

You can delete a server handle using the DELETE OBJECT statement. However, the delete request raises an ERROR condition if the ASYNC-REQUEST-COUNT attribute on the server handle is not zero (0).

5.8.7 Deleting Asynchronous Request Handles

You can delete an asynchronous request handle using the DELETE OBJECT statement. This frees all resources (memory) used by this handle. The recommended technique is to delete the handle at the end of the event procedure using the SELF handle (that is, DELETE OBJECT SELF).

5.8.8 Debugging Asynchronous Requests

The AppServer supports two models for debugging 4GL code, the remote and distributed debugging models. An asynchronous request can only be debugged using the remote debugging model. The distributed debugging model can only be used for debugging synchronous requests. For more information on the AppServer debugging models, see Chapter 7, “Debugging.”

5.8.9 Mixing Synchronous and Asynchronous Requests

If you attempt to execute a synchronous request on an AppServer connection with outstanding asynchronous requests, Progress raises the ERROR condition. You can determine if there are any outstanding asynchronous requests for an AppServer by checking the value of the ASYNC-REQUEST-COUNT attribute of the appropriate server handle.

You can also ensure that all asynchronous requests for a given server handle have completed execution by the way you set up your event management in the client. Specify the PROCEDURE-COMPLETE event in the WAIT-FOR statement for the asynchronous request handle that corresponds to the last asynchronous request submitted before the WAIT-FOR statement is executed. For example, assume that you submit three asynchronous requests for some server handle and that the asynchronous request handles are returned in HANDLE variables hdl1, hdl2, and hdl3. To ensure that there are no outstanding asynchronous requests before submitting a synchronous request, execute the WAIT-FOR statement specifying the PROCEDURE-COMPLETE event for hdl3.

5.8.10 Running Asynchronous Requests on the SESSION System Handle

You can execute a RUN...ASYNCHRONOUS statement using a server object handle or a SESSION system handle. If you use a SESSION handle, the request is executed synchronously on the client as if it was called using a synchronous RUN statement. However, unlike a standard synchronous procedure call that returns the results directly to the RUN statement, the results are returned by the event procedure specified on the RUN...ASYNCHRONOUS statement as though it were responding to a PROCEDURE-COMPLETE event from an AppServer.

Thus, using the SESSION handle, the event procedure runs synchronously as part of the RUN statement and returns execution to the statement immediately following the same RUN statement. This is different from using a server handle, where the event procedure runs only in the context of a blocking I/O or PROCESS EVENTS statement that processes the PROCEDURE-COMPLETE event. However, this still allows you to use the RUN...ASYNCHRONOUS statement to program local and remote requests in a similar way, for example, when an AppServer is not available.

NOTE: In addition to order of execution, using the SESSION handle to run an asynchronous request as a local synchronous request results in error handling that is slightly different from using a server handle. For this reason, you might want to develop using a local AppServer when a network AppServer is not available. For more information on running a local AppServer, see Appendix A, "Running the AppServer Without a Network." For more information on error handling when using the SESSION handle, see the RUN statement entry in the *Progress Language Reference*.

5.8.11 Examples

The following examples show simple implementations of various client and remote procedures. The first two examples (Figure 5–1 and Figure 5–2) compare two similar implementations, one synchronous and the other asynchronous. These are non-executing examples, shortened for illustration. The third example (`client.p` and `server.p`) shows a fully executing, asynchronous implementation of client and remote procedures.

Synchronous Request Execution Model

Figure 5–1 shows a remote procedure executed synchronously. The application sets a simple status flag (Done) to indicate that the remote request has completed.

As an event-driven GUI application, the client first executes a WAIT-FOR statement after having set up code for handing events (1). When the client executes `sync.p` as a *synchronous remote procedure* (2), the client blocks while the AppServer completes the request (3) and only continues execution after the remote procedure completes (4). Results from a synchronous remote procedure call are returned to the client immediately after the point of invocation (4), exactly as they are returned from a local procedure call.

Note that in this example all of the remote processing and handling of the result occurs in the trigger for bStatus. While straight forward to implement, the client can block for an indeterminate period of time at the synchronous RUN statement waiting for `sync.p` to complete, depending on the application. Asynchronous requests allow a client to avoid this bottleneck.

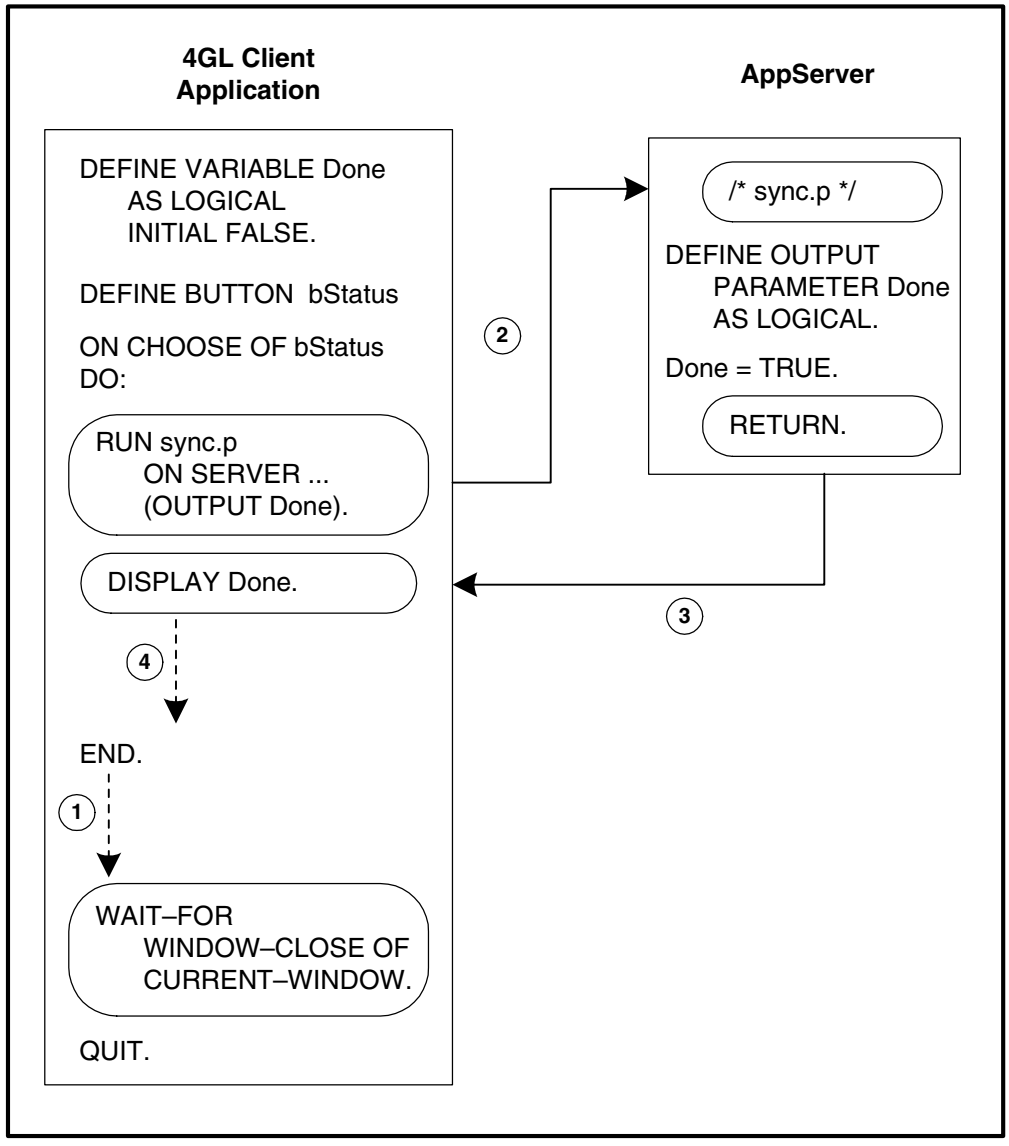


Figure 5–1: Synchronous Requests

Asynchronous Request Execution Model

Figure 5–2 shows a remote procedure executed asynchronously. As in Figure 5–1, the client is an event-driven GUI and the application sets a simple status flag (Done) to indicate that the remote request has completed.

In this implementation, the client first executes `async.p` as an *asynchronous remote procedure* (1, specified by the ASYNCHRONOUS keyword on the RUN statement). The client immediately continues execution, until it reaches the WAIT-FOR statement to get events (2) **at the same time** that the AppServer executes the remote request. Thus, at this point (1 and 2), the client and remote request are running in parallel. The client can continue to execute, calling additional asynchronous remote procedures on the same or different AppServer connections.

As each asynchronous request completes (like `async.p` at 3), the client handles the results in a specified *event procedure* (4). This event procedure, specified in the asynchronous RUN statement, is essentially a "trigger" that executes on the client in response to a PROCEDURE-COMPLETE event posted to the client when the associated asynchronous request completes.

As with user-interface events, the client can handle PROCEDURE-COMPLETE events in the context of a blocking I/O statement, such as the WAIT-FOR statement (4), or by executing the PROCESS EVENTS statement. Progress maps the PROCEDURE-COMPLETE event for each asynchronous request to the appropriate event procedure using a unique *asynchronous request handle* that is generated for each request (not shown). This handle provides the mechanism that you can use to monitor the status of each request.

Note that the example in Figure 5–2 provides the same functionality as Figure 5–1. In fact, `sync.p` and `async.p` are identical, except for their names, which are changed for readability. The AppServer has no idea whether its procedures are being called synchronously or asynchronously. The type of request is entirely a function of the client and its implementation.

The main difference, is that the bStatus trigger in Figure 5–2 tests the status of the remote request and performs actions based on whether the request finished, rather than invoking the request itself, as in Figure 5–1. Thus, the synchronous bottleneck is removed. In all such cases, the synchronous bottleneck is avoided by handling the results asynchronously.

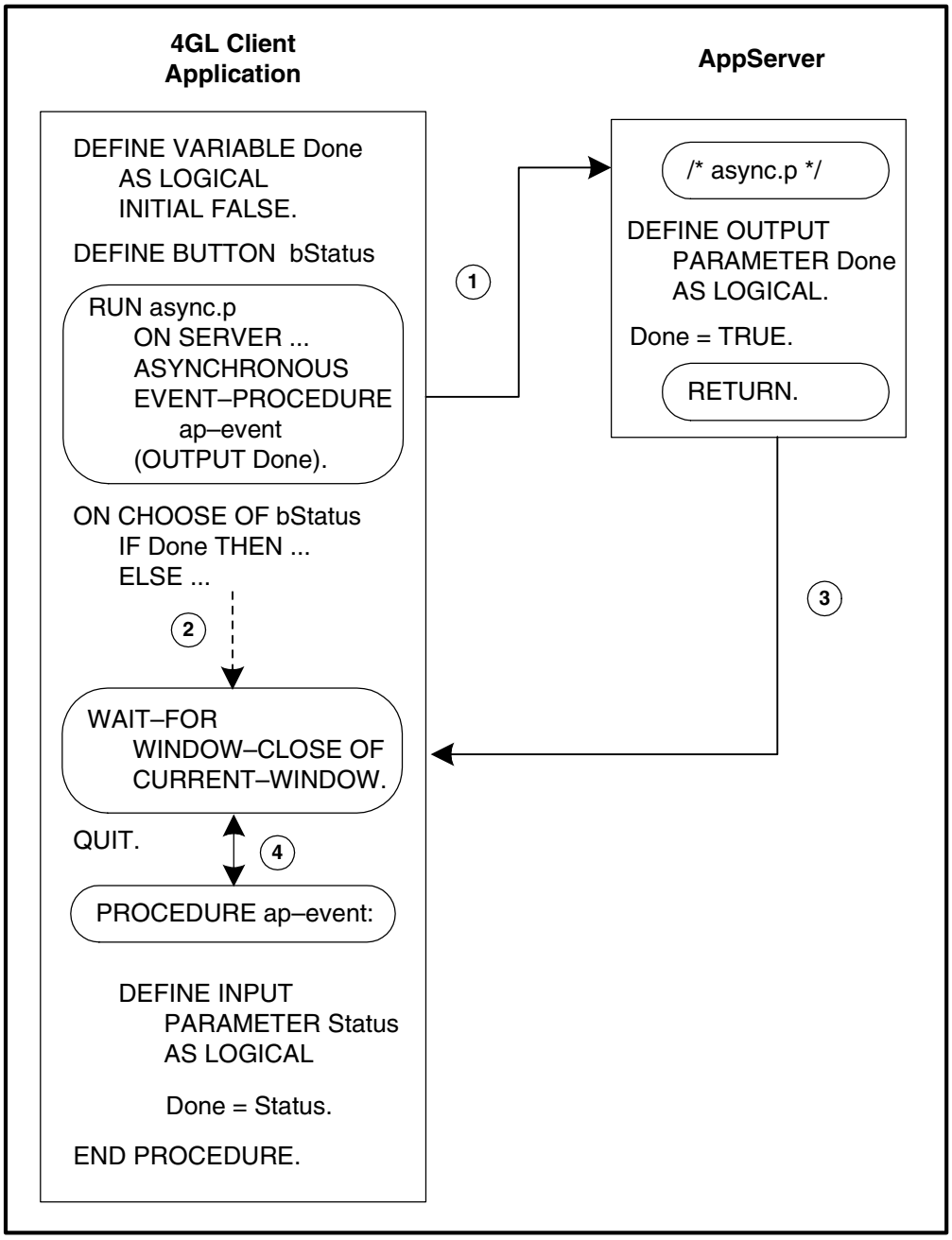


Figure 5–2: Asynchronous Requests

A Complete Asynchronous Request Example

In the following example, the 4GL client procedure, `client.p`, calls `server.p` (following) asynchronously on an AppServer. The `server.p` procedure returns the customer number corresponding to a customer name provided by `client.p`.

The `client.p` procedure blocks and waits for the PROCEDURE-COMPLETE event that triggers execution of the GetCustNum event procedure. The GetCustNum procedure displays the result and deletes the corresponding asynchronous request handle (referenced by the SELF system handle).

In a more typical example, additional event handlers (triggers) would be active for user-interface events, allowing the user of `client.p` to perform other tasks with the procedure while completion of the request is pending. In this case, the client application blocks immediately after submitting the request to wait for the result and terminate on the PROCEDURE-COMPLETE event:

client.p

```
DEFINE VARIABLE async-request AS HANDLE.
DEFINE VARIABLE s-hdl AS HANDLE.
DEFINE VARIABLE customer-name as CHARACTER INITIAL("WILNER").
DEFINE VARIABLE ret AS LOGICAL.

CREATE SERVER s-hdl.

ret = CONNECT("-S 5162  -H zeus -AppService Inventory").

RUN server.p ON s-hdl
   ASYNCHRONOUS SET async-request
   EVENT-PROCEDURE "GetCustNum"
   (INPUT customer-name, OUTPUT customer-number AS INTEGER).

WAIT-FOR PROCEDURE-COMPLETE OF async-request.


PROCEDURE GetCustNum:
   DEFINE INPUT PARAMETER customer-number AS INTEGER.

   DISPLAY customer-number.
   DELETE OBJECT SELF.

END.
```

server.p

```
DEFINE INPUT PARAMETER customer-name AS CHARACTER.
DEFINE OUTPUT PARAMETER customer-number AS INTEGER.

FIND customer WHERE name = customer-name.
customer-number = cust-num.

END.
```

6

Design and Implementation Considerations

This chapter highlights specific performance, transaction management, deployment, security internationalization issues related to planning and implementing a distributed application with Progress AppServer. It explains some programming techniques, especially those commonly used by both 4GL client and AppServer procedures, to help you write distributed applications.

The topics include:

- Primary performance considerations
- Secondary performance considerations
- Transaction and record management considerations
- Deployment considerations
- Security considerations
- Multi-language support
- Distributed application design and implementation

This chapter assumes that you are familiar with all previous chapters, especially Chapter 2, "Overview of the Progress AppServer," and Chapter 4, "Programming the AppServer."

6.1 Primary Performance Considerations

There are two fundamental performance reasons for implementing an application using Progress AppServer:

- To minimize network traffic by putting data processing closer to the database
- To offload resource-intensive processing tasks to a machine more capable of executing the tasks

These two points can work together to provide maximum performance benefits. The following sections discuss each point separately.

6.1.1 Minimizing Network Traffic

When you deploy a Progress AppServer on the database server machine and connect to a database using a self-service connection, you have the opportunity to minimize network traffic.

In a traditional client/server model, table joins and application-specific record filtering are done by the Progress 4GL client, implying that many records must be passed over the network. By using an AppServer, all this processing can be completed local to the database and network traffic is restricted to the final results being returned to the client application.

For example, consider a sales representative application. One of the application's activities might be to accumulate a set of statistics for each sales representative. These statistics might include the number of orders, the number of customers, and the average order value. In the traditional client/server model, all the records for all sales representatives are passed back to the client application where the statistical information is generated.

Using the AppServer, you can provide a remote procedure that returns a temporary table that contains a single record for each sales representative along with each representative's associated statistical information. The procedure that you write and is executed within the AppServer uses the same query that was used when the application was executed using the traditional client/server model. However, using the AppServer, the query was executed using a self-service connection to the database. Also, your procedure performs all other record processing and filtering that is necessary to reduce the data to the actual results required by the client application. The end result of transforming your application from the traditional client/server model to a model that takes advantage of the Progress AppServer is that one record is sent over the network for each sales representative rather than several.

6.1.2 Offloading Resource-intensive Processing Tasks

You can additionally obtain performance gains using the AppServer by offloading resource-intensive processing to a machine that is separate from the client machine and more capable of executing these tasks. For example, performance gains might be obtained by offloading tasks, such as complex mathematical algorithms or activity associated with transaction processing to a database server machine. These types of performance gains can be achieved with the Progress AppServer.

6.1.3 Choosing an AppServer Operating Mode

You can configure an AppServer to run in one of three operating modes:

- State-reset
- State-aware
- Stateless

In addition, stateless operating mode supports two connection states, bound and unbound, that are determined by your AppServer application at run time. For information on how each operating mode affects the operation of an AppServer installation, see Chapter 2, "Overview of the Progress AppServer."

Application Characteristics to Consider

To understand the trade-offs between these operating modes, you must first determine the goals of your application. In general, each operating mode offers different trade-offs among three general characteristics:

- Design complexity
- Resource consumption
- Performance (request throughput and response time)

The best operating mode for your application depends on a number of factors, including the:

- Complexity of the application
- Amount of context that needs to be maintained between requests
- Current and future performance goals for the application
- Computer resources available for the AppServer machine.

In some cases, it might be significantly more complex to program your AppServer application with one operating mode rather than another. In many cases there might be more than one operating mode and design strategy that is appropriate. The purpose of this section is to explain some of the trade-offs associated with each operating mode so you can determine the most appropriate operating mode and design strategy to use for your situation.

Table 6–1 lists each operating mode, describing it in terms of the three general characteristics with performance divided between throughput and response time.

Table 6–1: Characteristic Operating Mode Trade-offs *(1 of 2)*

Operating Mode	Design Complexity	Resource Consumption	Request Throughput (requests/second)	Average Response Time (seconds/request)
State-reset	Easiest to program. AppServer session context automatically reset at the end of each connection.	One client per Application Server process.	Might be limited where computing resources are limited because the number of clients that can be concurrently serviced is limited to the number of server processes.	Fast response time where computing resources are not limited. The client communicates directly with a dedicated Application Server process.
State-aware	Appropriate AppServer session context must be reset manually, typically in the Disconnect procedure. Tends to be complex if context is complex.	One client per Application Server process.	Might perform better than state-reset where reusable context can be established in the Startup procedure and then reused for each connection. Assumes minimal context needs to be reset after each connection.	Might perform better than state-reset where reusable context can be established in the Startup procedure and then reused for each connection. Assumes minimal context needs to be reset after each connection.

Table 6–1: Characteristic Operating Mode Trade-offs *(2 of 2)*

Operating Mode	Design Complexity	Resource Consumption	Request Throughput (requests/second)	Average Response Time (seconds/request)
Stateless	Appropriate AppServer session context must be established and managed for each request using the Activate procedure, then reset using the Deactivate procedure. Tends to be extremely complex if context is complex.	Many clients per Application Server process(unbound); One client per Application Server process (bound).	Maximum throughput under heavy client load where computing resources are limited and requests are short. Assumes minimal use of bound connections. Bound connections of long duration can significantly limit throughput. Even when connections are unbound, large complex connection context can also limit throughput.	Where the client load is minimal and computing resources are not limited, does not perform as well as state-aware or state-reset because two IPC messages are required per request. Bound connections of long duration can significantly limit response time. Even when connections are unbound, large complex connection context can also limit response time.

Managing Larger Amounts of Context

If you application has a large amount of context to manage between each request, then state-reset or state-aware are appropriate choices. Stateless is not a good choice because of the complexity and performance impact of managing this connection context through the Activate and Deactivate procedures for each request.

If there is a large amount of context that you need to establish for each connection that is independent of the user who is running the client application (for example, loading a TEMP-TABLE with static data), and only a small portion of the context is user specific, then state-aware might be a better choice then state-reset. This is true because with state-aware, you only need to establish the connection-independent context once, in the Startup procedure. You generally establish the user-specific context within the Connect procedure, using the user ID and password information provided by the client application.

The Startup procedure is not available for state-reset. Thus, both the connection-independent context and the user-specific context have to be established within the Connect procedure. Under these conditions, the time it takes to connect to a state-aware AppServer can be significantly shorter than the time it takes to connect to a state-reset AppServer because all of the connection-independent context is established before the client actually attempts to connect to the AppServer.

Managing Smaller Amounts of Context

If there is a minimal amount of context that needs to be maintained between each request and you have limited computing resources that need to service a large client load, the stateless operating mode might be the appropriate choice for your application. This increases your ability to service a large number of clients, because no Application Server process is dedicated to any particular client. The minimal context that needs to be maintained between requests can be stored either in the SERVER-CONNECTION-CONTEXT attribute or within an application-specific context database. No matter where you store the context, you probably want to use the Activate and Deactivate procedures to manage that context for each request.

Limiting Bound Connections

However, you need to use bound connections on a stateless AppServer with care. If client connections to a stateless AppServer are bound the majority of the time, the stateless operating mode is not a good choice, especially under heavy client load. In this case, the stateless AppServer is effectively running much like a state-aware or state-reset AppServer where each Application Server process remains effectively dedicated to one client for the duration of the bound connection.

In some cases, if most of the context that you need to maintain between requests is minimal, but occasionally you need to maintain a large amount of context for a small portion of the application, then stateless might still be an appropriate choice. In that case, you can temporarily make the connection bound by having the client run a remote persistent procedure and delete shortly after, or by setting the SERVER-CONNECTION-BOUND request attribute to TRUE within the AppServer session and setting it to FALSE shortly after.

NOTE: You might want to run all of your requests on a bound connection, if you use a *briefly-bound* approach. In a briefly-bound approach, you reserve all context maintenance to a few requests on the bound connection during processing that is free of user interaction or other client-blocking activity. Then, you make the connection unbound before performing any blocking or other intensive activity (think time) on the client. With briefly-bound connections, the AppServer remains free to service other clients most of the time and with fewer Application Server process required.

In addition, all requests for a stateless AppServer go through the Application Broker. Thus, two messages are sent for each request. One message travels between the client and the Application broker and the other message travels between the Application Broker and the Application Server process. However, because a state-aware or state-reset AppServer maintains a direct connection between the client and the Application Server process, only one message is sent per request. Under these conditions, your only choice, to both maintain AppServer context and increase the response of the AppServer to client connections, might be to use state-aware or state-reset and to add additional computing resources at the AppServer site to sustain the client connections.

Considering Ample Computer Resources

Note that for the most part, using stateless only makes sense when you must provide high throughput with minimal computing resources. Even if you only have minimal context that needs to be maintained between requests, if you have plenty of computing resources relative to your current and expected client load, then there is no inherent advantage to using stateless. Under these more favorable conditions, even if there is only minimal context to manage, using state-aware (or state-reset) might provide a somewhat simpler programming model.

Considering the Future

When considering whether to use stateless, be sure to consider your current and future performance goals. Redesigning a state-aware or state-reset application to be stateless can be very complicated. Even if your current goals do not require stateless, stateless might still be the right choice based on your future performance requirements.

6.1.4 Setting the Number of Application Server Processes

For best performance, you must determine the optimal settings for the Server Pool Parameters in the Progress Explorer:

- Initial Server Instances to Start (`initialSrvrInstance` property in `ubroker.properties`)
- Minimum Server Instances (`minSrvrInstance` property in `ubroker.properties`)
- Maximum Server Instances (`maxSrvrInstance` property in `ubroker.properties`)

In general, you must consider the number of clients, the design of your application (including the AppServer operating mode), and the hardware resources that run your AppServer. For more information, see Chapter 3, “Administration.”

6.1.5 Choosing a Transaction Model

As discussed in Chapter 2, "Overview of the Progress AppServer," the client application and AppServer run in two separate sessions. Transactions are not propagated between these sessions. However, the AppServer supports two types of transactions:

- **Normal 4GL transactions**—Where a single transaction begins and ends within the scope of a single remote procedure request from the client. The AppServer starts and manages the transaction within the scope of the remote procedure in exactly the same way a 4GL client starts and manages a transaction within a local client procedure.

- **Automatic transactions—**Where a single transaction can span multiple remote procedure requests from the client. The AppServer can terminate the transaction at any time from the moment it starts until the end of the connection.

The choice of which type of transaction to use depends entirely upon the nature of your application. By definition, automatic transactions that span more than one remote procedure request tend to be larger than normal 4GL transactions (assuming similar data transfers per request). Also, between multiple client requests with an open automatic transaction, there is more opportunity to make the transaction even larger if UI interactions occur on the client that effectively become part of the same transaction. This, in turn, creates more opportunity for lock conflicts and deadlocks among database clients, including the AppServer session.

For more information on trade-offs and issues for both types of transactions, see the "Transaction and Record Management Considerations" section.

6.1.6 Using Load Balancing

If you install the AppServer Load Balancer for a NameServer installation, you can optimize overall performance based on the machine capacity of each installation. Using this option, the NameServer distributes connection requests among the AppServer installations according to weight factors that you set on each AppServer. For more information on load balancing see Chapter 3, "Administration."

6.2 Secondary Performance Considerationss

This section highlights additional specific performance considerations that you should review and understand as you design and implement an application using the Progress AppServer.

6.2.1 Connect and Disconnect Performance Issues

This section discusses performance issues related to connecting to and disconnecting from AppServers.

As with any computer architecture that uses remote-procedure-call technology, the time that it takes to make a connection can be costly. This initial startup time is increased when you use an AppServer Connect procedure.

Using the AppServer Connect Procedure

The AppServer Connect procedure provides a convenient place to perform connection activities. Although these connection activities have value for a variety of reasons, be aware that using these could, to various extents, affect your overall connection performance.

You might want to perform any of the following activities independently of each other, or in combination with any other items:

- Authenticate and authorize based on additional parameters you can pass to the Connect procedure (*user-id*, *password*, and/or *app-server-info*).
- For a state-reset or state-aware AppServer, provide security for procedure execution by using the EXPORT( ) method to define the allowed export list for client applications.

  NOTE: For a stateless AppServer, you probably want to set the export list in the Startup procedure.

- Connect to one or more databases, or to other AppServers on a state-aware or state-reset AppServer.
- Initiate application-specific audit trails and/or activity logs.

It might be difficult to predict exact connection speed and the effects of using any, all, or some combination of these options. However, in general, each of these options has some overhead. Although none of these options are required, you might want to consider them as part of your security model. See the "Security Considerations" section for more information.

Performance-related Issue for the CONNECT( ) Method

Connecting to AppServers can be expensive due to process startup and initialization tasks defined in your AppServer Connect procedure. For this reason, you might want to connect to AppServers at client application startup. Also, in most designs, it is advisable that you do not disconnect from AppServers if you intend to use services offered by a particular AppServer more than once prior to ending the client application session.

Performance-related Issues for the DISCONNECT( ) Method

On a state-reset or state-aware AppServer, the DISCONNECT( ) method has no significant performance impact on your client application regardless of whether there is an AppServer Disconnect procedure defined. The AppServer Disconnect procedure is run by the AppServer after the client connection is terminated and DISCONNECT( ) method execution has completed. That is, unlike other procedures executed in an AppServer session, the AppServer Disconnect procedure is run by a state-reset or state-aware AppServer concurrently with processing occurring in the client application.

For a state-reset or state-aware AppServer, you might be able to take advantage of the way the AppServer Disconnect procedure is run to improve overall application performance. When designing your distributed application, consider tasks to be performed by an AppServer that could be performed independently of the connection between the client application and the AppServer. These tasks would typically be ones where the client application does not require any information about the success or failure of the operation. Time-consuming actions are very good candidates for consideration. By moving these tasks to the AppServer Disconnect procedure, these actions can be performed concurrently with other client application processing, improving overall application performance.

On a stateless AppServer, any defined Disconnect procedure runs before the client connection is terminated. Thus, especially for frequent and short-duration connections, you probably want to ensure that the Disconnect procedure does not perform any processing that significantly delays the execution of the DISCONNECT( ) method. This can both impact the client and delay the availability of an Application Server process.

6.2.2 Compiled Compared to Precompiled Code

The execution process for noncompiled code on an AppServer is identical to the process for regular run-time compiled code on a 4GL client. Standard Progress compilation processing activities and behaviors occur. Therefore, when noncompiled code must be used, from a performance perspective, there is the possibility of some improvement gains when the AppServer is running on a more powerful machine than the client application.

For example, if the AppServer you are running is on a faster machine, then the compilation process might occur faster for code compiled on it. Also, if Application Server processes have a self-service connection to the database, then compile-time schema information will not have to be sent over the network.

6.2.3 Progress Browse Design Considerations

In a 4GL client application, the Progress browse widget is bound to a query at compile time and controls record retrieval and presentation at run time.

In a traditional client/server application where a browse query satisfies a large number of records, the browse widget does not require all records to be delivered immediately; it caches some records not initially in the browse view port and, transparent to the user, requests additional records when the user scrolls to the end of the browse records. Therefore, there are no significant time delays relating to the delivery of a complete set of results prior to viewing data in the browse widget.

If browse data is to be retrieved from an AppServer, the following must occur:

1. A query running on an AppServer gathers data from the database.
2. Results are returned to the client application using temporary table parameters.
3. The 4GL client application must open the browse query against the returned temporary table.

This implies that if the remote query satisfies a large number of records, there might be a significant delay prior to viewing data in the browse widget. You might choose to accept this delay since once this delay is incurred no further network access is required. However, if this time behavior is unacceptable, consider choosing alternatives such as:

- Limiting the number of records that are delivered to the client application. For example, transfer only the first 200 records to the client application.

- Creating your own caching and delivery scheme, using the APPEND option of temporary table parameters. For example, transfer to the client application the first 200 records and an indicator as to whether there are more records available. The client application can then notify the user that there are additional records available. If the user actually requires the additional records, the client application can request the next batch of records from the Application Server process, using the APPEND option, to add to the records that already have been transferred. See the *Progress Programming Handbook* for information about the APPEND option of temporary table parameters.

  NOTE: This alternative assumes that the Application Server process building the original query is dedicated in state-reset or state-aware operating mode or is bound in stateless operating mode to the client application.

6.2.4 Performance Issues Related to Schema Transmission

When there is a remote connection to a database from a 4GL client, database schema information has to be constructed at the client. This involves network traffic to communicate the schema information from the database server to the client (even local schema caches require updates due to schema changes). For information on local schema caches, see the section on reducing connection times when connecting 4GL clients to databases in the *Progress Client Deployment Guide*.

By using the Application Server process as the only mechanism to access the database, a 4GL client no longer needs to connect to the database. This eliminates network traffic, associated with the delivery of the schema information, to the remote client.

6.3 Transaction and Record Management Considerations

The following sections describe trade-offs and techniques to help you design and implement the transaction model for your distributed application.

6.3.1 Comparing Normal 4GL and Automatic Transactions

Normal 4GL transactions are designed to keep transaction scope very small, within the context of a single remote procedure call. Small transactions minimize the chances of conflict during concurrent accesses to a single database. This type of transaction is also identical to the type of transaction supported for 4GL clients.

Because automatic transactions can remain active for multiple remote procedure calls, the scope of these transactions expands accordingly. This expanded scope allows much greater opportunity for conflict during concurrent accesses to a single database. As a result, you must be very cautious about how you use automatic transactions.

For example, you might provide an interface that closely mirrors the actual transaction mechanics, called from the client like this:

```
RUN startTransaction.p ON SERVER ...
RUN commitTransaction.p ON SERVER ...
```

Another example is the interface provided by `a-txtest.p` (see Chapter 4, "Programming the AppServer").

However, you might better simplify this process for the client with a more abstract interface, called from the client like this:

```
RUN initiateCustomerUpdate.p ON SERVER ...
RUN finishCustomerUpdate.p ON SERVER ....
```

These examples appear to accomplish the same thing. However, the first example implies that remote procedure return values monitor the transaction mechanism, while the second example implies that the remote procedure return values monitor the status of customer information. A well-designed outcome of the second example might be your guaranteeing that `finishCustomerUpdate.p` always commits the transaction appropriately, without the client having to be concerned with it.

6.3.2 Ensuring a Small Transaction Size

One of the main advantages of using normal 4GL transactions on the AppServer is their small scope, requiring no communication with the client for the AppServer transaction to complete. In general, to implement a normal 4GL transaction in an AppServer procedure, you follow the same programming practices as when writing a small-scoped transaction for a Progress 4GL client.

In particular, avoid including code that can block the Application Server process or otherwise delay execution, such as:

- Writing to or reading from external files
- Invoking any other operating system calls (for example, the OS-COMMAND statement)

For more information on implementing transactions in a 4GL procedure, see the *Progress Programming Handbook*.

6.3.3 Returning Transaction Status to the Client

When a client runs an AppServer procedure that implements a normal 4GL transaction, the client can know the status of the transaction at the point the transaction succeeds or fails. You can communicate this status on the AppServer to the client by passing back a return value (RETURN *string* or RETURN ERROR *string*) or output parameter value from the remote procedure that encapsulates the transaction.

Thus, your AppServer procedure completely hides the actual mechanics of a normal 4GL transaction from the client. For example, your return value might indicate whether or not a customer record was successfully updated. Managing the several reasons why the customer record might not be updated is the responsibility of your AppServer procedure implementation. Most importantly, once the remote procedure that implements the transaction returns to the client, all record locks are released and all records are available for access by other database clients.

6.3.4 Maintaining Transaction Consistency Between Client and AppServer

One disadvantage of using normal 4GL transactions on the AppServer is that, because a normal 4GL transaction completes and releases all locks before returning to the client, you may have to do extra work to ensure that the transaction completes as the client expects.

A typical scenario where this behavior might pose a problem is where the AppServer returns data derived from a database to a client application. The client then modifies the data and returns it to the AppServer with the intent that the changes to the data are to be applied to the database. Because database locks are not held across interactions with the AppServer, the AppServer cannot simply change the data in the database because the data might have already been changed by another user. Thus, transaction consistency in this situation cannot be easily maintained.

One approach to work around this problem, especially on a state-aware or state-reset AppServer is to have the AppServer keep a copy of the original data that is returned to the client application. On an AppServer running in state-reset operating mode, state-aware operating mode, or stateless operating mode over a bound connection, you can maintain a separate buffer or temporary table to hold a copy of the data. On a stateless AppServer over an unbound connection, you must use the SERVER-CONNECTION-CONTEXT attribute of the SESSION system handle or a context database to maintain a copy of the data. For more information, see the sections on managing stateless connection context in Chapter 4, “Programming the AppServer.”

When it is time to apply the updated data, the Application Server process handling the request determines whether the database was changed by another user by comparing the original data to the data in the database. Figure 6–1 describes this approach.

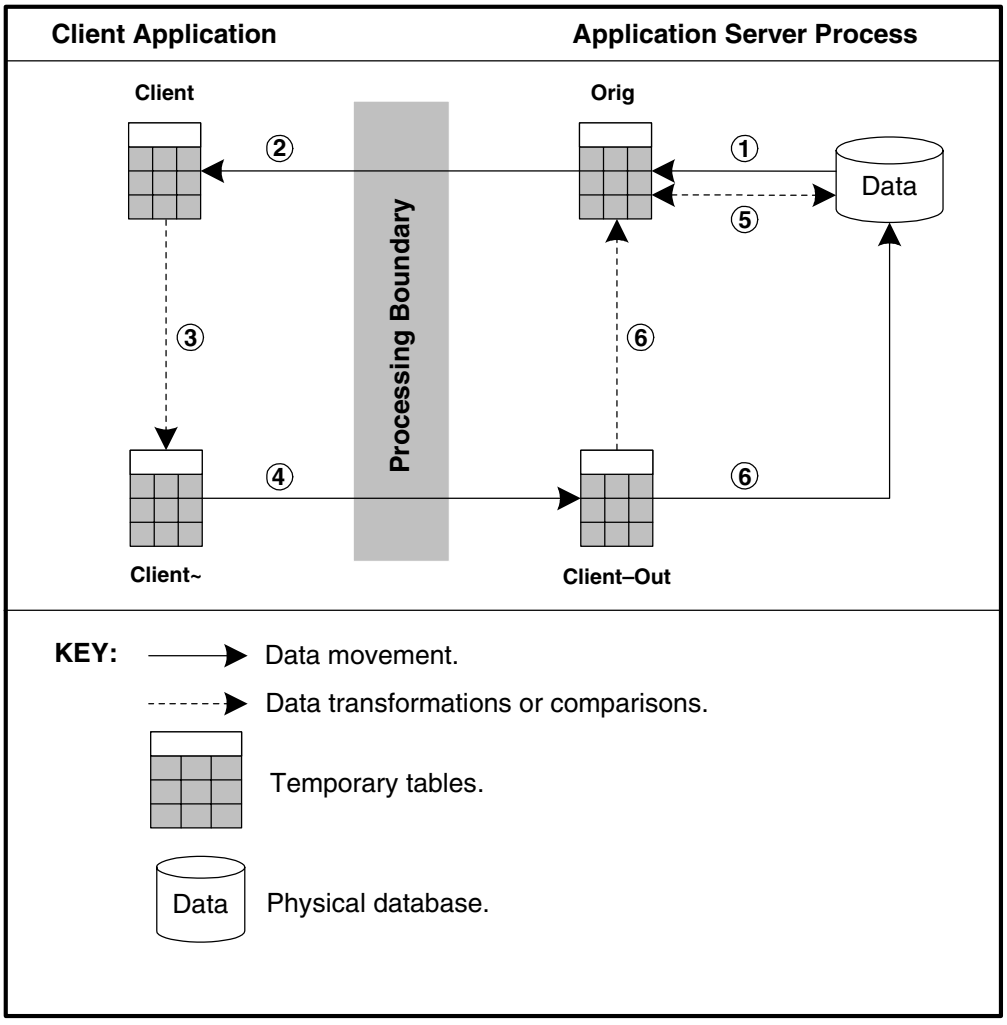


Figure 6–1: Transaction Data Flow

The numbers in Figure 6–1 correspond to the following explanation of transaction data flow:

1. The client application runs a remote persistent procedure on the AppServer to request the data. The AppServer gets the data from the database and stores it in a temporary table (**Orig**).

2. As a result of the remote persistent procedure call, the AppServer returns a copy of the **Orig** temporary table to the client application (**Client**).

3. The client application makes application-specific changes to its copy (**Client~**) that include creating, deleting, and modifying records.

4. The client application sends its modified copy (**Client~**) to the AppServer by running an internal procedure of the persistent procedure instantiated in Step 1. The AppServer stores its copy of the table in **Client-Out**.

5. Within a single transaction, the AppServer compares the records that are in the **Orig** temporary table to the data in the database. If there are any differences, the Application Server process knows that the data was changed by another user, aborts the transaction, and notifies the client application that the updates failed.

6. If the data has not been changed (within the same transaction as Step 5), the AppServer then compares the **Client-Out** table to the **Orig** table. For each record within the **Client-Out** table that has been created, modified, or deleted, the AppServer makes the same changes to the data within the database.

There are variations on the approach suggested in Figure 6–1 that provide similar results, and exactly how you vary this scheme will depend on your particular application requirements. For example, in Step 5, even if the AppServer finds a difference between the records in the **Orig** temporary table and the records in the database, it might not be necessary to abort the transaction. An alternative approach is to simply note which record has changed, find the corresponding record in the **Client-Out** temporary table returned to the AppServer, and skip that when applying the changes in Step 6. Only you can decide what approach is appropriate for your application.

6.3.5 Managing Conflicts Between Client and AppServer

All AppServer transactions (normal or automatic) occur in a context that is separate and distinct from the client. One way of understanding this is that if both a client and an AppServer access the same database, each one accesses the database as a different user, whether or not the client is connected to the AppServer. As a result, any combination of AppServer clients and AppServer sessions can cause lock contention with each other as multiple database clients of the same database. (For more information on locks and lock contention, also known as deadlock, see the *Progress Programming Handbook*.)

Thus, it is important when performing direct database access from both client and AppServer sessions that you avoid or otherwise manage the possibility that all of these sessions might concurrently access the same record. If they do, and you attempt to run a transaction that changes the record on either the client or the AppServer, the SHARE-LOCK held by each one will prevent the record from being changed, and ultimately the transaction from being committed. If the continued execution of both the client and any Application Sever session depends on the transaction being completed, then a deadlock occurs that prevents the entire application from continuing.

To avoid a deadlock, you can use the Lock Wait Timeout (-lkwtmo) startup parameter for both a 4GL client and the AppServer session (specified as part of the `srvrStartupParam` property value in your AppServer properties file). This parameter sets a limited time that any 4GL procedure must wait to continue execution because of a lock. For more information on this parameter, see the *Progress Startup Command and Parameter Reference*. However, this is not a solution to any fundamental design problems that might avoid the conflict altogether.

Another technique for managing lock conflicts is to use the NO-WAIT and NO-ERROR options on any FIND statement that also applies an EXCLUSIVE-LOCK to the record. You can then test the resulting record buffer using the LOCKED function to detect a conflict:

```
FIND FIRST customer EXCLUSIVE-LOCK NO-WAIT NO-ERROR.
IF LOCKED customer THEN RETURN "customer,FIRST,LOCKED".
```

6.3.6 Buffer Currency and NO-LOCK

If you try to find a record with NO-LOCK and that record is already located in a buffer, then depending on the exact nature of the request, Progress may not re-read the record from the database.

The fact that Progress does not always re-read NO-LOCK records may cause unexpected behavior in your distributed application. More specifically, although a record was updated within one session (Client or AppServer), that update may not be seen within another session if that record was previously read with NO-LOCK. There are two scenarios where this may occur.

These scenarios are called *currency conflicts* to denote the fact that they involve a buffer whose contents is not current:

- **Client-Server Currency Conflict**

  In this scenario, a client finds a record NO-LOCK, and then sends a request to an AppServer to update the same record. When the request finishes, the client attempts to find the same record. Because Progress may not re-read the record, the client buffer might contain the copy of the record before the update occurred rather than the updated copy of the record.

- **Server-Server Currency Conflict**

  In this scenario, a client sends a request to an AppServer whose operating mode is stateless. The Application Server process who processes the request updates some record. The client then sends another request to the AppServer. This request happens to go to an Application Server process that is different then the one that processed the first request. The Application Server process attempts to find the same record that was updated using NO-LOCK. If the record was already stored in a buffer due to some previous clients request executed at this Application Server process, Progress may not re-read the record. When this occurs, the Application Server process buffer will contain the copy of the record before the update occurred rather than the updated copy of the record.

If resolving client-server or server-server currency conflicts is important to you, there are three general approaches that you can use:

- Read the current record
- Release the record
- Set the -rereadnolock parameter

Reading the Current Record

If you know exactly which buffer is out of date, use FIND CURRENT or GET CURRENT with NO-LOCK. The CURRENT keyword indicates to Progress that the record must be re-read.

Releasing the Record

Use the RELEASE statement on all buffers that may be out of date before the record is re-read. The RELEASE statement will clear the records from all buffers to which it is applied. At that point, all records that Progress reads will need to be read from the database because a buffer copy no longer exists.

Setting the —rereadnolock Parameter

The -rereadnolock parameter indicates to Progress that when an attempt is made to find a record NO-LOCK, even if the record is already in a buffer, then the record should be re-read from the database. Use it as a 4GL Client startup parameter to resolve Client-Server Currency Conflicts. Use it as an AppServer startup parameter via the Progress Explorer or by setting the `srvrStartupParam` property in the `ubroker.properties` file for the appropriate AppServer to resolve Server-Server Currency Conflicts.

Note there are several things to keep in mind when using the —rereadnolock startup parameter:

- The -rereadnolock parameter has no affect on records that are being retrieved via RECID or ROWID. In that case, Progress will not re-read the record. It will use the copy of the record already stored in the buffer. If the most current version of the record is needed, then use the RELEASE statement on all buffers that contain a copy of the record before reading the record, or use FIND CURRENT or GET CURRENT statement to re-read the record.

- The -rereadnolock parameter has no affect on the behavior of the query cache used for a NO-LOCK query as specified via the CACHE n phrase of the DEFINE QUERY statement. If the record is in the cache, it will not be re-read regardless of whether -rereadnolock is set. To force the record to always be re-read set CACHE 0. Note that setting the cache size to zero (0) may significantly degrade performance if the database is being accessed across a network. Only set the cache size to zero (0) when it is critical to retrieve the most current version of a record.

- The -rereadnolock parameter has no affect on the behavior of the prefetch cache that is used by default when retrieving records NO-LOCK across the network. By default, when executing a CAN-FIND function, or the FIND, FOR, or OPEN QUERY statements on a database which is being accessed across a network, Progress fetches several records at a time, and stores those records within a prefetch cache. Progress will only sends a request to the database server to fetch more records if the requested record is not contained within the current prefetch cache. If the record is in this cache, a new copy of that record will not be read even if -rereadnolock is set. To eliminate this cache so that the most current version of the record is always read use the NO-PREFETCH keyword in the appropriate statements. Note that using the NO-PREFETCH keyword may significantly degrade performance. Only set NO-PREFETCH when it is critical to retrieve the most current version of a record.

6.4 Deployment Considerations

The following section highlights some deployment considerations that you might want to consider as part of your strategy for deploying an application with the Progress AppServer.

6.4.1 Minimizing the Effects of Schema Changes

In the traditional client/server model where all 4GL clients are connected to a database, database schema changes can invalidate r-code run by the client. In contrast, by designing your 4GL client application to be independent of database schema, you can use the AppServer to minimize the effects of schema changes by masking the changes from the client application.

The following list identifies specific techniques you should use to shield these schema changes from affecting client applications:

- Set up database access so that it is accomplished entirely through Application Server processes (the client is not connected to the database and all records are transferred from the Application Server process through temporary table parameters).
- Avoid using LIKE syntax for temporary tables and variables in client application code and in procedure parameters for remote procedures.
- Application Server process code must map data based on the new schema to the existing procedure parameters for the remote procedure.

6.4.2 Advantages of Using Portable R-code

R-code is the intermediate binary code that Progress generates when it compiles Progress 4GL source files. This is the code that is actually run when a procedure is executed. Progress r-code is portable between two dissimilar platforms if it does not contain any code that is dependent on the user interface or a DataServer connection.

Using portable r-code, you can perform application-server development on a platform that is different from your application-server deployment platform. For example, you can compile r-code that is destined to run on a UNIX Application Server process on a Windows 95 4GL client. Further, this means that by compiling on the Windows 95 4GL client, a compiler license for the UNIX platform is not required.

Note the following cautions if you intend to use portable r-code:

- Examine your code for platform-specific code blocks. The presence of these code blocks can limit your ability to make your r-code portable.
- Ensure that your Progress 4GL code addresses all target r-code platforms on which you want your code to run.

The following example compares platform-specific preprocessor directives that will be resolved at compile time with a comparable 4GL code that is resolved at run time.

Preprocessor directive resolved at compile time:

```
&IF &OPSYS = "WIN32" &THEN
  ...
&ELSE
  &IF &OPSYS = "UNIX" &THEN
  ...
  &ELSE
  ...
```

Non-preprocessor 4GL code resolved at run time:

```
IF OPSYS = "WIN32" THEN
  ...
ELSE
  IF OPSYS = "UNIX" THEN
  ...
  ELSE
  ...
```

If the code from the first example is compiled on a Windows NT client, the “WIN32" logic would be executed even if the resulting r-code is run by an Application Server process on a UNIX host. Therefore, if there can be multiple deployment platforms for Application Server process code, you should use run-time resolution for platform-specific blocks as the second code example in this section shows.

6.4.3 Server Configurations

The Progress AppServer architecture provides potential for improvements in performance, security, and deployment at the expense of server resources. By locating Progress AppServer processes on the same machine as your database server, you might require:

- Additional memory (physical and virtual)
- Disk space (to locate application code to be run by Application Server processes)
- Additional network resources

Actual memory requirements that extend beyond database needs vary, depending on the number and configuration of Application Brokers, Application Server processes, and NameServers, the number of clients to be serviced, the operating mode, as well as the actual 4GL procedures to be executed by the Application Server processes.

6.4.4 Network Deployment Considerations

As you plan your overall deployment strategy, keep in mind that the Progress AppServer supports only:

- Transport Control Protocol/Internet Protocol (TCP/IP) for client access to the AppServer
- Universal Datagram Protocol (UDP) for client (and AppServer) access to the NameServer

However, an Application Server process, like any 4GL client session, can communicate with databases and DataServers using TCP/IP or SNA.

If you have to connect the Progress AppServer to a Progress DataServer to access a non-Progress database (and possibly through another network protocol), do the following:

- Use the appropriate Progress DataServer.
- Use the PROBUILD utility to create a customized Application Server process executable if the appropriate DataServer is not already built into the client.

  For more information on building an Application Server process executable, see the sections on PROBUILD in the *Progress Client Deployment Guide*.

6.5 Security Considerations

The general security framework for your networking system is governed by security elements associated with your operating system, networking software, file system, application, and database. In addition to these features, the Progress AppServer offers you specific security options.

As you read through this section, consider your security goals. For example, what type of database access do you want to make available for your user community? How much control do you want to exert over individual users' ability to run specific procedures? Do you have a business need to produce audit trail details? This type of appraisal will help you determine whether to implement a tight security model, with strict control over application data, or a loose security model, with fewer access controls. That is, although each security option is unique and independent of other options, the total effect of the options you choose to implement will tend toward a tighter or looser security model.

6.5.1 Overview of the Progress AppServer Security Options

Table 6–2 summarizes the various application-specific security options available.

Table 6–2: Progress AppServer Security Considerations

Security Consideration	Design Strategy
Authentication and authorization	Use the AppServer Connect procedure to validate a user.
AppServer session access	Limit users' access to specific procedures through the EXPORT( ) method.
Database access	Limit client application access to a database. Do not allow client application access to a database.
Audit trails	Use the AppServer Connect and AppServer Disconnect procedures to assist in management of audit trails.
Run-time compilation	Protect any executable code that you deploy in source form.
Operating system	Protect the files that you deploy for an AppServer.

The remaining text in this section provides additional information about each of these considerations.

For information about developing these features, see Chapter 4, "Programming the AppServer." Also, for more information about standard Progress security features, see the *Progress Programming Handbook* and the *Progress Language Reference*.

6.5.2 Authentication and Authorization

You typically handle authentication and authorization tasks at the point of connection, using the Connect procedure, which can accept parameters for user authentication and refuse a client connection according to the result. For a state-reset or state-aware AppServer, you can take full advantage of this to authorize, on a user basis:

- What procedures (entry points) can be run by the connected Application Server process by setting an export list using the EXPORT( ) method.
- Connections to databases and other AppServers from the connected Application Server process

However, for stateless AppServers, you cannot directly authorize and implement these options at connect time.

Authorization on a State-reset or State-aware AppServer

The following code shows a typical authentication and authorization example. First, the user_id and password established by the client application's CONNECT( ) method are authenticated against an external source (in this case, valid users are identified in a database table named app_user). Secondly, the particular user is authorized to request execution of an established, user-specific, list of procedures (entry points) through the EXPORT( ) method.

```
DEFINE INPUT PARAMETER user_id         AS CHARACTER.
DEFINE INPUT PARAMETER password        AS CHARACTER.
DEFINE INPUT PARAMETER app_server_info AS CHARACTER.

/* Authenticate user - if the user/password does not exist an error
   condition will automatically be raised */
FIND app_user
  WHERE app_user.user_id = user_id
    AND app_user.password = ENCODE(password)
  NO-LOCK.

/* Authorize access to particular procedures */
IF NOT SESSION:EXPORT(app_user.can_run) THEN
DO:
  /* Log failure message to Application Server process logfile and refuse
     the connection */
  MESSAGE "Failed to create export list for" app_user.user_id.
  RETURN ERROR.
END.
```

To create a tight security model, establish an export list in conjunction with operating-system security to restrict access from the client applications host to the remote procedure sources. For more information on operating-system security, see the information on operating systems in the "Security Considerations" section. For more information on creating an export list using the EXPORT( ) method, see the "AppServer Session Access" section.

Authorization on a Stateless AppServer

For a stateless AppServer, you cannot easily set an export list and database connections at connect time, because the Application Server process that runs the Connect procedure is not necessarily the one that executes subsequent remote procedure requests for the connected client. Setting an export list and database connections works much more effectively as a global activity that you implement in the Startup procedure, which executes for every Application Server process when it starts up.

You can, however, pass connection-based authentication and authorization information to each Application Server process that handles a remote procedure request for a client connection. The Application Server process can retrieve and resave the authorization information for the next Application Server process that handles the connection using the Activate and Deactivate procedures. For more information on using these procedures, see Chapter 4, "Programming the AppServer."

6.5.3 Database Access

The traditional client/server model requires a connection from the Progress 4GL client to the database server. The Progress AppServer allows alternative strategies to be considered. One strategy is for the 4GL client to have no connection to the database and for all database access to be achieved through an AppServer. Such an implementation provides tight security since database access is limited to the parameters, and functionality contained in the procedures that are remotely executed by the AppServer.

A less severe strategy is to allow the Progress 4GL client read-only access to the database, but provide full update access to the Application Server process. This preserves full control of updates through the restricted set of procedures that can be run in an Application Server process, yet allows the client a broader range of database query capabilities. This alternative might be required to achieve certain performance objectives, for example, those peculiar to browse widget behavior. See the "Progress Browse Design Considerations" section for detailed information about browse behavior.

6.5.4 AppServer Session Access

You can publish entry points (remote procedure pathnames) that control access to an AppServer session at run time using the EXPORT( ) method on the SESSION handle. Publishing an export list the remote procedures that a client application can execute to those that you specify with this method. Because this is a run-time operation, you can change the list at any time that your application requires, and you can reset the list to empty, allowing access to the entire set of procedures available from the current AppServer PROPATH setting.

However, setting this list applies only to the AppServer session in which you execute the EXPORT( ) method. This means that you might have to manage the list differently in your application depending on the operating mode that you configure for the AppServer.

For more information on setting and resetting entry points in an AppServer session, see Chapter 4, "Programming the AppServer."

Operating Mode Interactions

Because of how the EXPORT( ) method and AppServer operating modes work together, you must use it differently, depending on your AppServer operating mode. When you invoke the method with a list of procedure names, the method adds this list to any existing list of entry points that you have previously specified for an AppServer session.

If the AppServer is running in state-reset operating mode, each Application Server process automatically resets any export list to empty with each new connection. However, if the AppServer is running in state-aware or stateless operating mode, you must manage all setting and resetting of entry points manually. Otherwise, each time you execute the method with a list of procedure names, they are added to the existing list of entry points. Thus, unless you explicitly reset the list, on a state-aware AppServer, the number of entry points set for an Application Server process can grow with each client connection, and on a stateless AppServer, the number can grow with each client request.

How to Set and Reset an Export List for Each Operating Mode

Each AppServer operating mode requires a different arrangement for setting and resetting the published entry points during an AppServer session:

- **State-reset operating mode**—You can set an export list any time before a client application uses them; you never need to reset your an export list.
- **State-aware operating mode**—You can set an export list any time before a client application uses them; if you set them in remote procedures, you probably want to reset them with each connection.
- **Stateless operating mode**—You can set an export list any time before a client application uses them; you generally set them in the Startup procedure for the entire AppServer session.

State-reset Operating Mode

When the connection terminates on a state-reset AppServer, the export list for the disconnected Application Server process is automatically reset to empty. Thus, you must set any export list on a state-reset AppServer for each new connection and in some procedure that is guaranteed to be executed when you need to have the list set.

You might want to set the export list in the Connect procedure if you know the export list criteria at the start of each connection. The criteria can be based on the user or any other information that you determine at the time of connection. Otherwise, you must set or reset the list in some remote procedure executed by the client.

State-aware Operating Mode

For a state-aware AppServer, you can conveniently set a single export list for all Application Server processes, regardless of the connection, using the Startup procedure. Otherwise, you can set it, like on a state-reset AppServer, in the Connect procedure or some remote procedure executed by the client.

When the connection terminates on a state-aware AppServer, the disconnected Application Server process maintains the last setting of the export list. If you want the export list to be empty at the start of every connection, you must explicitly reset the list at some point between connections. You can guarantee this for every AppServer session by resetting the list in the AppServer Connect or Disconnect procedure:

```
ret = SESSION:EXPORT(). /* Reset to empty */
```

Stateless Operating Mode

For a stateless AppServer, you can conveniently set a single export list for all Application Server processes (global to the AppServer session), using the Startup procedure. Because export list settings apply only to the Application Server process in which they occur, setting and resetting an export list at any other point on a stateless AppServer requires careful attention.

CAUTION: Setting an export list during remote procedure requests to a stateless AppServer is an error-prone activity. If your application requires this fine degree of control, the rest of this section explains how you can do it. However, you might consider implementing your AppServer using state-aware operating mode, instead.

Like a state-aware AppServer, once you set an export list, the list remains set in the AppServer session until you reset it. For an unbound connection, because you cannot guarantee when an Application Server process will execute what procedure, you must manage any export list settings appropriately during each request. If you do not, you can leave an unpredictable set of export list settings across Application Server processes in the AppServer pool.

For an unbound connection, you can maintain consistent export list settings using the Connect, Activate, and Deactivate configuration procedures. This is especially useful to provide user-based access control:

1 ♦ In the Connect procedure, establish an export list based on the authenticated user.

2 ♦ Save the export list using stateless context storage, such as the SERVER-CONNECTION-CONTEXT attribute or a context database.

3 ♦ In the Activate procedure, retrieve the export list from your stateless context storage and set it using the EXPORT( ) method. The list is now set to filter the current remote procedure call and ensure its validity for the user.

4 ♦ In the Deactivate procedure, reset the export list to empty or to any other global session value using the EXPORT( ) method. This leaves the Application Server process to handle remote procedures calls from all connections, as appropriate.

NOTE: The Disconnect procedure is not effective to reset a user-based export list, especially for an unbound connection, because the Disconnect procedure can run in an Application Server process other than the one where the last export list setting occurred.

You can also use the Activate procedure to make application-based settings, where you set the list depending on some application state other than the authenticated user. In this case, you might determine the setting indirectly from stateless context storage or from some other source available to the 4GL. For more information on using stateless context storage, see Chapter 4, "Programming the AppServer."

6.5.5 Generating Audit Trails

Certain classes of applications require audit trails to be generated to trace events such as:

- Who connected to a particular service and when they connected
- What activity they performed during the life of the connection
- What breaches in security they tried to perform during the life of the connection

AppServer Application Audit Trails

In an AppServer session, you can access the current connection ID using the SERVER-CONNECTION-ID attribute on the SESSION handle.

Ideal locations to generate an audit trail on an AppServer include the AppServer Connect and Disconnect procedures. For example, your Connect procedure can use the *user-id* parameter and the SERVER-CONNECTION-ID attribute to write an audit trail record when a client application connects to the AppServer. Also, any code to create an audit trail record in the Disconnect procedure is always executed, since this procedure is guaranteed to run whether the connection is explicitly disconnected or forcibly disconnected in response to the QUIT condition.

On an AppServer operating in stateless operating mode, you can also use the AppServer Activate and Deactivate procedures to generate audit trails for each request. However, this use can reduce performance, depending on the load on your AppServer and the amount of information in the audit trail.

For more information on programming these procedures and on accessing the SERVER-CONNECTION-ID attribute, see Chapter 4, "Programming the AppServer."

NOTE: Audit information created in an Application Server process using the MESSAGE statement or other external output syntax such as the DISPLAY statement, is sent to the AppServer log file unless otherwise directed.

Client Application Audit Trails

In a 4GL client session, you can access the current connection ID using the CLIENT-CONNECTION-ID attribute on the server object handle used to connect the AppServer. For an open client (Java or ActiveX Controller) application, Progress provides appropriate methods for you to access the connection ID.

Typically, you might generate audit trail information on the client in code associated with AppServer connection and disconnection requests, or even with each remote procedure request, depending on how much information you need to capture.

For more information on accessing the CLIENT-CONNECTION-ID attribute from a 4GL client application, see Chapter 5, "Programming the Client Application." For information on the methods for accessing the connection ID from an open client, see the *Progress Open Client Developer's Guide*.

6.5.6 Run-time Compilation

Compiling code at run-time (compile-time execution) gives you the capability to execute dynamic procedures that execute queries and other immediate compile-and-run-related activities. (Although, with Version 9, you are more likely to use dynamic queries than dynamic procedures to execute immediate queries.) Compile-time execution can be an asset while developing an application, but a security issue when deploying, if other safeguards are not provided to protect the source code.

If an application is deployed using runtime compilation, ensure that your security framework is tightly defined so that your other security efforts cannot be violated. For example, you should ensure that all source files are appropriately secured at the operating-system level.

6.5.7 Operating System

In any application environment, access to directories and files can pose a security risk. To provide a tightly secured environment in which to run the Progress AppServer, do not allow client access (using remote logins, shells, copying, or file sharing) to directories that contain:

- AppServer startup/shutdown scripts
- AppServer log files
- Application code to be executed by Application Server processes

6.6 Multi-language Support

Progress supports multi-language (for example, French and German) applications through use of the CURRENT-LANGUAGE statement and function. You can use this 4GL support in AppServer procedures to coordinate different languages between the client and Application Server process. In stateless operating mode, you must use the activate procedure to set the correct language for each client request.

If you do write a multi-language application, you must ensure that the client and your server-side code handle the following features in a compatible fashion:

- **Collation**—If the Application Server process and client use different collations, the client might need to re-sort output TEMP-TABLE or result set parameters. A 4GL client can do this by defining its own index on the appropriate columns.
- **Date and number formats (-d and -E startup parameters)**—To minimize any problems using different date and number formats, always exchange dates as a 4GL DATE values and exchange numeric fields as 4GL INTEGER or DECIMAL values, rather than as character strings.
- **Case rules (such as in BASIC and French)—**Based on the anticipated language environment, you can specify tables at AppServer startup (using the -cpcase startup parameter) that the Application Server process uses to handle case conversions.
- **Year offset (-yy startup parameter)**—Using the Century (-yy) startup parameter, you can ensure that the Application Server process calculates dates based on a specified hundred-year period that is compatible with the client.

For more information on using the CURRENT-LANGUAGE statement and function in a Progress session, see the *Progress Internationalization Guide*.

6.7 Distributed Application Design and Implementation

This section provides some suggestions and techniques to write distributed applications using the Progress AppServer, and to migrate your application code from your present configuration to a distributed Application Server process environment.

6.7.1 Understanding the Scope of Persistent Procedures

A remote procedure on the AppServer that you run with the PERSISTENT option, and a local procedure that you run with the PERSISTENT option both perform in identical fashion. That is, its context is created when the instantiating procedure starts to execute and that context persists after it returns until the end of the Progress session, or until it is explicitly deleted.

When a remote procedure is run persistently, the persistent context is managed strictly within the AppServer session where the persistent procedure is run. This is consistent with the existence of the processing boundary that separates an AppServer session distinctly from a client application session. If a remote procedure creates other local Progress objects (persistent procedures), these objects are only available locally within the Application Server process in which they were created. These objects cannot be shared between a 4GL client application and the AppServer session.

6.7.2 Understanding Procedure Handles in Distributed Progress Sessions

Every active Progress procedure, whether running on a 4GL client or in an Application Server process, has a procedure handle associated with it. Among the various attributes of the procedure handle, the PROXY, REMOTE, and PERSISTENT attributes combine to provide important information about the procedure and its handle in a distributed application environment.

REMOTE Attribute

The THIS-PROCEDURE system handle returns the current procedure handle value from within any executing block of an external procedure. A current procedure handle created within an AppServer session for a remote procedure request is a *remote procedure handle*. Its REMOTE attribute is, therefore, set to TRUE. If the REMOTE attribute is FALSE, then the current procedure was created as the result of a *local procedure* call (that is, a procedure call initiated and executed within the current Progress session context).

PERSISTENT Attribute

When you execute a persistent procedure from a local procedure call, as with any procedure, the persistent procedure can reference its own context using the THIS-PROCEDURE system handle. This system handle returns a procedure handle to the persistent procedure's own context.

Also, the procedure that executes the RUN statement that creates the persistent procedure can obtain a reference to the same persistent procedure context using the SET option of the RUN statement. This option returns the same procedure handle as the THIS-PROCEDURE system handle accessed from within the persistent procedure.

Thus, for a local persistent procedure, there is only one handle for a specific persistent procedure context, but there might be many references to it. For any such reference to a persistent procedure handle, the PERSISTENT attribute returns the value TRUE.

PROXY Attribute

When a 4GL client application executes a remote persistent procedure, two persistent procedure handles are created: one within the client application session and another separate handle within the AppServer session where the persistent procedure is created. Progress internally maintains a mapping between the two handles. The handle within the client application is a *proxy persistent procedure handle*, and its PROXY attribute and its PERSISTENT attributes are set to TRUE, but its REMOTE attribute is FALSE. The handle within the Application Server process is a *remote persistent procedure handle*, and its REMOTE attribute and the PERSISTENT attribute are set to TRUE, but its PROXY attribute is FALSE.

Unlike persistent procedure handles used for local procedures, the proxy persistent procedure handle and the remote persistent procedure handle are truly separate handles. For example, setting the PRIVATE-DATA attribute on a remote persistent procedure handle has no effect on the PRIVATE-DATA attribute of the corresponding proxy persistent procedure handle.

How the Attributes Work Together

Table 6–3 summarizes the settings of the PROXY, REMOTE, and PERSISTENT procedure handle attributes for various types of procedure call scenarios.

Table 6–3: PROXY, REMOTE, and PERSISTENT Procedure Handle Attribute Settings

Procedure Call Type	Caller Handle	Callee Handle (THIS-PROCEDURE)
Local Procedure	Not applicable	PROXY = FALSE REMOTE = FALSE PERSISTENT = FALSE
Local Persistent Procedure	PROXY = FALSE REMOTE = FALSE PERSISTENT = TRUE	PROXY = FALSE REMOTE = FALSE PERSISTENT = TRUE
Remote Procedure	Not applicable	PROXY = FALSE REMOTE = TRUE PERSISTENT = FALSE
Remote Persistent Procedure	PROXY = TRUE REMOTE = FALSE PERSISTENT = TRUE	PROXY = FALSE REMOTE = TRUE PERSISTENT = TRUE

In Table 6–3, Caller Handle refers to the handle returned by the RUN statement. This column is relevant only for persistent procedures because you can only reference the procedure handle of a procedure that you call persistently using the SET option of the RUN statement. Callee Handle refers to the procedure handle within the procedure that is being executed. You can reference this handle using the THIS-PROCEDURE system handle, whether or not the current procedure was called persistently.

6.7.3 Understanding Condition Handling in Distributed Progress Sessions

The processing boundary that exists between a 4GL client application session and an AppServer session influences how error conditions are viewed and handled in each session. Table 6–4 defines the four basic Progress conditions and describes how raising these conditions in one session affects the processing in the other, associated session.

Table 6–4: Progress Conditions *(1 of 2)*

Condition	Description
ERROR	An unhandled ERROR condition raised in a client application has no effect on any of the Application Server processes to which it is connected. Conversely, an unhandled ERROR condition raised in an Application Server process has no effect on the client application to which it is connected. Handling this condition conforms to standard Progress rules. For more information about these rules, see the *Progress Programming Handbook*. The only way an AppServer session can raise ERROR in the client application is for a remote procedure or an Activate procedure to execute the RETURN ERROR statement. This raises ERROR on the RUN statement in the client application.
ENDKEY	An unhandled ENDKEY condition raised in a client application has no effect on any of the Application Server processes to which it is connected. Conversely, an unhandled ENDKEY condition raised in an Application Server process has no affect on the client application to which it is connected. Handling this condition conforms to standard Progress rules. For more information about these rules, see the *Progress Programming Handbook*. An AppServer session raises the ENDKEY condition if an attempt to find a database record fails or an INPUT FROM statement reaches the end of an input stream. Like a batch client, there are no user-interface events that can raise the ENDKEY condition.

Table 6–4: Progress Conditions *(2 of 2)*

Condition	Description
STOP	A STOP condition raised in a client application while it is executing a remote procedure causes a STOP condition to be raised in the context of the executing procedure in the AppServer session. An unhandled STOP condition in an AppServer session results in the request being returned with the STOP condition. **On a root client application**, an unhandled STOP condition causes Progress to restart the client Startup Procedure (-p). This restarted session retains connections with databases, but deletes outstanding persistent procedures. This restart also disconnects any Application Server processes it is connected to and deletes active remote persistent procedures. **In an Application Server process**, an unhandled STOP condition causes the STOP condition to be propagated back to the client application where it is raised again in the context of the outstanding remote procedure request. It does not cause a restart of the AppServer session. All active remote persistent procedures and Application Server process connections remain intact.
QUIT	An unhandled QUIT condition raised in a client application disconnects the client application from all Application Server processes it is connected to and deletes the proxy persistent procedure handles for that client application. An unhandled QUIT condition raised in an AppServer session results in the immediate, successful completion of the current remote procedure request. The client application is then automatically disconnected from the Application Server process.

6.7.4 Schema Triggers

Schema trigger code defined in the database is executed by database clients. Any database access by an Application Server process that invokes database trigger code causes the Application Server process to run such code. This is because the AppServer application acts as a client to the database.

There is no functional difference between how schema triggers run in an AppServer session and how they run in a 4GL client session. However, trigger code executed by an Application Server process:

- Fails if it requires user input or output to a user interface other than a character user interface
- Sends any output to the AppServer log file, unless specifically directed otherwise

Invoking schema triggers on a client application raises a basic security issue. Code in the Application Server process is protected because of *encapsulation*; code on your client application is not protected.

6.7.5 Conditions That Make Migrating Your Application Model Easier

Certain conditions related to your current application framework allow you to more easily take advantage of Progress AppServer functionality. These conditions include any of the following:

- Your present server has adequate resources such as memory, speed, and disk space to be easily scaled up for use with the Progress AppServer.
- Your current model has a predominance of small, well-structured, identifiable transactions. The Progress AppServer favors a small transaction model. For more information, see the "Transaction and Record Management Considerations" section.
- You are already using temporary tables to manipulate data before committing the data to the database. Or, if you are not using temporary tables, it is easy to get your present code to work with temporary tables rather than deal directly with your database.
- Your current model is already set up such that neither procedures nor persistent procedures rely on shared/global data or buffers as parameters. These types of information cannot be shared between a client application and an Application Server process.
- Your current model already has a distinct division between the logic associated with the user interface and the logic associated with your application. This enables your code to be modular and portable, in keeping with how you will want your code to be deployed in a distributed environment using the Progress AppServer<$startrange>Design and implementation:designing your distributed application model.

7

Debugging

This chapter presents information about how to examine code associated with a distributed application that uses the Progress AppServer. Specifically, this chapter:

- Describes how the Progress Debugger supports debugging distributed applications.
- Identifies some AppServer log file information that you might want to examine to help you troubleshoot your application.

7.1 AppServer Debugging

The Progress Debugger supports debugging application code in a distributed environment in two modes. These modes provide two different models for debugging your distributed application. No matter what mode you are using, standard debugging commands, such as executing one or more lines of a procedure and its subprocedures, breaking and continuing at specified lines, displaying data from a procedure that you are running, and modifying any of this data, can now be used to debug remote procedures.

The two debugging modes include:

- Distributed debugging
- Remote debugging

To initiate any debugging on an AppServer, you must run it with the `debuggerEnabled` property (in the `ubroker.properties` file) set to 1.

The following sections describe each of these modes. For a complete description of the Progress Debugger, see the *Progress Debugger Guide*.

7.1.1 Distributed Debugging

The *distributed debugging* allows you to debug a several associated 4GL sessions within a distributed application environment. That is, one debugger session, running in one session within the distributed environment, controls the debugging process for all sessions associated with it. The procedure call stack reflects all procedures in all sessions involved, enabling the root client application to be aware of the AppServer sessions that it is debugging. The *root client application* is the client application that initiates the Debugger using distributed debugging.

You can only use distributed debugging on AppServers running in state-reset and state-aware operating modes.

You can run the Debugger using distributed debugging to debug local procedures, remote procedures, and remote procedures executed by other remote procedures. An Application Server process that is once, twice, or further removed from the initial debugger session can be debugged in this manner. More information on this debugging mode follows in later sections.

7.1.2 Remote Debugging

In *remote debugging*, the Debugger is initiated directly by 4GL code running in an AppServer session. Remote debugging is the only debugging mode that you can use to debug remote procedures that are:

- Running on a stateless AppServer
- Executed by an Open Client (non-4GL) application
- Executed asynchronously in a 4GL client application

This mode restricts your debugging activities to debugging only AppServer configuration and remote application procedures; you cannot debug any procedure in the client application session. It allows you to debug application code running on an Application Server process independently of the calling client application.

For example, an engineer who has developed a set of AppServer procedures does not have to examine the client code that his AppServer code will eventually support. When he is ready to debug his work, he only needs to examine his own code. In this situation, remote debugging is the ideal choice. When he debugs his code using this mode, he sees only his code on the Application Server process. If the Application Server process where the engineer has developed his code calls remote procedures on other Application Server process processes, he can also debug these other remote procedures using this mode.

In this debugging mode, the client application driving the Application Server process is unaware that the Debugger process is running on the Application Server process. That is, the procedure call stack only reveals entries up to the top level remote procedure call. It does not reveal anything about the client application.

7.2 Using Distributed Debugging

Distributed debugging is the mode you are most likely to use for 4GL client applications that drive state-reset or state-aware AppServers because it more naturally follows the structure of your distributed application. It allows you to fully examine your procedures just as they are invoked in this environment.

NOTE: You cannot use this mode for Java and ActiveX Controller client applications.

From your root client application, you can initiate distributed debugging from the Procedure Editor or the AppBuilder Tools. Then you can debug remote procedures in any of the connected AppServer sessions. Next, you can set breakpoints, step into remote procedures, and examine program variables and buffers in an integrated, rather than a disjointed manner.

7.2.1 About the Debugger Window

When you are using distributed debugging, the Debugger window and the procedure call stack help orient you to the procedures available to debug. For example, the title bar of the Debugger Listing Panel changes as you change the procedures that you are debugging. Table 7–1 indicates the dynamic information changes that occur in the title bar.

Table 7–1: Content of the Debugger Listing Panel Title Bar

When you are stepping through a...	Then the title bar reads...
Local procedure, such as `example.p`	`example.p` (LOCAL)
Remote procedure, such as `remote.p`, on an Application Server process whose server handle's NAME attribute is RemoteServer1	`remote.p` (ON SERVER RemoteServer1)
Remote procedure, such as `distant.p`, on an Application Server process whose server handle's NAME attribute is RemoteServer2, which was run from a procedure on RemoteServer1 (RemoteServer2 is an Application Server process once removed from the root client application.)	`distant.p` (ON SERVER RemoteServer2 via RemoteServer1)

The procedure call stack reflects the entire call stack for the distributed application. That is, the SHOW STACK command displays location information for each procedure call stack entry.

For example, if `example.p`, line 4, runs `remote.p`, and `remote.p`, line 3 runs `distant.p` and you are stepping through line 17, the SHOW STACK displays the following information.

```
example.p():4
remote.p(...):3 ON SERVER RemoteServer1
distant.p(...):17 ON SERVER RemoteServer2 Via RemoteServer1
```

The following list notes other observations to keep in mind when using distributed debugging:

- Because the Debugger is initiated by the root client application, a version of the `prodebug` executable does not have to be running on the Application Server process that you are debugging.

- The root client application must run in an environment that supports the Debugger. For example, if your root client application is on a UNIX platform, you need to have Motif running on that platform so that you can display and use the Debugger Window. (On UNIX, the Debugger is a Motif application.)
- You can only examine variables within an AppServer session when code is being executed on the AppServer as indicated by the procedure call stack.

7.2.2 Restrictions on Debugger Operation

Note the following restrictions when running the Debugger for a distributed application:

- You cannot execute a RUN command from the Debugger window command line while the current procedure is running in an AppServer session.
- The Debug menu Interrupt option is not supported when executing remote 4GL code. For information on this option, see the section on the Debug menu in the *Progress Debugger Guide*.

7.3 Using Remote Debugging

You use remote debugging to debug an AppServer application independently of the client application or to debug any AppServer application running in stateless operating mode. With remote debugging, the Debugger does not have access to the client application that calls remote procedures in the AppServer session. When viewing the procedure call stack associated with a debugging session initiated in this mode, you never see procedure call stack entries for any 4GL client application that might be driving the AppServer.

Because the Debugger is a 4GL debugger only, you must use remote debugging to debug AppServer procedures driven by Java or ActiveX Controller client applications.

You must also use remote debugging if you want to step through code in any of the AppServer configuration procedures that you specify in the AppServer properties file (`ubroker.properties`). These include the:

- Startup procedure (`srvrStartupProc` property)
- Shutdown procedure (`srvrShutdownProc` property)
- Connect procedure (`srvrConnectProc` property)
- Disconnect procedure (`srvrDisconnProc` property)
- Activate procedure (`srvrActivateProc` property)
- Deactivate procedure (`srvrDeactivateProc` property)

The Debugger does not have access to these procedures with distributed debugging.

7.3.1 Basic Tasks

To execute a remote debugging session you must both invoke the Debugger process and set an initial breakpoint from within your remote 4GL code. For more information on general debugging tasks, such as using the Debugger window and exiting the Debugger process, see the *Progress Debugger Guide*. This section describes the basic tasks required to initiate remote debugging.

Invoking the prodebug Process

To debug using remote debugging, you must explicitly invoke the debugger using the methods of the DEBUGGER system handle executed from an AppServer session. For more information on the DEBUGGER system handle, see the *Progress Debugger Guide* and the reference entries for the handle, its methods, and its attributes in the *Progress Language Reference*.

On UNIX platforms, the Debugger process (prodebug) runs as a Motif application, and it requires setting either the PDDISPLAY or DISPLAY environment variable. These variables tell prodebug where to realize its Debugger window. Because of how prodebug starts, the environment variable must be set before the AppServer is started. That is, as part of the startup process for an AppServer, the Application Server process inherits the environment variables set up for the Application Broker. Similarly, when the Application Server process starts up prodebug, prodebug inherits these same environment variables, including PDDISPLAY and DISPLAY, from the Application Server process.

Establishing an Initial Breakpoint

With remote debugging, you must programmatically set an initial breakpoint in the 4GL code by using the SET-BREAK( ) method of the DEBUGGER handle. For information on the DEBUGGER handle, see the *Progress Debugger Guide* and the *Progress Language Reference*.

The following example shows how to use the INITIATE( ) and SET-BREAK( ) methods for the DEBUGGER handle.

```
DEFINE VARIABLE ret AS LOGICAL.
ret = DEBUGGER:INITIATE().
ret = DEBUGGER:SET-BREAK(). /* Set-Break() with no parameters breaks at next
line */
RETURN.  /* debugger gets control at this line */
```

7.4 General Debugging Requirements

The following requirements must be satisfied before you can debug an AppServer using either distributed debugging or remote debugging:

- You must configure the AppServer for debugging by setting the debuggerEnabled property to 1 in the AppServer properties file (ubroker.properties). If you do not start an AppServer with this property setting, you cannot use the Debugger to debug code on the AppServer.
- When you set up an AppServer for debugging, it does not activate the Progress Debugger when the AppServer starts. Rather, it identifies each Application Server process as available for debugging by a client application. In contrast, when you startup a a 4GL client using the Debugger (-debug) startup parameter, the Debugger is your immediate point of entry. For more information on starting and stopping the Debugger, see the *Progress Debugger Guide*.

- To use the Progress Debugger with:
  - Distributed debugging, you must start it on the platform where the 4GL client is running.
  - Remote debugging, you must start it on a UNIX platform where the AppServer is running. You cannot do remote debugging on a Windows platform.

  The Debugger is currently supported for a 4GL client running under Windows NT or Windows 95/98 and on all Progress-supported UNIX platforms running Motif. You cannot use the Debugger for remote debugging on Windows platforms because it relies on X-Windows, which only works on UNIX.

7.5 Examining the Progress AppServer Log File

When you investigate errors and troubleshoot code, examine the contents of the Progress AppServer log (specified using the `srvrLogFile` property in `ubroker.properties`). It can contain Progress system messages (informational, error, and warning) and output from 4GL MESSAGE statements that are not assigned an output destination by the OUTPUT TO KEEP-MESSAGES statement. For more information, see the OUTPUT TO statement entry in the *Progress Language Reference*.

Also, you can choose to track and record procedure information using the `loggingLevel` property in the `ubroker.properties` file. This property tells an Application Server process to put procedure call trace messages into the Progress AppServer log file. Specifically, it issues a start message when a remote procedure begins and an end message when a remote procedure returns. Then, when you see a system message in the Progress AppServer log file, you can determine which procedure generated a specific message.

You can specify three logging levels for the `loggingLevel` property:

- Level 1—Errors and information
- Level 2—Terse (the default)
- Level 3—Verbose

Level 1—Errors and Information

This includes all error messages and informational messages that are very infrequent. These include startup and shutdown messages that occur once during the life of the AppServer session.

Level 2—Terse (the Default)

This includes Level 1 plus information messages that occur very infrequently, but also indicate that the system is operating properly. These include AppServer registration events, client connection and disconnection events, and so on.

Level 3—Verbose

This includes Level 2 plus informational messages that occur more frequently and might help the client to debug their application. These include messages such client requests to the Application Broker, keep-alive messages from the broker, and logging of each remote procedure execution event.

For information on maintaining the AppServer log file, see Chapter 3, “Administration.”

A

Running the AppServer Without a Network

If necessary (for example, when no network is available), you can run the AppServer in a standalone environment. On UNIX, you only need to specify `localhost` as the NameServer and AppServer host name for connections and administrative functions. However on NT 4.0, you must prepare the machine for standalone operation before you begin operation.

This appendix describes:

- Preparing NT 4.0 for standalone operation
- Creating a non-network configuration
- Installing the Remote Access Service

A.1 Preparing NT 4.0 for Standalone Operation

To prepare NT 4.0 for running the AppServer in a standalone environment, you must create a non-network configuration and install the Remote Access Service to allow the AdminServer to successfully load.

Whenever you reboot after completing these procedures, Windows prompts you to specify a configuration. When you want to run the AppServer standalone, choose the No Network configuration that you create using the following procedures.

A.2 Creating a Non-network Configuration

To create a non-network configuration, follow these steps:

1 ♦ From the Control Panel, run the System application.

2 ♦ Choose the Hardware Profiles tab and make a copy of your Original Configuration.

3 ♦ Name the new profile No Network and choose OK.

4 ♦ Choose the Properties button to display a General and Network tab.

5 ♦ Choose Network and check the Network-disabled hardware profile toggle-box.

6 ♦ Choose OK to finish creating the configuration.

A.3 Installing the Remote Access Service

To install the Remote Access Service, follow these steps:

1 ♦ From the Control Panel, run the Network application.

2 ♦ Choose the Services tab.

3 ♦ Choose the Add button.

4 ♦ Select Remote Access Service from the list and choose OK

5 ♦ Choose OK to finish the installation.

6 ♦ Reboot the system, and when prompted for a configuration, choose No Network.

You can now run an AppServer and its utilities on your system in a standalone environment.

Index

A

Activate procedure
  overview 4–9
  usage requirements 4–11

ActiveX Controller clients 2–2

Administration framework 3–2
  preparing to use 3–8

Administrative tasks
  configuring AppServer components 3–8
  running without a network A–1
  setting up the execution environment 3–17
  starting and managing AppServers 3–19
    NT requirements 3–9
    UNIX requirements 3–9
    using command-line utilities 3–22
    using Progress Explorer 3–20
  summary 3–30

AdminServer
  definition 3–3
  starting
    on NT 3–31
    on UNIX 3–31

APPEND option 6–12

Application Broker
  administration 3–6
  definition 2–17
  overview 2–4

Application partitioning
  definition 1–2
  Progress AppServer 2–2
    reasons to implement 2–2

Application PROPATH
  AppServer setting 3–12

Application Server
  administration 3–7
  customizing the executable 3–18
  definition 2–17
  overview 2–4

Application Server session
  definition 2–6

Application Service
  assigning names 2–23
    using Progress Explorer 3–11
  default service 2–24
  definition 2–23
  load balancing support 2–23
  overview 2–18

Application Service names
  AppServer setting 3–11

AppServer, *See also* Progress AppServer

AppServer execution
  preparing the environment 3–17

AppServer installation
  definition 2–4

AppServer instance
  definition 2–4

AppServer log files
  debugging 7–8

AppServer session
  connection management
    state-aware operating mode 2–28
    stateless operating mode 2–30
    state-reset operating mode 2–27
  context management
    state-aware operating mode 2–29
    stateless operating mode 2–31
    state-reset operating mode 2–28
  definition 2–6
  performance
    state-aware operating mode 2–29
    stateless operating mode 2–31
    state-reset operating mode 2–28

-AppService connection parameter 5–10

ASBMAN utility 3–5
  examples 3–25
  syntax 3–23
  using 3–22

ASCONFIG utility 3–8
  example 3–17
  syntax 3–16
  using 3–15

Asynchronous and synchronous requests 2–32
  examples 5–38
  process control flow 2–33

ASYNCHRONOUS option 5–17

Asynchronous remote procedures
  debugging 7–3
  error handling 5–31
    compared to synchronous 5–27
  obtaining return value 5–33
  overview 5–14
  returned parameters 5–31

Asynchronous request handles 5–2, 5–28

Asynchronous request return values 5–33

Asynchronous requests
  4GL execution model 5–40
  cancelling 5–34
  debugging 5–36
  definition 2–3
  deleting
    asynchronous request handles 5–36
    persistent procedures 5–35
    server handles 5–35
  disconnecting AppServers 5–24
  examples 5–38
    asynchronous comparison 5–40
    complete client and server 5–42
    synchronous comparison 5–38
  executing 5–28
  handling response 5–29
  managing 5–27
  mixing with synchronous 5–36
  persistent procedures 5–35
  queue size 2–36
  queuing model 2–35
  queuing overview 2–34
  results overview 2–34
  running on a local SESSION 5–37

ASYNC-REQUEST-COUNT attribute 5–2
  coordinating request types 5–36
  deleting persistent procedures 5–35
  deleting server handles 5–35

Attributes
ASYNC-REQUEST-COUNT 5–2
coordinating request types 5–36
deleting persistent procedures 5–35
deleting server handles 5–35
PROCEDURE-COMPLETE event 5–29
CLIENT-CONNECTION-ID 5–13
DEFAULT-COMMIT 4–26
FIRST-PROCEDURE 5–19
LAST-PROCEDURE 5–19
PERSISTENT 6–33
PROXY 6–33
REMOTE 6–33
SERVER-CONNECTION-BOUND 4–18
SERVER-CONNECTION-BOUND-REQUEST 2–31, 4–16
SERVER-CONNECTION-CONTEXT 2–32, 4–7, 4–9, 4–10, 4–15
SERVER-CONNECTION-ID 4–4, 4–9, 4–10, 4–16
SERVER-OPERATING-MODE 2–27
TRANSACTION 4–23
TRANS-INIT-PROCEDURE 4–26

Audience xiii

Audit trails 6–30

Auto startup
AppServer setting 3–11

Automatic transactions 4–22
controlling 4–23
example 4–27
implementing 4–23
restarting 4–26
terminating 4–25

AVAILABLE status 3–26

B

Bold typeface
as typographical convention xvi

Bound connections 2–30
setting 4–17

Browse widget 6–11

Business logic
definition 2–3

BUSY status 3–26

C

Cancelling asynchronous requests 5–34

CANCEL-REQUESTS( ) method 5–34

CHAINED option 4–26

Client application
accessing AppServer resources 5–8
connecting an AppServer 5–9
creating a server handle 5–9
definition 2–17
deleting the server handle 5–25
disconnecting an AppServer 5–24
executing remote procedures 5–14

Client application session
definition 2–6

Client applications
administration 3–4

CLIENT-CONNECTION-ID attribute 5–13

Code pages
client and AppServer 3–28
run-time character conversions 3–29
settings 3–28

Command-line utilities
summary 3–31

Compiled and precompiled code 6–11

Component distribution 2–19

Configuration procedures
Activate and Deactivate 4–9
Connect and Disconnect 4–7
overview 4–5
Startup and Shutdown 4–5

Configurations 2–10
AppServers connected to AppServers 2–14
clients connected to many AppServers 2–12
many clients connected to one AppServer 2–10

Configuring
AppServers
in ubroker.properties file 3–7, 3–13
using Progress Explorer 3–10
NameServers
using Progress Explorer 3–10

Connect procedure
overview 4–7
performance 6–9
usage requirements 4–8

CONNECT( ) method
application arguments 5–12
client connections 5–10
Connect procedure 4–8
connection argument 5–10
example 5–12
performance 6–10

Connecting to an AppServer 5–9

Connection ID
accessing
on AppServer 4–4
on AppServer client 5–13
definition 2–9

Connection process
application services 2–23
operating modes compared 2–26
operation 2–24
overview 2–23

Connection-based model
components 2–8
overview 2–8

Connection-level fault tolerance, definition 2–18

Context management
stateless operating mode 4–15

Controlling NameServer
AppServer setting 3–11
definition 2–18

CREATE SERVER statement 5–9

Customizing the Application Server executable 3–18

D

Database access
security 6–27

Deactivate procedure
overview 4–10
usage requirements 4–11

Debugger enabling
AppServer setting 3–13

Debugger window 7–4

debuggerEnabled property 7–7

Debugging
asynchronous requests 7–3
Debugger window 7–4
Distributed debugging 7–2
general requirements 7–7
Remote debugging 7–3
restrictions
distributed debugging 7–5
See also Distributed debugging
See also Remote debugging

Default service
AppServer setting 3–11
definition 2–24

DEFAULT-COMMIT attribute 4–26

DELETE OBJECT statement 4–19, 5–20, 5–25

DELETE PROCEDURE statement 4–19, 5–20

Deleting
asynchronous request handles 5–36
remote persistent procedures 5–20
asynchronous requests 5–35
server handles 5–25
asynchronous requests 5–35

Deployment
minimizing the effects of schema changes 6–21
on a network 6–23
server configurations 6–23
using portable r-code 6–21

Design and implementation
deployment 6–21 to 6–23
designing your distributed application model 6–33 to 6–38
security 6–24 to 6–32

Designing your distributed application model
migrating your application model 6–38
scope of persistent procedures 6–33

Disconnect procedure
overview 4–8
usage requirements 4–8

DISCONNECT( ) method
definition 5–24
performance 6–10

DISPLAY environment variable 7–6

Distributed debugging
additional considerations 7–4
definition 7–2
general requirements 7–7
restrictions 7–5
using 7–3

Distributed sessions
persistent procedure scope 6–33
procedure handle effects 6–33 to 6–35
processing boundary 2–6
schema triggers 6–37

E

ENDKEY condition 6–36
on AppServer 4–20
unhandled 4–20

Entry points to AppServer 4–11
See also Export list

Environment variables
DISPLAY 7–6
NameServer setting 3–13
PDDISPLAY 7–6
where to set 3–17

ERROR condition 6–36
on AppServer 4–19
unhandled 4–20

Error handling
AppServer session 4–19 to 4–21
ENDKEY condition 4–20
ERROR condition 4–19
QUIT condition 4–20
STOP condition 4–20
unhandled conditions 4–20
asynchronous remote procedures 5–31
compared to synchronous 5–27
client application 5–26
distributed session interactions 6–35 to 6–37
4GL client, STOP condition 5–26
remote persistent procedures 5–26
synchronous remote procedures 5–26

Error messages
displaying descriptions xxi

Event procedures
asynchronous request results 5–30
defining 5–30

Event queue 2–35

Executing
asynchronous requests 5–28
synchronous requests 5–14

Export list
definition 4–11
management 4–12
resetting 4–14
resolving remote procedure names 4–14
run-time operation 4–12
setting 4–13

EXPORT( ) method 4–11
calling 4–12
security for executing procedures 6–25

F

Fault-tolerant AppServers
overview 2–20

Fault-tolerant NameServers
overview 2–19

4GL elements
AppServer programming 4–2
4GL client programming 5–2

Files
log 3–18
ubroker.properties 3–7

FIRST-PROCEDURE attribute 5–19

FUNCTION statement
specifying super procedure prototypes 4–30

G

General debugging requirements 7–7

H

-H connection parameter 5–10

Handles
asynchronous request object 5–2, 5–28
server object 5–2
SESSION 4–2, 5–4, 5–15

Help
xxi

I

Implementation and design
deployment 6–21 to 6–23
designing your distributed application model 6–33 to 6–37
security 6–24 to 6–32

IN option, running remote internal procedures 5–16

IN SUPER option 4–30

Initial servers to start 3–26

INITIATE( ) method 7–7

Internationalization
code-page management 3–28
multi-language support 6–32

Italic typeface
as typographical convention xvi

J

Java clients 2–2

K

Keystrokes xvi

L

LAST-PROCEDURE attribute 5–19

LIMBO status 3–26

Load balancing
  overview 2–20

local procedure call
  definition 4–18

LOCKED status 3–26

Log files 3–18

Logging properties
  Application Broker setting 3–11
  Application Server process setting 3–12

loggingLevel property 7–8

Logical three-tier model 1–3
  definition 1–2

M

Managing asynchronous requests 5–27

Manual
  syntax notation xvii

Manual, organization of xiv

Maximum servers kept running 3–26

Messages
  displaying descriptions xxi

Methods
  CANCEL-REQUESTS() 5–34
  CONNECT( ) 4–8, 5–10
  DISCONNECT( ) 5–24
  EXPORT( ) 4–11
  INITIATE( ) 7–7
  SET-BREAK( ) 7–7
  SET-COMMIT( ) 4–25
  SET-ROLLBACK( ) 4–25

Migrating your current application model 6–38

Minimum servers kept running 3–26

Mixing synchronous and asynchronous requests 5–36

Monospaced typeface
  as typographical convention xvi

Multi-language support 6–32

N

NameServer
  administration 3–6
  configuring
    using Progress Explorer 3–10
  connection function 2–9
  controlling, overview 2–18
  fault tolerance 2–19
  neighbors, overview 2–19
  overview 2–4, 2–18
  replication, overview 2–19

Neighbor NameServers
  overview 2–19

Network deployment 6–23

NO-LOCK option 6–18

Normal 4GL transactions 4–21
  implementing 4–22

NSCONFIG utility 3–8

NSMAN utility 3–5

NT requirements 3–9

N-tier model
  definition 1–2
  logical three-tier model 1–3

O

ON SERVER option 5–14

Open Clients
  AppServer access 2–3
  programming the AppServer to use 4–29
  separation of context 2–6
  types supported 2–2

Operating Mode
  AppServer setting 3–11

Operating modes
  checking from AppServer 4GL 2–27
  choosing for configuration 6–3
  overview 2–21
  State-aware 2–28
  Stateless 2–30
  State-reset 2–27
  trade-offs between 6–4
    application characteristics 6–3
    considering future requirements 6–7
    handling large context 6–5
    handling small context 6–6
    limiting bound connections 6–6
    with ample computer resources 6–7
  understanding 2–27

Operating system security 6–32

Options
  APPEND 6–12
  ASYNCHRONOUS 5–17
  CHAINED 4–26
  IN (remote internal procedures) 5–16
  IN SUPER 4–30
  NO-LOCK 6–18
  ON SERVER 5–14
  PERSISTENT SET 5–15
  TRANSACTION DISTINCT 5–14

Owner Info
  AppServer setting 3–11

P

PDDISPLAY environment variable 7–6

Performance
  choosing a transaction model 6–8
  choosing an operating mode 6–3
  compiled and precompiled code 6–11
  connect and disconnect 6–9
  CONNECT( ) method 6–10
  DISCONNECT( ) method 6–10
  load balancing 6–8
  minimizing network traffic 6–2
  offloading resource-intensive processing tasks 6–3
  Progress browse widget 6–11
  schema transmission 6–12

PERSISTENT attribute
  distributed sessions 6–33

PERSISTENT SET option 5–15

-pf connection parameter 5–10

Physical n-tier model 1–3

Port Number
  AppServer setting 3–11

Portable r-code, advantages 6–21

PROBUILD utility 3–18

Procedure handles, *See* Proxy persistent procedure handles, Remote procedure handles

PROCEDURE statement
  specifying super procedure prototypes 4–30

PROCEDURE-COMPLETE event 5–29

prodebug executable
  distributed debugging 7–4

Progress AppServer 3–1
administration framework 3–2
basic components 2–4
code-page management 3–28
component distribution 2–16
configuration procedures 4–5
configurations 2–10
configuring
using Progress Explorer 3–10
configuring the components 3–8
connection parameters 5–10
connection process, overview 2–23
connection-based model 2–8
controlling entry points 4–11
debugging 7–2
See also Remote debugging 7–2
fault-tolerant instances 2–20
fault-tolerant NameServers 2–19
intelligent data interface 2–3
load balancing, overview 2–20
maintenance benefits 2–3
multi-language support 6–32
Open Client access 2–3
operating modes
choosing for configuration 6–3
overview 2–21
overview 2–2
performance considerations
primary 6–2
secondary 6–9
programming for Open Clients 4–29
purpose 2–2
record management 6–18
registration overview 2–22
returning values 4–19
running without a network A–1
run-time components 2–16
security benefits 2–3
session startup parameters 2–22
starting and managing 3–19
startup and shutdown, overview 2–22
stateless context management 4–15
status checking 3–26
transaction management 4–21

Progress browse widget
performance 6–11

Progress Explorer
configuring
AppServers 3–10
NameServers 3–10
configuring an AppServer instance 3–9
overview 3–4
starting and managing AppServers 3–20

Progress startup parameters 2–22

PROPATH environment variable
export list effects 4–11
overriding 3–12
remote procedure name resolution 4–14

Properties file 3–7, 3–13
editing and validating 3–13
guidelines for editing 3–14
hierarchy 3–14

ProtoGen utility 4–29

PROXY attribute
distributed sessions 6–33

Proxy persistent procedure handles
accessing 5–19
definition 4–18, 5–19
local persistent procedure handles
compared 5–20

Q

QUIT condition 6–37
on AppServer 4–20
unhandled 4–20

R

R-code for SmartDataObjects 3–18

Record management
buffer currency conflicts 6–18

REMOTE attribute
distributed sessions 6–33

Remote debugging
  definition 7–3
  general requirements 7–7
  initial breakpoint 7–7
  initiating 7–6
  using 7–6

Remote persistent procedure handles
  definition 4–18, 5–19

Remote persistent procedures
  definition 2–4
  deleting
    from a 4GL client 5–20
    from the AppServer session 4–19
  error handling 5–26
  stateless bound connections 2–31

Remote procedure handles 4–18

Remote procedures
  asynchronous 5–14
  definition 2–2
  examples 5–20
    non-persistent 5–21
    persistent, calling a user-defined function 5–23
    persistent, calling an internal procedure 5–22
  general behavior 5–18
  how to execute 2–4
  PROXY and REMOTE attributes 6–33
  running 5–14
  synchronous 5–14

Replicated NameServers, overview 2–19

Request types
  comparing 2–32
  operating modes 2–34
  process flow 2–33
  specifying 2–34

Requests 2–2

Requirements, *See* System requirements

-rereadnolock startup parameter 6–20

Resolving remote procedure names 4–14

Response queue 2–34

RETURN ERROR statement
  Activate procedure 4–10
  Connect procedure 4–8

RETURN-VALUE function
  across sessions 5–27

RUN Statement
  handling conditions 5–26

RUN statement 5–14
  PROXY, REMOTE, and PERSISTENT attribute effects 6–34

RUN statement (remote)
  asynchronous execution 5–14, 5–17
  handling conditions 5–26
  run-time parameters 5–15
  synchronous execution 5–14

Running remote internal procedures 5–15

Running remote procedures 5–14
  persistently 5–15
  PROXY and REMOTE attributes 6–33

Run-time compilation
  security 6–31

S

-S connection parameter 5–10

Schema changes 6–21

Schema transmission 6–12

Schema triggers
  distributed session effects 6–37

Scope of persistent procedures 6–33

Security
audit trails 6–30
AppServer application 6–30
client application 6–31
authentication and authorization 6–25
stateless AppServers 6–26
state-reset and state-aware AppServers 6–26
database access 6–27
export lists for AppServer access 6–27
operating mode interactions 6–28
operating system 6–32
options 6–24
run-time compiled code 6–31

Send queue 2–34

Separate sessions 2–5
processing boundary 2–6

Separation of context 2–6
Open clients and AppServer 2–6

Server Control Parameters
AppServer setting 3–13

Server Executable File
AppServer setting 3–12

Server handles 5–2, 5–7

Server Parameters
AppServer setting 3–12

Server Pool Parameters
AppServer setting 3–13
choosing the settings 3–26
properties affected 6–7

Server Port Range
AppServer setting 3–12

SERVER-CONNECTION-BOUND attribute
managing stateless context 4–18

SERVER-CONNECTION-BOUND-REQUEST attribute
managing stateless context 4–16
overview 2–31

SERVER-CONNECTION-CONTEXT attribute
Activate procedure 4–9
Connect procedure 4–7
Deactivate procedure 4–10
managing stateless context 4–15
overview 2–32

SERVER-CONNECTION-ID attribute 4–4
Activate procedure 4–9
Connect procedure 4–7
Deactivate procedure 4–10
managing stateless context 4–16

SERVER-OPERATING-MODE attribute 2–27

SESSION system handle
AppServer session 4–2
client session 5–7
EXPORT() method 4–2
local execution of asynchronous requests 5–37
local execution of remote procedures 5–15
SERVER-CONNECTION-BOUND attribute, managing stateless connection context 4–18
SERVER-CONNECTION-BOUND-REQUEST attribute, managing stateless connection context 4–17
SERVER-CONNECTION-CONTEXT attribute 4–7, 4–9, 4–10
managing stateless connection context 4–15
SERVER-CONNECTION-ID attribute 4–4
managing stateless connection context 4–16

SET-BREAK( ) method 7–7

SET-COMMIT( ) method 4–25

SET-ROLLBACK( ) method 4–25

Shutdown procedure
overview 4–6
usage requirements 4–6

SmartDataObject support 3–18

Starting an AppServer
  NSMAN command-line utility 3–22
  Progress Explorer 3–20

STARTING status 3–26

Startup procedure
  overview 4–5
  usage requirements 4–6

State-aware operating mode
  authentication and authorization 6–26
  definition and operation 2–28
  design trade-offs 6–4
  export list interactions 6–29
  overview 2–21

Stateless operating mode
  authentication and authorization 6–26
  bound and unbound connections 2–30
  connection management
    making bound 4–17
    making unbound 4–17
  definition and operation 2–30
  design trade-offs 6–4
  export list interactions 6–29
  managing connection context 4–15
  overview 2–22

Statements
  CREATE SERVER 5–9
  DELETE OBJECT 4–19, 5–20, 5–25
  DELETE PROCEDURE 4–19, 5–20
  FUNCTION 4–30
  PROCEDURE 4–30
  RETURN ERROR 4–8, 4–10
  RUN 5–26
  RUN (remote) 5–14
  TRANSACTION-MODE 4–23

State-reset operating mode
  authentication and authorization 6–26
  definition and operation 2–27
  design trade-offs 6–4
  export list interactions 6–28
  overview 2–21

Status, AppServer run-time 3–26

STOP condition 6–37
  on 4GL client 5–26
  on AppServer 4–20
    unhandled 4–20

Super procedure prototypes 4–29

Synchronous and asynchronous requests 2–32
  examples 5–38
  process control flow 2–33

Synchronous remote procedures
  error handling 5–26

Synchronous requests
  definition 2–3
  4GL execution model 5–38

Syntax
  ASBMAN utility 3–23
  ASCONFIG utility 3–16

Syntax notation xvii

System requirements
  NT and Windows 3–9
  UNIX 3–9

T

THIS-PROCEDURE system handle
  distributed sessions 6–33

Top level execution 5–18

TRANSACTION attribute 4–23

TRANSACTION DISTINCT option 5–14

Transaction initiating procedure
  definition 4–23

Transaction object
  definition 4–23
  initializing 4–23

Transaction object handle 4–23

TRANSACTION-MODE statement 4–23

Transactions
buffer currency conflicts 6–18
choosing a model 6–8
comparing normal 4GL and automatic 6–13
distributed session conflicts 6–18
distributed session consistency 6–15
implementing
automatic 4–23
normal 4GL 4–22
maintaining small size 6–14
managing 4–21
returning status to client 6–14
types 4–21
types, *See also* Automatic transactions
types, *See also* Normal 4GL transactions

TRANS-INIT-PROCEDURE attribute 4–26

Two-tier model 1–3

Typographical conventions xvi

U

ubroker.properties file 3–7
hierarchy 3–15
See also Properties file

Unbound connections 2–30
setting 4–17

UNIX requirements 3–9

W

Windows requirements 3–9

Working directory
AppServer setting 3–11
overview 3–18