



NetWeave Integrated Solutions, Inc.

# NetWeave Programmer's Guide

---

*User's Guide for Version 2.0 January 2008*



[www.netweave.com](http://www.netweave.com)

Copyright © 2002-2008 NetWeave Integrated Solutions, Inc.. All rights reserved.

Netweave is a registered trademark of Netweave Integrated Solutions, Inc.

Windows is a registered trademark of Microsoft Corporation.

CICS, MVS, and MQSeries are registered trademarks of the IBM Corporation.

UNIX is a registered trademark of The Open Group.

Tandem, Guardian, VMS, and OpenVMS are registered trademarks of Hewlett-Packard.

All other trademarks are noted in the text and are the property of their respective owners

# Table of Contents

<b>INTRODUCTION TO PROGRAMMING NETWEAVE .....</b>	<b>1</b>
What is NetWeave?.....	1
The NetWeave Documentation Suite.....	2
What's In This Manual .....	3
<b>TOPICS FOR DESIGNERS .....</b>	<b>4</b>
Choosing the Middleware That's Right for You.....	4
The Messaging Services.....	4
Synchronous Messaging.....	4
Asynchronous (Queued) Messaging .....	5
Broadcast Services .....	6
Data Server.....	6
How Do You Decide? .....	7
The NetWeave Agent.....	8
Multiplexing Connections On a Single Channel (Message Concentration).....	9
Using the NetWeave Agent for Message Routing .....	9
Protocol Conversion.....	10
System Security.....	10
Transaction Protection .....	11
Using the Agent to Connect to Legacy Applications.....	12
IBM/CICS .....	12
Providing Access to Pathway Serverclasses in a Tandem Environment.....	14
Tandem Pathsend Protocol.....	14
Synchronous (IPC) Messaging.....	15
Client-server.....	15
Peer-to-Peer Messaging .....	16
Sleep: the Key to a Peer Process .....	16
Exiting from Sleep Forever .....	18
Making Connections in Asynchronous Mode.....	19
Internal Queues for Asynchronous Messages .....	20
Asynchronous (Queued) Message Services.....	21
How a Queue Works .....	21
Queue Location .....	22
Queued Messaging Functions .....	23
Designs with Multiple Queues .....	24
Sizing a Queue .....	26
Expedited Messages.....	26
Broadcast Services .....	27

Data Server Services .....	30
Flat Files.....	30
Record-Oriented Files .....	31
File Access Resources in the NetWeave Header File (netweave.h) .....	31
Asynchronous Notification of File Changes .....	32
Using NetWeave's SQL Interface to Access Relational Databases.....	33
Tandem NonStop SQL.....	33
System Security .....	35
Data Translation Services .....	37
Defining Data to NetWeave.....	37
Message Translation .....	38
Record Translation.....	38
Threaded Service .....	39
<b>TOPICS FOR PROGRAMMERS .....</b>	<b>41</b>
The NetWeave Header File.....	41
Enumerations and Constants.....	42
Typedefs.....	43
Item Lists .....	44
Programming Tips: Synchronous Calls .....	47
Example #1: Synchronous Calls in a Client-Server Environment.....	47
Example #2: Infrastructure.....	49
Example #3: A Simple Distributor for Queued Messages .....	51
Programming Tips: Asynchronous Calls .....	56
Context.....	56
Waiting for Events to Happen.....	57
Associating Events with Callbacks .....	57
TANDEM .....	58
UNIX.....	59
Digital Equipment Corp., VMS and OpenVMS .....	59
Microsoft Windows NT, Windows Millennium .....	59
Asynchronous Server: Case Study.....	60
Road Map for ASERVER.....	60
Prototyping with Synchronous Calls.....	70
Mixing Synchronous and Asynchronous Calls.....	70
Calls with Timeouts .....	70
Tips on Testing NetWeave Applications .....	71
Threaded Dispatcher dp_pong: Case Study.....	72
The Windows Versions of the NetWeave DLL.....	76
W95.....	76

NT/2000 .....	76
Contents of the NetWeave for Windows 95 and NT Releases .....	76
<b>SAMPLE NETWEAVE PROGRAMS .....</b>	<b>78</b>
Accessing the NetWeave ftp Site.....	78
Code Samples.....	79
General Purpose Examples .....	79
WinNT .....	80
NTPongSamp.zip .....	80
NTWindemo.zip.....	81
UNIX.....	82
Ibm_cics\cobol .....	83
Win31.....	83
POWERBUILDER .....	83
VisualBasic (vb30).....	83
TANDEM .....	84
TANDEM\cfile\T1.ZIP Data server.....	84
TANDEM\fifo\T2.ZIP Queued Mesages .....	85
TANDEM\ipc\T3.ZIP IPC Messaging.....	86
TANDEM\COBOL85\T4.ZIP Flat Files and Kernel Functions .....	87
TANDEM\CREV\T9.ZIP .....	88
TANDEM\CREV\T10.ZIP .....	88
UNISYS .....	88
<b>GLOSSARY .....</b>	<b>89</b>



# Introduction to Programming NetWeave

## What is NetWeave?

NetWeave is a software-based middleware product that allows disparate computing systems to interoperate. NetWeave supports a wide variety of computing platforms including those from IBM, Compaq/Digital, Unisys, Compaq/Tandem, Stratus and large UNIX systems, as well as all PC, UNIX and Macintosh workstations. To allow applications on these platforms to communicate with applications on other platforms, NetWeave offers the following services:

- Messaging services (including transactional, queued, and broadcast styles)
- Remote data access and update
- File transfer capabilities
- Data conversion
- Transaction management
- Authentication, identification, and encryption services

The NetWeave Product Suite includes the following components:

Component	Description
NetWeave Distributed Services Product (NWDS)	The application library and a NetWeave Agent process that are the heart of the NetWeave product.
The Reliable File Transfer Utility Version 2	A utility for reliable, restartable transfer of binary and text files between NWDS-supported hardware platforms. Available as both an API and a command line program.
NetWeave Version 2 (NW2) application (i.e., old NetWeave)	An older, non-wire-level compatible version of the NetWeave product. Although NetWeave still supports existing installations of NW2, all new implementations use NWDS.
Tandem Print Process	A Despooler process that takes files spooled to the Tandem Spooler and delivers them to other platforms as files.
NetWeave COM+ Interface (for Windows 9x/NT/2000 platforms)	An interface used in integrated environments that want to access the Microsoft COM environment.
NetWeave sample programs	Sample code that demonstrates how to use the product in a variety of supported environments. One of the sample programs (PerfPong) measures performance on the network.
Interactive Test program (TST)	A simple command line-driven program that lets you use the NetWeave API interactively.

## The NetWeave Documentation Suite

The table below lists the documentation for the NetWeave product. You can download these documents from the NetWeave ftp site by connecting to <ftp://www.netweave.com/middleware/doc/200/MSWord> (for Microsoft Word format) or <ftp://www.netweave.com/middleware/doc/200/PDF> (for PDF format).

Document	Description	Audience
Configuration Guide	Explains how to install and configure a NetWeave system for a customer's particular environment.	Systems programmers who maintain the communications layer on which NetWeave rests. Operations personnel who start and shut down NetWeave processes in a distributed environment. Designers and managers who configure the applications that use NetWeave.
Programmer's Guide (this document)	Explains how to use NetWeave functions to meet the requirements of distributed systems. Illustrates working sample programs.	Analysts who design and programmers who build applications in a distributed computing environment.
Applications Programming Interface Guide	Explains how to use the NetWeave API function calls for client-transaction applications, messaging services, and data server applications.	Programmers who create and maintain NetWeave applications on any system.
IBM/CICS Configuration Supplement	Explains and illustrates features of NetWeave that are unique to the IBM MVS/CICS environment.	System managers and analysts who must install NetWeave in the IBM MVS/CICS environment.
NetWeave Print Process for Tandem	Describes how to install and use this printer driver to transfer spooled output from a Tandem system to any computer on which NetWeave is implemented.	Tandem system managers.
Enhancements	Lists enhancements and significant bug fixes by release version.	Project managers and programmers who plan, build and maintain NetWeave applications.
Performance Tester	Explains how to use the Performance Test program (distributed as a sample program with NetWeave) to simulate and measure various NetWeave application configurations.	Project managers and programmers who plan, build and maintain NetWeave applications.
TST Guide	Explains how to use the interactive testing tool called TST.	Project Managers and novice programmers.
NetWeave COM+ Interface Document	Explains how to install and use the NWCOM interface.	Microsoft Visual Basic and Visual C++ Programmers.



## What's In This Manual

This manual consists of three main sections:

- Topics for Designers is a guide for analysts and designers who need to understand how to use NetWeave to meet the requirements of their distributed systems.
- Topics for Programmers is a guide for programmers who must translate the design requirements into working programs.
- Guide to Programs explains how to use the working sample programs on the NetWeave ftp site. Both designers and programmers who want to build applications on distributed systems need to study these programs and understand how they work.





## Topics for Designers

This section discusses design considerations for building applications with the NetWeave API. It assumes that you are familiar with middleware, have some experience with program design for a distributed, heterogeneous environment, and understand how the tradeoffs inherent in the various middleware tools can affect design decisions.

When designing a system, you must make the decisions about which tools to use and how best to use them at several levels. At the highest level, you analyze the application problem to determine which middleware services are the most appropriate. Your initial high-level choices then narrow the range of choices available at the lower levels. Some common questions that need to be addressed early on in the design process:

- Is this a messaging application or a database application?
- Will any existing applications be modified to use the NetWeave API?
- How much new development will there be? On which platforms?
- Is one type of NetWeave service enough, or will a combination serve you better?

The information in this section will help application designers and integrators make some of the early high-level decisions required for a successful implementation.

## Choosing the Middleware That's Right for You

There are two types of middleware tools:

- Messaging services
- Data server services

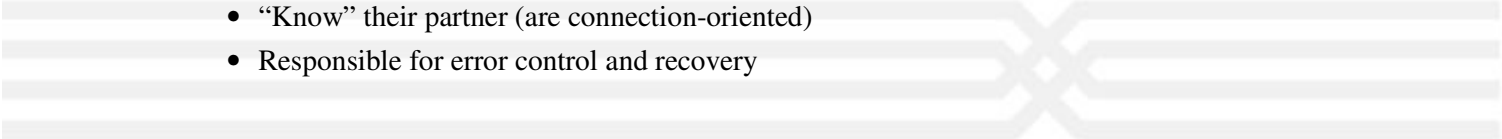
### The Messaging Services


Messaging middleware includes synchronous messaging (IPC messaging services such as client-server and peer-to-peer), asynchronous (queued) messaging, and broadcast services.

With messaging middleware, the server application shields from client applications on other systems the details of how a local system's files and databases are organized and accessed. Therefore, the design team has to define messages that contain the specific information that the message recipient needs to do its job.

### Synchronous Messaging

Synchronous messaging is used for applications where programs communicate and synchronize operations by exchanging messages, such as on-line transaction processing or high speed, real-time process control applications that have the following characteristics:

- Highly interactive
  - "Know" their partner (are connection-oriented)
  - Responsible for error control and recovery
- 



In synchronous messaging, the client blocks until it gets a response from the server that a message has been successfully received. Therefore, application designers must define contingencies for problems either on the network or on the remote (partner) computer. In the real world, there is very little difference between a network problem and a problem with a remote process, and both can usually be treated the same way by both client and server applications.

NetWeave uses the term *IPC messaging* to refer to the general exchange of messages between peer processes. The two models of interprocess communication are client-server and peer-to-peer.

The client-server model is the most common form of IPC messaging. The client application sends a request message to the server program, which retrieves information or updates a local database on behalf of the (remote) client application. Client-server designs are commonly associated with on-line transaction processing (OLTP) where there is substantial interaction among processes and where high throughput is essential.

For many businesses, NetWeave offers an extremely effective means to preserve the substantial investment in existing legacy business and commercial applications. These legacy applications can be enhanced to perform as NetWeave application servers, often with few (if any) changes to the existing code.

Client-server designs must address the following concerns:

- How will system security be compromised by the addition of remote clients?
- How do we make multiple, coordinated changes to applications on the server machine?
- How do legacy applications communicate with the new client applications?
- How should error checking change to reflect the fact that a considerable amount of user input validation now occurs on the workstation machine?

For more information about IPC messaging, see page 15.

## Asynchronous (Queued) Messaging

Queued message services, which are always asynchronous, are used to connect autonomous applications. Because queued messaging is connectionless, the interaction between applications consists of a very simple unidirectional flow of information that does not require acknowledgment. The middleware assures *reliable message delivery* and *error recovery*.

Queued message services have much lower throughput statistics than do client-transaction services. Because of the slower throughput, most queued messaging services have a scheme for prioritizing messages: high priority messages have their own queue, and the message delivery software gives preference to this queue.

Design issues for asynchronous messaging:

- Where should the queue be located, and how large should it be?
- Should there be only one consumer per queue?
- What kind of performance can be expected?
- How can a particular message be expedited?

For more information about queued messaging, see page 21.



## Broadcast Services

Broadcast services are very effective in a distributed environment where you don't know or care how many entities will be receiving the information. Just as a TV network broadcasts the news without worrying about which TV set actually receives the signal, an application that generates a message (sender) does not need to know anything about a particular application that receives the information (receiver). A sender may continue to generate broadcasts even in the absence of receivers.

To use NetWeave's broadcast services, senders and receivers must be connected via an IP backbone. Broadcast services are implemented on top of *UDP datagram* services. Like TCP/IP, UDP belongs to the IP family of protocols. Datagrams are ideal for broadcasts because they are delivered to the IP network layer without regard to how many nodes in the network may consume the information: one sender, potentially limitless readers. This is why broadcast services scale far better than any other form of middleware. Where the broadcast audience is PCs and workstations (variable numbers in service at any one time and/or over time), the scaling of message delivery is often the hardest requirement to satisfy for a distributed application.

By design, UDP datagram services are connectionless. They are also "unreliable," in the sense that there is no guarantee that any given datagram will be delivered to a particular node in the IP network. To address this, the NetWeave broadcast API provides a reliable, scalable, high-speed message delivery mechanism to offset the inherent unreliability of the datagram service.

For more information about broadcast services, see page 27.

## Data Server

Data server services allow all computers in the network to access a particular computer platform's file system. Typically, an application on one computer reads and updates the files and records in another computer's file system to provide a single, standardized repository of corporate data.

Data server services are very similar to the familiar functions that perform disk I/O. These services make I/O on a remote system as easy and intuitive for the programmer as file I/O is on the local system. Whether the target is a record structure in a file system or a row in a table of a relational database, the remote application performs I/O operations without regard to the location of the file.

NetWeave's client-data base API supports access to SQL databases and record-oriented file systems on host computers. These functions may be integrated with NetWeave's authentication functions to control access to datasets and records in files. For more information about data server services, see page 30.

## How Do You Decide?

How do you decide which service to use to interconnect and coordinate applications in a distributed computing environment? Some rules of thumb for middleware services:

Service	Description
Synchronous messaging	Services that use synchronous messaging are tailored for OLTP, and have better throughput than queued (asynchronous) messaging. However, with synchronous messaging, the application must assume responsibility for error detection and recovery. Synchronous messaging is a good choice if applications on the host platform contain complicated business logic and/or maintain the relational integrity of the corporate database.
Asynchronous messaging	Queued (asynchronous) messages are best for workflow among autonomous applications. Use asynchronous messaging services for expedited messages and set up a queue per consumer to simplify recovery. This service is a good choice if applications on the host platform contain complicated business logic and/or maintain the relational integrity of the corporate database.
Broadcast	Broadcast is an ideal mechanism when you need to distribute information to a large number of users, the number of recipients is unknown, and acknowledgment is irrelevant. Broadcast works well if message rates can be kept relatively constant. If your application will have bursts of messages, consider queuing them on the sender's side and providing a Distributor process that performs the actual broadcast.
Data server	Data server services can always be implemented where the file and record structures are known. If no development will be done on a particular platform (the application is third-party, proprietary code that you can't access, or business decisions dictate that no more development be done on a particular machine), data server is often your only choice.

**NOTE:** Because NetWeave's services are not mutually exclusive, you should consider using combinations of services to design your system. Also, depending on the platforms in your network (but independent of whether you use messages or database services), you will need to consider security and data translation needs.



## The NetWeave Agent

NetWeave consists of a NetWeave Agent and a library that is linked with various applications. The Agent is a special process that is installed on each platform where NetWeave services are required. Typically the Agent is treated as an extension of the operating system, and is started as part of the boot procedure. Sometimes, for performance reasons and/or to segregate applications, there may be more than one NetWeave Agent operating on a given machine.

This section describes the role of the Agent in client-server designs for the following tasks:

- Message concentration
- Message routing
- Protocol conversion
- Security
- Transaction protection
- Connecting to legacy applications

Each NetWeave application is linked to the NetWeave library. This library, which conforms to the standards of the hardware platform on which it resides, is the interface between the user's application and the foreign platform. For example, on a Tandem computer the library is statically linked with the application, while on a PC the library is usually a dynamically linked library (DLL). In client-server applications, the library performs all of the lower-level operations required to form connections, transfer messages, and clean up disconnections.

To form a new connection, several layers within the NetWeave software must perform initialization functions. When a client application makes a TCP/IP connection with a server application, the NetWeave library exchanges several messages with its partner on the server platform in a process called negotiation. Because creating a connection requires several steps – each of which consumes system resources – one objective of client-server application design is to minimize the number of times connections are created and destroyed.

The NetWeave Agent mediates communication between a client application on one platform and a server application on another in the following situations:

Situation	Role of the Agent
Protocol translation	When sender and receiver do not use the same network protocol, a NetWeave Agent can perform the protocol switch.
Link consolidation	If many client applications want to establish messaging sessions with one or more applications on a remote machine, using a NetWeave Agent on one or both platforms reduces the effective number of network channels between the platforms.
Routing isolation	If they communicate through a NetWeave Agent, client applications and their configuration (INI) files do not need to know the details of a server's actual location and connection requirements. They simply have to know how to talk to the Agent, and they must have a name that they can use to refer to the destination application. The NetWeave Agent takes care of the details of how to connect to the application server.

Figure 1 illustrates two client-server models, one that uses an Agent, and one that doesn't:

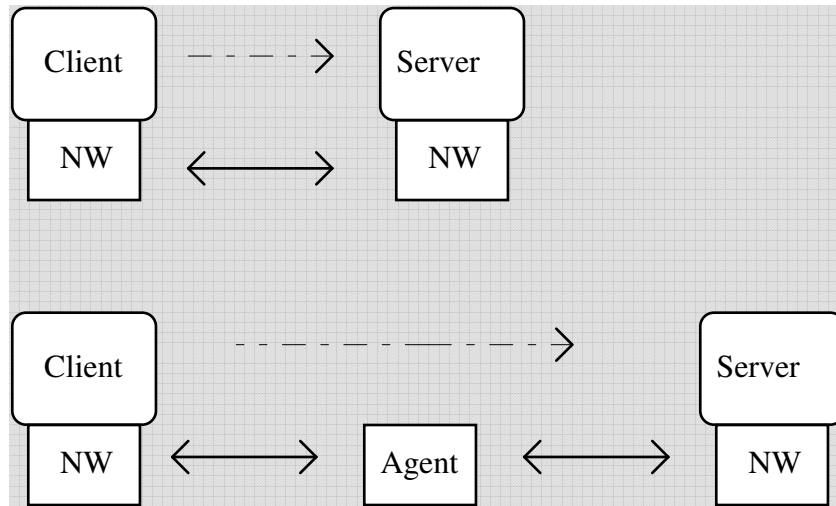


Figure 1. IPC messaging with and without a NetWeave Agent

**NOTE:** Although the Agent software is optimized for throughput, a design that does not use the Agent at all will give you better performance, simply because eliminating a process hop and a context switch often speeds things up. However, because link consolidation (described on the previous page) allows you to bunch traffic from different sessions into a single network transfer, this consolidation often yields better performance.

### Multiplexing Connections On a Single Channel (Message Concentration)

Some transport media (TCP/IP, SPX/IPX, and LAN protocols such as Netbios) allow unlimited connections, while others (APPC and packet switched networks) do not. Where the underlying transport media limit the number of connections, you can install an Agent on each platform, and all connections with the remote platform are then placed through the local Agent. The Agents on the two platforms can multiplex many connections and conversations between clients and servers through a single transport channel.

### Using the NetWeave Agent for Message Routing

NetWeave message routing is analogous to a hardware router in the physical network. A hardware router moves packets through the network, deciding which path a packet should take when several routes are available. The nodes at the end points (the sender and receiver of the message) don't know or care which path a given packet travels.

To increase service reliability and redundancy while minimizing NetWeave INI file maintenance, you can use the NetWeave Agent to perform routing functions at the application level. In Figure 2, a client application connects to a server via an Agent. Sometimes the server runs on platform A; other times it runs on B. To use Agent routing, you don't have to change the client or its INI file: just change the

Agent's INI file to point to the active server. The Agent may run on any of the three platforms, or even a fourth!

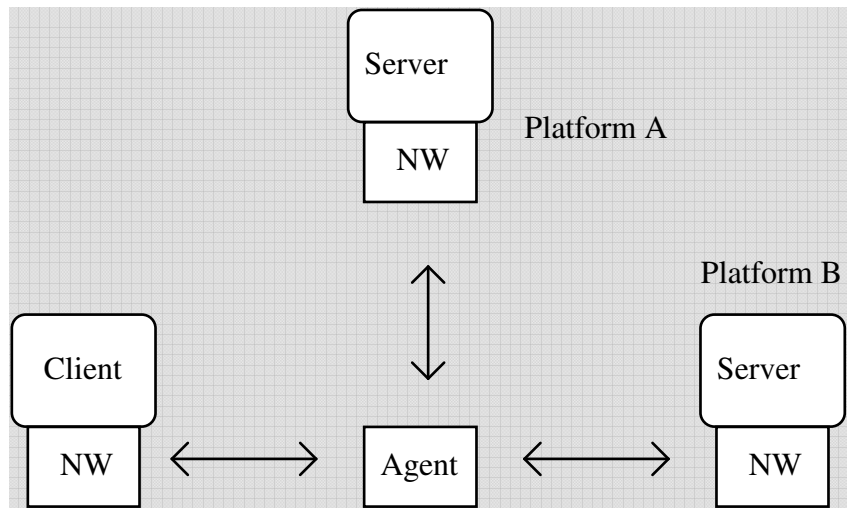


Figure 2. NetWeave Agent routing capabilities

## Protocol Conversion

For the NetWeave Agent, protocol conversion means receiving a message from a client via one communications protocol and relaying it to the server via a different one. Protocol conversion is often used when the server is a legacy application. The two most common uses of protocol conversion (connecting to IBM/CICS transactions, and connecting to Tandem Pathway serverclasses) are described in “Using the Agent to Connect to Legacy Applications” on page 12.

## System Security

A designer is responsible for the following aspects of secure communications:

- Authentication: is the client who she/he claims to be? Has the data been altered en route from the client?
- Encryption: is the data safe from prying eyes?

NetWeave provides a flexible, powerful interface for user authentication. The security functions are controlled by the Agent and are optional, in that you can choose whether to use NetWeave's security capabilities or not. For example, if your system/network exists in a closed, captive environment, you may not need any security at all.

For more information about system security, see page 35.

## Transaction Protection

A transaction protection monitor (TP) is software on a host computer that coordinates updates to the file system and databases. You can tell a TP that either all changes in a series must be executed successfully, or the program should back out to the original state that was in effect when the transaction began.

Another design objective when implementing OLTP is to construct messages that allow the server to control the transaction. All the information to be updated together is sent as a single request message. The server starts the transaction, applies the updates, instructs the TP to commit (i.e. apply the work done), and finally replies to the client

If the client, as keeper of context, must use more than one request message – either to the same or to a different, context-free server – to make coordinated updates, the client must initiate the transaction with the TP. The client must also tell the TP to either commit or roll back (abort) the transaction at the end. In addition, the NetWeave Agent on the host platform provides the interface between the client and the host's TP monitor, and the Agent (on the host) holds the transaction context on behalf of the remote client. To manage transactions, clients use the NetWeave functions that begin with `nwds_tp_`:

Function	Description
<code>nwds_tp_start</code>	Issued by the client to begin a transaction on the host (server's) computer.
<code>nwds_tp_commit</code>	Issued by the client to signal the TP to commit (apply) the work.
<code>nwds_tp_abort</code>	Issued by the client to signal the TP to abort (roll back) the work.

In Figure 3, the client's logical transaction spans more than one service request at the Agent's platform. The Agent, TP, and servers are assumed to be on a common platform. It doesn't matter whether the requests go to the same server or two different ones: for purposes of TP control, if the servers are "context free," two requests to the same server are logically indistinguishable from two requests to two different servers.



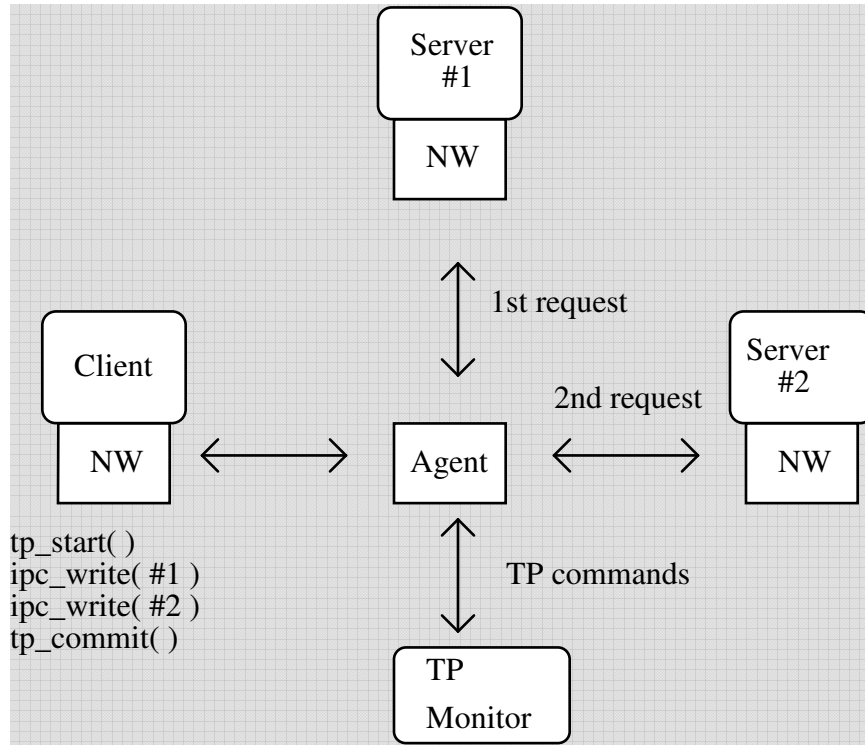


Figure 3. NetWeave transaction protection architecture

## Using the Agent to Connect to Legacy Applications

The NetWeave Agent is often used to provide connectivity to legacy applications, such as those on the IBM/CICS and Tandem Pathway platforms. Both cases represent substantial investments in installed software, and provide a natural — though proprietary — message-oriented interface to servers (called “transactions” in CICS and “serverclasses” in Pathway). The Agent converts the network protocol used by the client application on the remote platform to the proprietary interface of the host transaction monitor.

### IBM/CICS

CICS has two mechanisms for communicating with transactions (servers):

- Write a request to a transient data queue (TDQ) with an associated trigger level. A TDQ is an input queue for the transaction that buffers requests. It is not a two-way transmission mechanism for returning a reply.
- Call the CICS "execute" function for a specified transaction and pass the request message as a parameter.



The NetWeave Agent supports both mechanisms. To indicate which one to use, the application uses the item list parameter `NWDS_CICS_IMAGE_FLAG`, which can take one of two values:

Value	Description
<code>NWDS_CICS_IMAGE_TDQ_NAME</code>	Passes a message to a <i>long-running</i> transaction via its TDQ. Because a long-running transaction avoids the penalties associated with transaction startup (loading the image, opening files, etc.), you should use this method to support client-server applications for OLTP.
<code>NWDS_CICS_IMAGE_TRANS_ID</code>	Executes a specified transaction (once).

In Figure 4, a remote client sends a request message to the NetWeave Agent in the CICS region. The Agent writes the request to the transient data queue, and CICS signals the transaction (server) to read its message and process it.

**NOTE:** Because a TDQ buffer inputs requests, you can't return a reply from the server to the Agent via the TDQ (and hence back to the client). This is an important difference from the standard model of client-server where the server replies to the client when its work is completed.

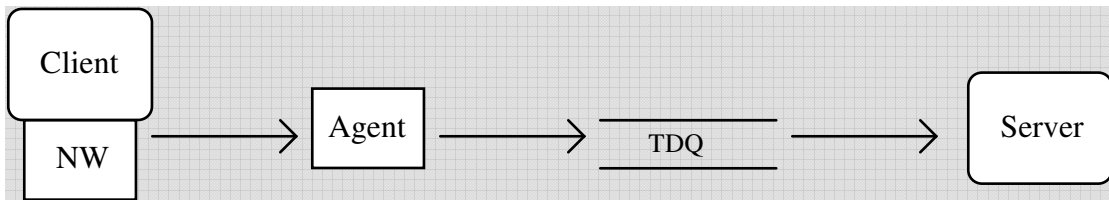


Figure 4. IBM MVS/CICS messaging environment

## Providing Access to Pathway Serverclasses in a Tandem Environment

NetWeave is installed in many situations where the client application replaces traditional Tandem requesters on PCs and workstations. The NetWeave Agent on Tandem provides access from client applications on external, remote platforms to Pathway serverclasses (servers). To form these connections to the serverclasses, NetWeave uses Tandem's Pathsend mechanism. You don't have to change or reconfigure anything in the Pathway environment to permit access by NetWeave clients.

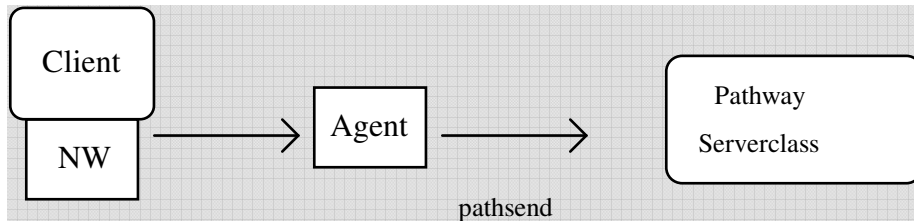


Figure 5. Tandem Pathway messaging environment

### Tandem Pathsend Protocol

A remote application may use the `nwds_ipc` API functions to talk to a Pathway serverclass. In most cases, the serverclass does not need to be NetWeave-enabled (no coding changes are required in the serverclass) to handle calls to NetWeave.

In releases before 1.05.05, NetWeave implemented its interface to Pathway serverclasses with the same syntax used by all other forms interprocess communication; that is, the `nwds_ipc` API worked the same way regardless of the underlying transport mechanism. In subsequent versions, the Pathsend changed because Tandem requires that a single-threaded serverclass be shielded from opens and closes of different linkmon processes. Because of these changes, Pathsend performance in version 2.0 is substantially better, and you can expect message rates to nearly double.

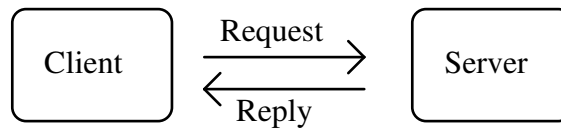
**NOTE:** If a COBOL serverclass is NetWeave-enabled, it should use the NetWeave API for IPC messaging, even if all messages will be delivered via Tandem's proprietary \$receive mechanism.

## Synchronous (IPC) Messaging

NetWeave uses the term *IPC messaging* to refer to the general exchange of messages between peer processes. The two models of interprocess communication are client-server and peer-to-peer.

### Client-server

The *client-server model* of messaging is the simplest and most common. In this model, the client application acts as the interface between the human user and the computer to receive inputs, edit them, and prompt the user where appropriate. The application uses the user's inputs to construct a message that it sends to the server program. The server program retrieves information or updates a local database on behalf of the (remote) client application. When the server's work is done, it replies to the client by sending either the requested information or a message about the status of the operations performed for the client.



**NOTE:** In client-server designs, the client application is blocked (waiting) while the server is doing its work. When you use the client-server model to implement OLTP, make the server as efficient as possible to minimize how long the client application (and the impatient human) must wait.

One way to increase server efficiency is to make the server application context-free by including in the client's request message all of the information that the server needs to perform its function (such as file positioning information). To make the client the keeper of context, the server may need to include context information in the reply. For more information about context, see page 56.

To build client-server applications, use the NetWeave functions that begin with `nwds_ipc_`. To create connections between the client (the active partner) and the server (the passive partner), use the following functions:

Function	Description
<code>nwds_ipc_publish</code>	Issued by the server to notify NetWeave that it is ready to receive new connections.
<code>nwds_ipc_connect</code>	Issued by the client to initiate the connection sequence.
<code>nwds_ipc_accept</code>	Issued by the server in response to receipt of a new connection.

Both `nwds_ipc_connect` and `nwds_ipc_accept` return special tokens (called NetWeave handles) to the client and server to represent the connection between them. Once a connection is established, the client and server use the following functions to exchange messages:

Function	Description
<code>nwds_ipc_write</code>	Used by the client to send the request and used by the server to send the reply.
<code>nwds_ipc_read</code>	Used by either to read a message from the other.

**NOTE:** In the standard client-server model, the receiver must initiate `nwds_ipc_read` before the `nwds_ipc_write` initiated by the sender can succeed.

To break an established connection, or to reject an attempted connection from an unauthorized user, use `nwds_ipc_shutdown`.

## Peer-to-Peer Messaging

Client-server technology, where an application client communicates with a single application, is one instance of the more general peer-to-peer model of message delivery. In peer-to-peer messaging, any application may send and receive messages from one or more other applications, and the “client” and “server” designation may change on a message-by-message basis. The peer process accepts and responds to any number of messages from any number of sources, either with or without acknowledgment. NetWeave uses the same IPC services for both client-server models and peer-to-peer models.

The peer-to-peer model is the most general form of IPC messaging, and has the following advantages over client-server messaging:

- Better performance, because scarce communications resources are used more effectively.
- Better scaling than with client-server designs, because a peer process (one that can both generate messages to other processes and receive [unsolicited] messages from other processes) can manage several messages simultaneously.

## Sleep: the Key to a Peer Process

Nothing is more important to a peer process than sleep, defined as the ability to wait for an event to happen and to know what to do when it does. A typical external event would be the receipt of an unsolicited message from a remote peer process.

A peer process usually defines the events it can handle and the possible responses to these events, and then waits for things to happen. Although this sounds simple, because algorithms for an asynchronous program are more complex than those for a synchronous program (such as a client or server application), asynchronous programs may need more maintenance. To make it easier to create and maintain successful asynchronous peer programs, the NetWeave API uses the `nwds_sleep` function.

NetWeave must control waiting in a program. Because many operations can occur within the NetWeave library before the next message is returned to the application, these internal operations can occur

asynchronously only if NetWeave controls the waiting mechanism. You can also have a conflict if user input (such as keyboard and mouse events) is processed synchronously, or if the application performs lengthy synchronous operations that can interfere with NetWeave's control of asynchronous events at the application level. To prevent these types of conflicts, you may want to make an application that interacts with a user synchronous, or use a multithreaded server that can handle several operations to be processed concurrently.

There are four ways to use the `nwds_sleep` function, though not all of these modes can be used on every platform:

- After all events are registered and set up, simply sleep forever, allowing callback functions to handle all event processing.
- Same as above, but periodically wake from the sleep function to check for conditions such as message timeouts or shutdown conditions.
- Awake from the sleep call whenever an external event occurs to check on some previously defined and registered condition. Generally this is not a very good idea (and should not be necessary), because you are creating two event wakeup conditions for every event.
- Implement `nwds_sleep` with no timeout, as you would for the UNIX `poll()` service. This is useful if the caller simply wants to check for pending NetWeave events and then do some other processing.

In most cases, `nwds_sleep` accepts an item list and a timeout value (in milliseconds). To specify which mode of repose you want to use for your program, use the `nwds_sleep` item types `NWDS_KERNEL_ONCE` and `NWDS_KERNEL_SUSPEND`. If a timeout value is negative, the application will wait forever. When the item list is omitted (NULL), the default operations are:

- Sleep forever (if the timeout value is negative).
- Suspend the user's application for the timeout number of milliseconds.

The table below lists the four `nwds_sleep` function modes. In this table, an *event* is defined as the completion of an asynchronous operation that the user started, or that NetWeave started on the user's behalf. An *event callback* is what happens in response to a particular event. (Technically, it is a structure that specifies which function to perform when the event occurs.)

The <code>nwds_sleep</code> function modes	Description
SUSPEND ONCE, WAIT FOREVER	If the timeout is negative and <code>kernel_once</code> is specified: <ol style="list-style-type: none"> <li>1. Wait forever for the first event.</li> <li>2. Call the event callback.</li> </ol>
SLEEP FOREVER	If the timeout is negative but <code>kernel_once</code> is not specified, keep calling the event callbacks forever.
SLEEP, WITH TIMEOUT	If the timeout is positive and <code>kernel_suspend</code> is specified, keep calling the event callbacks until the timer expires.

The <code>nwds_sleep</code> function modes	Description
SUSPEND ONCE, WITH TIMEOUT	If the timeout is positive but <code>kernel_suspend</code> is not specified: <ol style="list-style-type: none"> <li>1. Wait for the first event, which may be the timeout.</li> <li>2. If it isn't the timeout, cancel the timeout.</li> <li>3. Call the event callback.</li> </ol>

In NetWeave, to associate a callback function with an event, you normally use the callback structure passed in the API. For example, the notification (second) callback provided in the call to `nwds_ipc_connect` associates the receipt of an unsolicited message with the user's callback function.

To allow the application the simplicity of a single wait point, NetWeave also provides a mechanism for associating the `nwds_sleep` function with external, system-specific events. A programmer may want to respond to events unknown to NetWeave, such as asynchronous inputs from a user's mouse or keyboard. Because "event" is defined in the same way across all platforms, the APIs for defining an event to NetWeave are platform-specific. For example, the `define_event` functions on UNIX are based on a File Handle (File \*) while the `define_event` function on MVS is based on VMS Event Flags. For a complete description of `nwds_sleep` and its parameters, see the *NetWeave API Guide*.

## Exiting from Sleep Forever

Sleep forever is the simplest mechanism to use in a fully asynchronous program, because you can define all your connections when the program is initialized and then let the main loop sleep forever. Sometimes, however, it is convenient to return to the main program loop – if only to do some post-processing after all NetWeave operations are completed. There are two ways to escape from a Sleep Forever call:

- Set a callback to use the item list type `NWDS_KERNEL_EXIT` to tell NetWeave to terminate its current sleep forever processing loop.
- Use `nwds_sleep_callback`, which is more flexible than `NWDS_KERNEL_EXIT` but a bit more complicated to use. Instead of instructing the main sleep loop to terminate, NetWeave calls the user's `nwds_sleep_callback` function to perform the desired action.

For more information about `nwds_sleep_callback`, see *The NetWeave API Guide*.

`NWDS_KERNEL_EXIT` example:

```
static int exitSleep(void)
{
    static char    fname[] = "exitSleep";
    NWDS_ERRNO    status;
    NWDS_ITEM_LIST items[2];

    items[0].type = NWDS_KERNEL_EXIT;
    items[1].type = NWDS_END_OF_LIST;
```

```

status = nwds_sleep( -1L, items, NULL);
if (status != NWDS_SUCCESSFUL) {
    nwds_msglog(NWDS_MLSEERROR, "%s, nwds_sleep error %d", fname, status);
    return TRUE;
}
return FALSE;
} /*end exitSleep */

```

If you intend to use `nwds_sleep_callback` to exit from sleep forever, you must enter the sleep forever loop with the special item list type `NWDS_KERNEL_LOOP`. The code fragment below shows how to do this:

```

NWDS_ITEM_LIST  items[2];

items[0].type = NWDS_KERNEL_LOOP;
items[1].type = NWDS_END_OF_LIST;

status = nwds_sleep( -1L, items, NULL);

```

## Making Connections in Asynchronous Mode

In a peer-to-peer environment, once a connection between two nodes is established, both members of a messaging conversation can send any number of messages to a partner at any time. A connection between two nodes consists of an active partner (the one who places the call) and a passive partner (the one who waits for its partner to make a new call). By convention, the active partner is called the client and the passive partner is called the server.

To enable a client to connect to a server, the server must make itself known to NetWeave by issuing `nwds_ipc_publish` and specifying both a *public name* and a *new call callback*. The public name is a group name in the NetWeave INI file that identifies which communications protocol the server is using. The `new_call_callback` is the function that will be called when a client makes a new connection.

To initiate a connection, the client calls `nwds_ipc_connect` and specifies a new data callback and the public name to which it wants to connect. The public name must match the one that the server has already published. NetWeave returns to the client application a handle for this connection. When a message arrives on this channel, NetWeave will use the notification function to call the server's new data callback.

To complete the connection setup process, the server must issue `nwds_ipc_accept`. One of the `nwds_ipc_accept` parameters is the new data callback that the server uses to process data that it receives from the client. NetWeave returns to the server application a handle that identifies this connection. If the server wants to reject the call, it must first accept the call to retrieve the handle associated with the underlying connection, and then call `nwds_ipc_shutdown` to break the connection.





## Internal Queues for Asynchronous Messages

NetWeave maintains several internal queues for managing message delivery. As each asynchronous message is received, NetWeave places it on one of these internal queues and then calls a notification callback function to alert the application that there is a new message. You can specify the notification callback as part of the call to either `nwds_ipc_connect` or `nwds_ipc_accept`.

The message recipient does not have to read messages within the context of the notification callback. Rather, the designer may record in the application that a message is available and can be processed later. Because NetWeave saves messages on behalf of the recipient application, there is no actual I/O associated with the `nwds_ipc_read` call: NetWeave has already “read” the message and placed it on an internal queue.

**NOTE:** A call to `nwds_ipc_read` is always synchronous and always retrieves messages from a NetWeave internal queue.

## Asynchronous (Queued) Message Services

Queued message services, which are always asynchronous, are used to connect autonomous applications. Because queued messaging is connectionless, the interaction between applications consists of a very simple unidirectional flow of information that does not require acknowledgment. The middleware assures *reliable message delivery* and *error recovery*.

Queued messages are ideal for connecting two or more programs that have logically independent algorithms, a situation that typically occurs in the *workflow* model of system design. An example of workflow is an office comprised of autonomous departments. In this office, a purchase order moves through sales, warehousing/delivery, accounting, and post-sales marketing. Each department performs its operations with little or no interaction with other groups, and regards the other departments as black boxes: it is clear what information goes in and what comes out, but internal operations are invisible.

Queued message services have much lower throughput statistics than do client-transaction services. Because of the slower throughput, most queued messaging services have a scheme for prioritizing messages: high priority messages have their own queue, and the message delivery software gives preference to this queue.

### How a Queue Works

A queue (also known as a FIFO, because messages are processed on a First-In, First-Out basis) provides asynchronous message delivery that doesn't require acknowledgment. A FIFO connects one or more *producer* applications with one or more *consumer* applications. A producer puts messages at the tail of the queue, and a consumer gets messages from the head of the queue, as shown below:

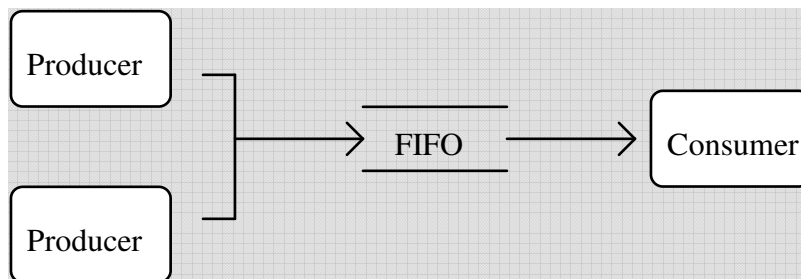


Figure 6. FIFO queuing environment

The client continues processing until it receives notification that an operation has completed.

Often, batch programs that were historically linked by tape transfers are instead loosely joined via a queue – in which case the FIFO queue replaces the tape. This means that one or more producer applications treat the queue as an endless tape, and the consumer process can continue as a batch process that starts up, reads the FIFO until empty, closes it, reschedules itself and exits.

The queued message service is responsible for error control and recovery, often referred to as *guaranteed message delivery*. The producer application's responsibility for the information ends when the queued message service is invoked. The consumer is guaranteed that all messages will be received unchanged, in the order in which they were generated.

To guarantee that messages will be delivered no matter what CPU or communications subsystem failures occur, NetWeave implements queues on disks instead of in memory. If you are concerned about



the extra disk activity that NetWeave imposes on an already too-busy system, you should use IPC messaging instead of message queues. IPC messaging has higher throughput than queued messaging because it does not store and retrieve messages from the disk.

To eliminate contention among producers and consumers, a NetWeave Agent controls input and output to queues and communicates with the NetWeave library to store and retrieve the messages.

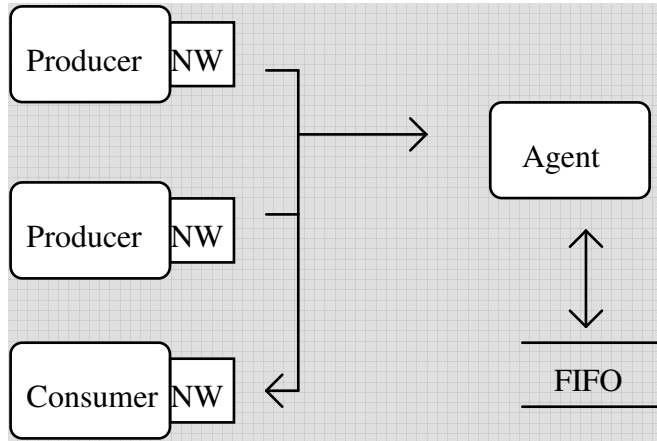


Figure 7. The NetWeave Library and Agent relationship in a queuing environment

### Queue Location

A NetWeave queue can be located on the producer's computer, on the consumer's computer, or even on a third system connected to the producer and consumer. A queue is usually created on the producer's computer to allow the producer to continue operating even if the consumer's system (or the network) goes down. Figure 8 shows a queue located on the producer's platform.

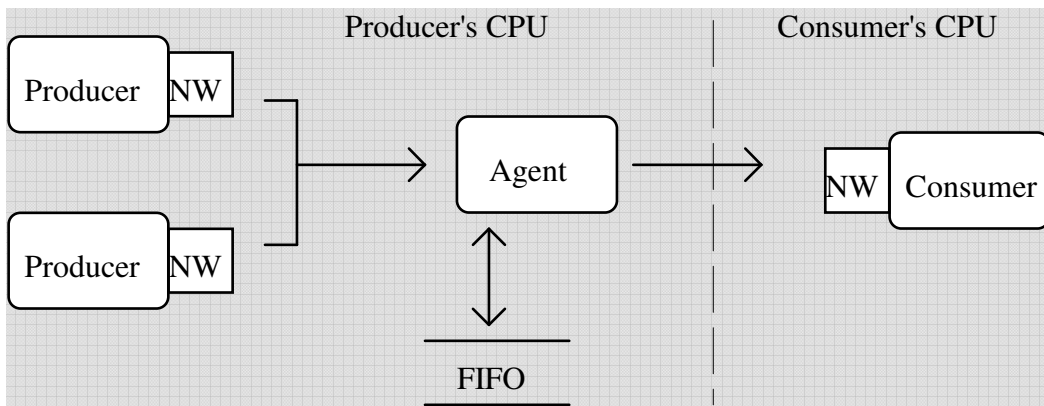


Figure 8. Client side queue location

Another good location for a queue is on the most reliable system in the network: either the producer, consumer, or a third system. Figure 9 shows a diagram of a queue located on an independent system:

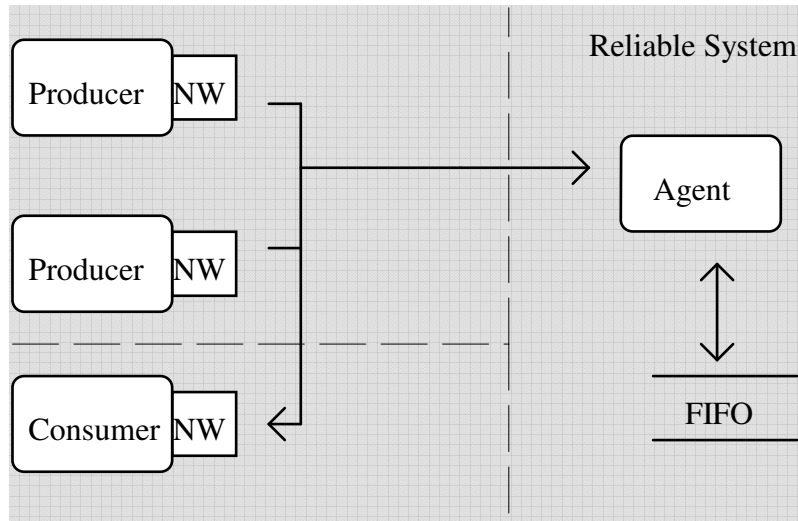


Figure 9. Queue located on (independent) reliable system

## Queued Messaging Functions

Because queues are implemented as files on a disk, the NetWeave API for queued messages has the following functions:

Function	Description
nwds_file_open	Opens a FIFO to read or write. NetWeave builds a connection between the application and the Agent that controls the queue.
nwds_file_close	Closes a FIFO. The file is closed and the connection is shut down.
nwds_file_write	Allows a producer to add a message to the tail of the queue.
nwds_file_read	Allows a consumer to read the first message from the head of the queue.
nwds_file_position	Used with nwds_file_read to read queues in transaction mode (see below).

In addition to the functions that access queues, there are three other functions for managing queues that you can call from applications dedicated to operations control:

Function	Description
nwds_file_create	Creates a FIFO.
nwds_file_remove	Deletes a FIFO.
nwds_file_info	Retrieves statistics about the current state of the queue.



To read a queue, a consumer can use either of the following NetWeave mechanisms:

Mode	Description
Standard mode	NetWeave updates internal pointers when the message is read. Use standard mode if more than one consumer will process a queue.
Transaction mode	When NetWeave reads a message, it doesn't change the internal pointers. After the consumer has finished processing the message, it calls <code>nwds_file_position</code> to change the pointers. Use transaction mode (with a single consumer per queue) to guarantee that no message is lost if a consumer application fails.

### Designs with Multiple Queues

If your system will encounter any of the following conditions, the system design may need to include more than one queue:

- Transaction-oriented message processing (queue per consumer)
- The need to isolate a producer application from the network (queue at the producer's system)
- Using queued messages with broadcast or IPC messages (multicast)
- Using multiple queues to implement multiple priority messaging

When a queue feeds a consumer on a system that uses transaction monitoring, you can lose messages from the queue if a message is read from the FIFO and the transaction is aborted later in the processing cycle. To protect against this type of message loss, open the FIFO with an item list set to read hold and use `nwds_file_position` set to -1 to signal when processing is complete.

**NOTE:** Even if the consumer system does not have a TP monitor, transaction-oriented queue processing may simplify application recovery.

If an application requires a queue per consumer, you can use the NetWeave Agent on each consumer's system to manage all I/O to and from the FIFOs, as shown in Figure 10:

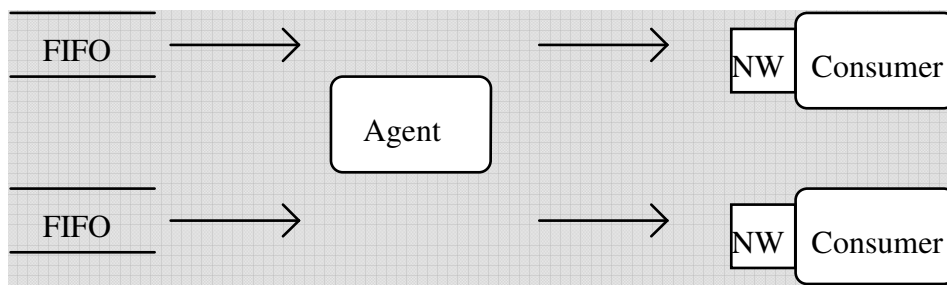


Figure 10. A NetWeave Agent managing multiple queues

If a design includes a queue on the producer system and one or more queues on the consumer systems, you may want to include another element of queue architecture, the Distributor application. The NetWeave API makes it very easy to create a Distributor application. In Figure 11, a Distributor on the consumer platform reads messages from the producer's queue and writes them to one or more consumer

queues. The Distributor is a natural place to add logic that monitors queue statistics, and it can be set up to do the following:

- Issue regular statistical reports about the number (and type) of messages processed
- Respond to messages from a network monitor program to report these statistics

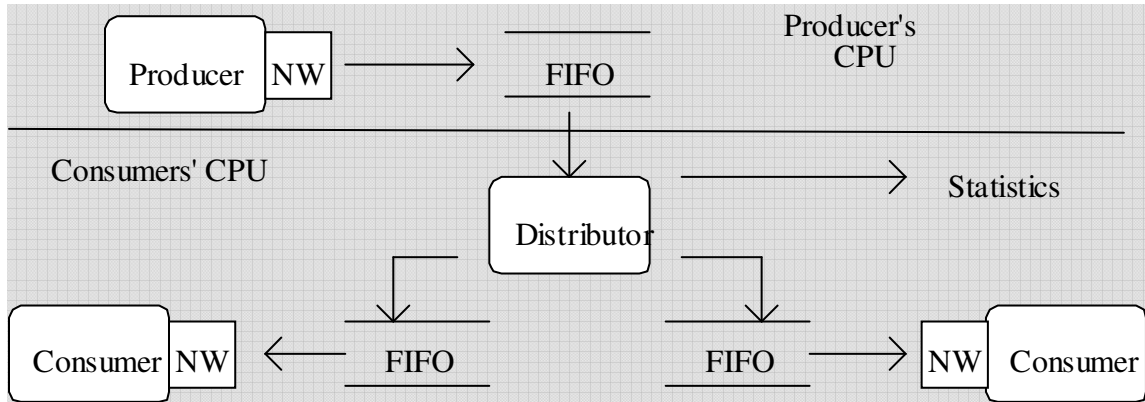


Figure 11. Multiple queues and a Distributor process on the consumer's system

To build a simple, elegant multicasting delivery system, you can combine broadcast services with queued message services. In multicast, a message is delivered to each receiving system – in this case, to each consumer. For example, a host system broadcasts a message to a network of workstations. A collector process on each workstation receives the broadcast and writes the message to a queue (or, for a Distributor process, to more than one queue) on the consumer workstation.

Figure 12 does not display the Agents, although there is one on the producer system and another one on the consumer system.

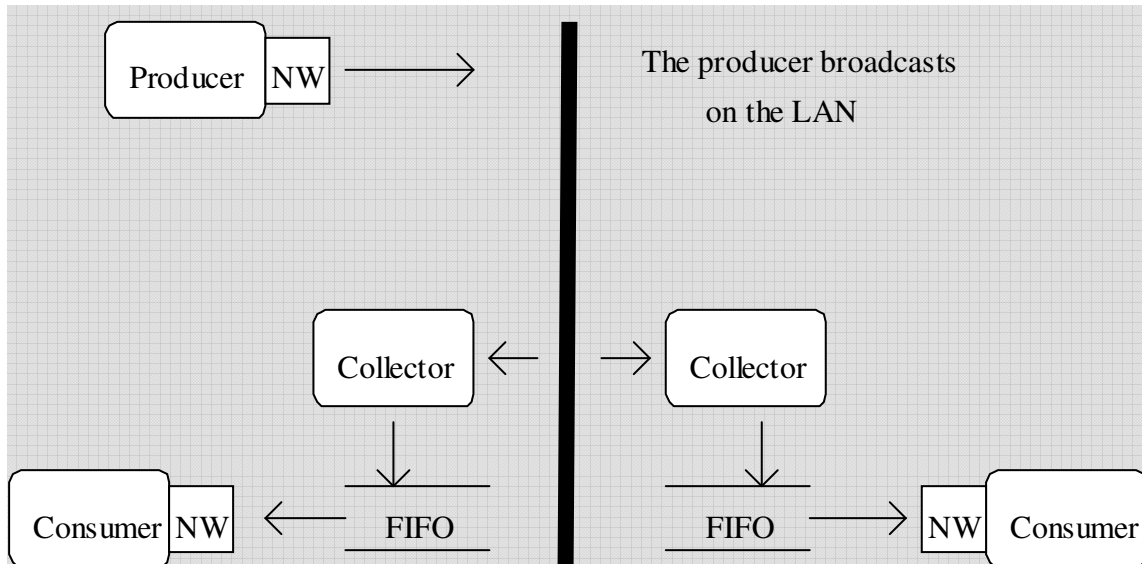



Figure 12. Combining multicast and queued messaging



## Sizing a Queue

NetWeave stores long messages as a series of segments. However, the application is not aware that the messages are being broken up and restored. Although a particular platform's disk subsystem determines what the maximum segment size can be, within this limit you can specify whatever segment size you want when you design the system.

The following parameters determine the size of a queue:

- Size of each message segment
- Number of segments per queue

If the FIFO will be used to deliver a mix of mostly small messages and an occasional large one, your queue is more efficient if you set the segment size to accommodate the small messages and then allow NetWeave to spread the large messages across several segments. If the mix contains equal numbers of large and small messages, you can improve the queue's throughput by setting the segment size to the maximum message size, or to the maximum message size of the queue.

## Expedited Messages

Because NetWeave has IPC message delivery to support OLTP, there is little incentive to add priority mechanisms to the queued message delivery. If an application has a class of messages that require immediate action, use IPC messaging or create a second high-priority queue that the recipient always checks before reading the standard message queue. If an application has several types of messages with differing processing priorities, use the Distributor design with multiple queues (see above) to build a queued message system with priorities.

## Broadcast Services

In message delivery by broadcast, one application (the sender or broadcaster) sends a message that will be received by any number of applications (receivers or registrants) on the network. Although the NetWeave API for broadcast services is part of the general IPC services, there are substantial differences between IPC message services and broadcasts.

Broadcast messages are sent to and received from a group name in the INI file. One of the parameters in the group specifies an IP port, identified by a number between 1 and 32K. IP ports numbered less than 1K, called the well-known ports, are reserved. (For the purposes of this discussion, we use the terms *port* and *INI file group* interchangeably.) You may use a single port to exchange several classes of messages between the same INI file group, or use multiple groups – more than one port – to subdivide the broadcasts.

Within the set of messages being sent to a particular broadcast port, you can further subdivide messages by *filter class*. The filter class is an integer that the application specifies as part of the NetWeave broadcast API on both send and receive operations. For each message that it sends, a sender identifies the port and filter class. In synchronous mode, a receiver reads all messages for a specified port and filter class. In asynchronous mode, a receiver may associate with each port and filter class combination a unique function to process the message type. How a sender broadcasts its messages (synchronously or asynchronously) does not affect how the receivers read them: some may read messages synchronously, others asynchronously.

It is easy to set up a sender to do broadcasts. For the function `nwds_ipc_broadcast`, you must specify the group name of a broadcast entity, a filter class, and a message. Because broadcasts are connectionless, you don't have to set up any other sender functions. A receiver that reads broadcasts synchronously is a single-threaded server. To design a synchronous receiver, use the following functions:

Function	Description
<code>nwds_ipc_register</code>	Identifies the group and filter class you want to receive. NetWeave returns a handle associated with this set of broadcasts.
<code>nwds_ipc_read</code>	To begin receiving messages, call this function immediately after <code>nwds_ipc_register</code> . Your application will block on <code>nwds_ipc_read</code> until the next broadcast is received.

As with asynchronous IPC messages, NetWeave notifies the application as each message is received, but holds each message in an internal queue until the application reads it. (Note the discussion in the previous sections about internal queues.) To design an asynchronous receiver, use the following functions:

Function	Description
<code>nwds_ipc_register</code>	Tells NetWeave what type of message (group name and filter class) the receiver wants, and which callback function will be called when that type of message is broadcast on the network. For each call to <code>nwds_ipc_register</code> , NetWeave returns a unique handle associated with the specified message type.



Function	Description
<code>nwds_ipc_options</code>	If needed, call <code>nwds_ipc_options</code> from the callback to determine how large the incoming message is, or who sent it.
<code>nwds_ipc_read</code>	From the callback, dequeue the message from the internal message queue.

To use NetWeave's broadcast services, senders and registrants must be connected by an IP backbone. Broadcast services are implemented on top of UDP datagram services. (UDP, like TCP/IP, is an IP protocol.)

By design, UDP datagram service is unreliable and connectionless, meaning that there is no context saved between the delivery of one message and the next. Because the service is connectionless, the call to `nwds_ipc_broadcast` does not have a handle associated with any underlying broadcast mechanism, and the handle returned to the receiver by `nwds_ipc_register` is not associated with any underlying communications. Instead, this handle identifies the internal queue of messages associated with the group name and filter class for which the recipient registered.

The "unreliable" label also means that there is no guarantee that any given datagram will be delivered to a particular node in the IP network. However, UDP does guarantee the internal integrity of each datagram that is delivered, and NetWeave will provide reliable broadcast message delivery as defined below:

- The rate of delivery is tunable to optimize the chance that each receiver will receive a given packet.
- The system detects loss of individual packets at a given receiver.
- Lost packets can be recovered as long as the data rate is not already approaching the application-defined maximum.
- The system detects unrecoverable interruptions in broadcast services and notifies the receiver.

There are two ways to ensure reliable NetWeave message delivery:

- Throttle a sender to prevent it from overwhelming the network or the intended receivers.
- Let a receiver recover from a failure to receive a particular packet.

The need to throttle a sender depends on the system's processing load and the sender's speed relative to the slowest potential receiver to whom you want to send messages. Use a prototype broadcast application, not your intuition, to measure what actually happens when your sender broadcasts to your slowest receiver. To measure performance, use the NetWeave diagnostic and training tool TST.

For more information about using the INI file parameters `THROTTLE_INTERVAL` and `MAX_THROTTLE_INTERVAL`, see the *NetWeave Configuration Guide*.

Think of a broadcast as a one-way flow of information from a sender to an unknown number of receivers. To help ensure a reliable message delivery mechanism over datagram services, NetWeave lets receivers send a limited number of control packets back to a sender. There are two types of control packets:



Packet type	Description
Resend	Identifies one or more packets that were not received, and that should be retransmitted by the sender.
Flow-control	A request to the sender to momentarily suspend broadcasts to allow the receiver to catch up.

Because system performance degrades quickly if significant numbers of control packets are generated during broadcasting, NetWeave logs each control packet to the trace file. If your receivers are generating an excessive number of control packets, consider throttling the sender.

To control when and how a receiver reacts to a missed packet, you can set the INI file parameters as described in the *NetWeave Configuration Guide*. NetWeave maintains internal information (sequence numbers) that senders can use to retransmit lost data, and receivers can use to either detect a missed packet or identify and ignore any duplicate packets.

## Data Server Services

Data server services allow all computers in the network to access a particular computer platform's file system. You can use database services to access relational database systems and legacy file systems (both record-oriented and flat file). Legacy applications are commercial and scientific applications written since the late 1970s that share one or more of the following features:

- The application resides on a single hardware platform.
- The user interface is the traditional character-oriented terminal.
- To access related application functions, you have to use menus and function keys.
- Application data are stored in record-oriented files.
- These records are typically accessed using keys and indices.

By providing open access to legacy databases, NetWeave lets a business preserve its substantial investment in applications development while providing a migration strategy from legacy systems to applications written for a distributed and heterogeneous environment. Figure 13 shows the typical configuration of distributed applications that use data server services:

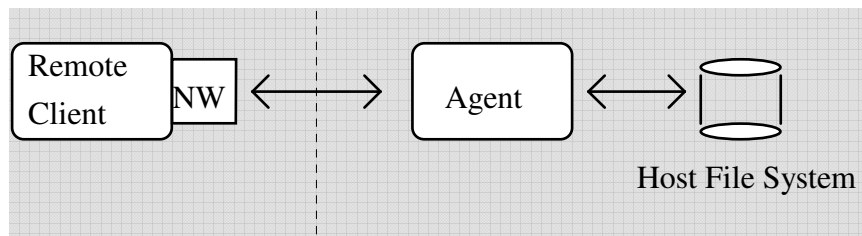


Figure 13. Typical data server configuration

From NetWeave's point of view, accesses and updates originate on a client platform and are delivered to the NetWeave Agent on the host (the computer where the files being updated and/or queried reside). In a typical installation, the NetWeave Agent can access the local file system and perform these file access functions on behalf of remote clients. The Agent is also responsible for all aspects of data security and translation. For more information about security and data translation, see the topics "System Security" on page 35 and "Data Translation Services" on page 37.

### Flat Files

A flat file contains information in a structure known only to the application. Because there is no external record definition or table schema that describes the structure of the data, only the logic of the application knows the file structure. As seen and processed by the operating system, a binary flat file does not have record structure. As long as you have a relative byte address, you can access information stored within the flat file simply by transferring continuous streams of bytes to or from the disk.

Some systems use a special form of flat file for managing variable-length text strings. For the purposes of this discussion, text files are considered flat files. Access to these types of text files is often limited to sequential reads from the beginning and sequential writes to the end. On all platforms, you can use NetWeave to access any flat file (binary or text) that can be processed according to the rules of ANSI C I/O functions.

A binary file is a sequence of bytes of any length and any content. A text file in C is a contiguous set of C strings. Each string may not exceed 255 bytes, though for some systems the limit is even lower. Although each system may adopt its own rules about how to terminate a text string within a C file, input to or output from a text file is always a C string, a series of printable bytes terminated by NULL.

## Record-Oriented Files

Because each hardware manufacturer handles record-oriented files in its own way, it is hard to generalize about NetWeave's file access functions beyond the following:

- You can create and remove files.
- You can test for the existence of a file.
- All record-oriented systems support sequential data access (read from beginning, write to EOF).
- Most systems support random retrieval of fixed-length records by relative record number.
- Most systems support random retrieval of variable-length records by keys and associated indices.

The indexed sequential format is the most important form of record-oriented file structure. To make it faster and easier to update individual records in the main file, the operating system builds an ancillary collection of indices based on record and key definitions that you specify.

Because record access and file management information is system-specific, NetWeave relies heavily on item lists to supply the customization required to process a record or file on a particular platform. Although the basic NetWeave API and functions for accessing legacy files are the same across all platforms, you must have a detailed understanding of how a remote file system works in order to access it through NetWeave.

## File Access Resources in the NetWeave Header File (`netweave.h`)

To build a distributed application that can access a file system on a particular platform, you must understand the properties of the host you need to access. The main NetWeave header file `netweave.h` provides a set of item list parameters that you can use for customized access to each supported file system. An item list is an array of structures that contain the following three elements:

Element	Description
Item type	Found in <code>netweave.h</code> in the section for the particular hardware vendor. The Item type identifies a specific optional parameter that is being specified in this item list entry.
Length	The length (in bytes) of the value of the item, determined by the item type's encoded class code.
Address	The address in data storage where the value of the item is located. When an item type is an enumeration, the possible values are also declared in <code>netweave.h</code> .



To manage legacy files, use the following API functions:

Function	Description
nwds_file_info	Determines whether the file exists. If it does, returns statistics about the file's current structure and content (the number of records).
nwds_file_create	Creates a new disk file whose characteristics are determined by system-specific item list elements that you specify.
nwds_file_remove	Removes (erases) a file from the disk.
nwds_file_open	Opens an existing file as specified in the item list that you define, and returns a handle that the application can use for future operations on that file.
nwds_file_close	Closes a file and releases any resources associated with it.

To access and update records, use the following functions:

Function	Description
nwds_file_position	Selects a current record for future operations.
nwds_file_read	Retrieves the record, and may lock it for to prevent future updates.
nwds_file_write	Adds a new record to the file.
nwds_file_delete	Deletes the current record from the file.
nwds_file_update	Changes the current record (first removes the lock, if needed).

## Asynchronous Notification of File Changes

NetWeave has a powerful mechanism for building distributed applications that can react quickly to change. In contrast to the repetitive, operator-intensive model of bulk file transfers and the scheduled batch jobs for processing them, NetWeave provides “trigger” functions that notify applications about file changes as they occur.

Figure 14 shows client #1 updating a record in the file. When the update is completed, the Agent notifies client #2 of the change. It does not matter where the client applications are located – either one or both may be local or remote.

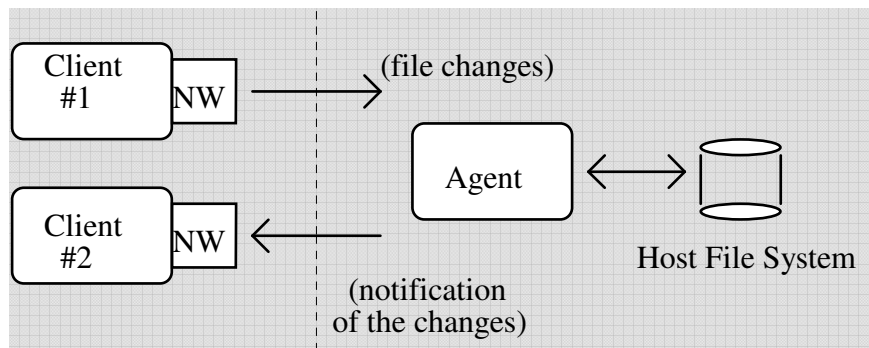


Figure 14. File trigger operations

To build an application that uses file triggers, use the following functions in this order:

Function	Description
nwds_file_open	Returns a trigger handle for NetWeave to use in subsequent operations.
nwds_trigger_register	Allows an application to register for any combination of adds, updates or deletes. The application supplies a (notification) callback procedure for NetWeave to call when a change occurs.
nwds_trigger_read	To add a new record or delete an old record, NetWeave returns the record image that was added or deleted. If a record was changed, NetWeave returns the images of the record both before and after the change. Often nwds_trigger_read is called from the notification callback registered with NetWeave.
nwds_file_close	Closes the file and removes the registrations.

## Using NetWeave's SQL Interface to Access Relational Databases

NetWeave provides a powerful set of functions for accessing and updating information in relational databases. Although these functions provide a standard interface to all databases that can be accessed through a dialect of SQL, the application is responsible for correctly forming queries and updates in the dialect of the particular target database.

## Tandem NonStop SQL

All functions in the `nwds_sql` API are available for NonStop SQL. The NetWeave Agent for Tandem keeps a static pool of special programs (the NSSQL servers) that supply the SQL services and maintain SQL context information for remote client applications. When the Agent is started, NetWeave uses the same script that starts the Agent to create the NSSQL servers, either under user control or as part of the system boot sequence. Each server must have a unique name that is specified with INI file entries.

For example: the command to start a NSSQL server named \$NS1 can accept two runtime parameters, the root group and the INI file. Normally an NSSQL server uses the same INI file as the Agent. If you

want the server to use a different INI file, change the root group and root INI file parameters passed at startup by entering the following:

```
run nssql/name $NS1/NS1 nwdsini
```

To work properly, the Agent's INI file must contain a well-known group called [SQLCONNECT group]. This group recognizes a single entry (CLASS) that lists the names of the groups associated with each of the NSSQL servers.

The example below shows a static pool of two NSSQL servers, one with root group NS1, the other with NS2. The Agent uses NetWeave's DOLLAR\_RECV protocol to communicate with the processes named \$NS1 and \$NS2.

```
[SQLCONNECT]
CLASS = { NS1, NS2 }

[NS1]
LOCAL_PROTOCOL = 1
PROTOCOL = DOLLAR_RECV
TANRECV_PROCESS = $NS1
@TRACE_FILE@ = $$.#NS1

[NS2]
LOCAL_PROTOCOL = 1
PROTOCOL = DOLLAR_RECV
TANRECV_PROCESS = $NS2
@TRACE_FILE@ = $$.#NS2
```

To install a NSSQL server in the Tandem SQL environment, use the NonStop SQL compiler to link with your catalog as follows:

```
sqlcomp/in nssql/ catalog <your catalog>, explain
```

## System Security

The NetWeave API functions that implement security provide a platform-independent authentication interface that is same across every platform. Because the O/S vendor already provides system-level security, NetWeave's function here is to provide wire-level (as opposed to system-level) security. In addition, NetWeave implements a challenge/response mechanism that the application may also use, if desired.

The illustration below shows the differences between OS validation and challenge-response when using the functions `nwds_logon`, `nwds_password`, and `nwds_logoff`. This diagram applies to both message-oriented middleware and data server services:

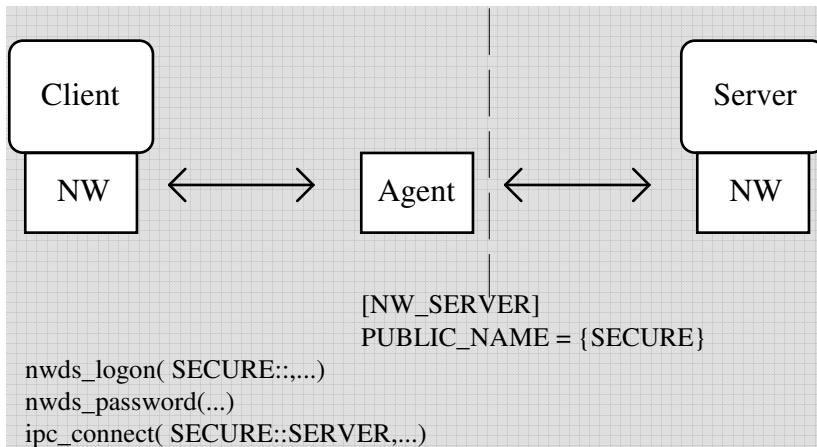


Figure 15. NetWeave security diagram

For IPC messaging, the Agent controls access to the application server by routing all communication between client and server through the Agent (named SECURE in this example). In this capacity, the Agent acts as a firewall for the server.

For queued message services and client database services, you can access the files of the host system or queues on another platform only through the NetWeave Agent. When authentication is enabled at an Agent, all functions are rejected until the remote client successfully finishes logging on.



Figure 16 shows how the Agent generates a challenge string (a stream of bytes) that is returned to the client in the response to the logon request. If the client is who it claims to be, it changes the challenge string to a unique response string that is returned to the Agent via the `nwds_password` command. For example, if the client's algorithm specifies that each vowel be replaced by its succeeding consonant, a challenge of "who are you?" will result in a response of "whp brf ypv?"

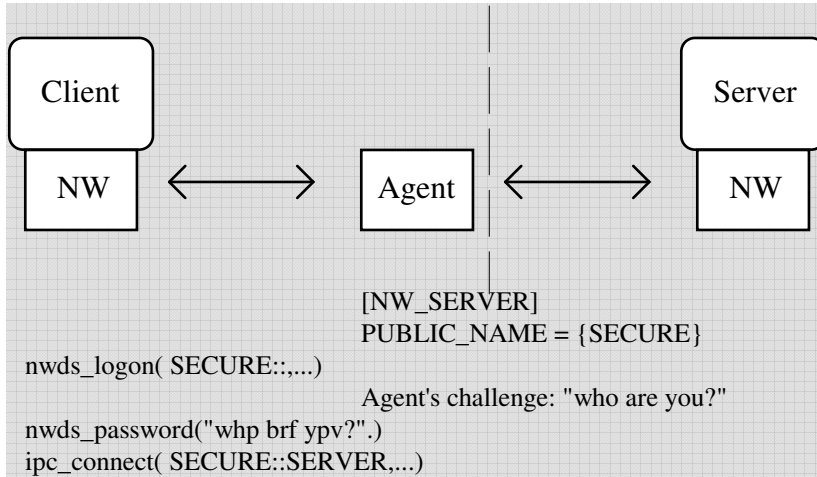


Figure 16. NetWeave challenge-response mechanism

## Data Translation Services

Data translation is useful for representing data in a consistent way across different computer platforms. If you are using one of the three forms of message-oriented middleware (IPC messaging, queued message services, or broadcasts) and all of the platforms in your system use the same character encoding scheme, you can avoid data translation altogether by constructing your messages entirely of character (printable) data. Although messages composed entirely of characters are much easier to read and debug during development, the message size tends to be larger and character/numeric translation is slower.

The following hardware platform characteristics determine whether you need to consider data translation in your design:

Issue	Explanation
Character encoding scheme (ASCII or EBCDIC)	The following NetWeave-supported platforms use EBCDIC: <ul style="list-style-type: none"> <li>• IBM mainframes</li> <li>• AS/400</li> <li>• Unisys A Series architecture</li> </ul> All others use ASCII.
Byte ordering in numeric data types (big endian vs. little endian architectures)	In big endian machines, the low-order byte is on the left. Big endian machines include IBM mainframes, Tandem, Stratus, and all UNIX systems except those implemented on Intel-compatible chips.  Little endian machines have the low-order byte on the right. Little endian machines include VMS and OpenVMS from Digital, all PCs, and UNIX systems built on Intel-compatible chips.


## Defining Data to NetWeave

To define the structure of a message or record to NetWeave, use the following syntax to create groups in your INI file:

```
[MyData]
DDL_ENTRY=1
DDL_NUM_FIELDS=3
DDL_FIELD_1=SHORT 2
DDL_FIELD_2=LONG 4
DDL_FIELD_3=STRING 20
```

In the example above, note the following:

1. The group `[MyData]` has three fields: a 16-bit integer, a 32-bit integer, and a variable-length string up to 20 characters (including the NULL terminator).
2. The token `DDL_ENTRY` is a binary entry whose default is 0 (false).

- 
3. If `DDL_NUM_FIELDS` is present and non-zero, NetWeave expects it to indicate how many definitions will follow.
  4. Field definitions have the format `<type length>`, where `type` is one of the following:
    - `SHORT`: length must be 2 to indicate a 16-bit integer
    - `LONG`: length must be 4 to indicate a 32-bit integer
    - `CHAR` and `STRING` may take arbitrarily large lengths. (For `STRING`, the length is a maximum, and the field contains a NULL-terminated string of bytes).

## Message Translation

Each message that NetWeave needs to translate must be defined in a unique group in the INI file of each system that will handle the message. On the sender's side, do the following:

1. Add the item type `nwds_ipc_convert_name` to the `nwds_ipc_write` call.
2. Set the value of the item list entry to a string of bytes containing the name of the group.

On the receiver's side, the message format determines which translation procedure should be used:

1. If all messages from a given sender have the same format, identify the sender by calling `nwds_ipc_options`. Then, call `nwds_ipc_read` with the item type again set as it was for the sender.
2. If a sender transmits more than one form of a message, the receiver must interrogate the message to determine how to decode it. Buried in the message are clues to its format.

To determine the location of your clues in an incoming message, you need to understand how NetWeave transfers data between systems. NetWeave encodes all character data as ASCII for transmission. Both 16-bit and 32-bit fields are transferred in four bytes, and the fields are "byte-stuffed" so that each field begins on a 32-bit (four-byte) boundary.

Often the receiving application is an existing program that cannot be changed. Therefore, the sending application must construct the message in the exact format that the receiver expects to read it, and it must interpret the reply from the stream of bytes that the receiver returns. In these cases, the simple message translation features in NetWeave may be inadequate, and you may want to consider the powerful translation features in *evolve*<sup>™</sup> middleware from Vertex.

## Record Translation

The INI file group that defines a user's file record layout must contain the structural definition of a record. Although this means that the file can contain only one record definition, it also simplifies the translation process. If the INI files on both the client side and the Agent side contain the same definition of the record, translation occurs automatically.



## Threaded Service

A thread is a unit of programming logic that performs the same functions as a program or process, but uses fewer system resources than a traditional process would. Threads typically operate within the context of a process, and are able to share many process resources within other threads within that process. Each thread has its own call stack and CPU state.

The input from the mouse, output to the monitor, a socket, a file descriptor – all are examples of system resources. A given system resource may belong to either the process as a whole or to an individual thread. A resource that belongs to the process is considered a process-based global resource. A resource that belongs to a thread is considered thread-based, and can be accessed only by that thread.

NetWeave provides both synchronous (blocking) and asynchronous versions of most functions. Although services written with synchronous calls are easier to understand and maintain, they cannot be expanded easily to service increasing numbers of requests. Servers that use asynchronous function calls are called *multi-threaded* because they can process an arbitrarily large number of service requests simultaneously. Because the callback mechanism is supported on all platforms, NetWeave uses it for asynchronous function calls.

Multi-threaded programming increases the amount of parallel processing within a larger application process, which improves performance and allows for more efficient use of system resources. In many ways, a multi-threaded architecture has the same benefits as an asynchronous program, with the additional benefit that each thread is implemented using a simple synchronous programming model, making it easier to implement and debug. A thread therefore, provides a mechanism for using synchronous communications in an asynchronous environment.

NetWeave uses both process-based resources and thread-based resources. The INI file parameters and the output to the common trace file are process-based resources, because each thread that uses NetWeave services within a given process shares these resources. Each `nwds_handle` obtained from a NetWeave function call is a thread-based resource, and cannot be shared between threads within a process. An `nwds_handle` is analogous to either a TCP/IP socket or a descriptor for a disk file other than the trace file.

Each thread that makes a NetWeave call is referred to as a *NetWeave thread*. A NetWeave thread must call `nwds_init` before calling any other NetWeave function, and it must call `nwds_exit` before the thread exits. To provide the process-based resources, we recommend that you use a *master thread* to make the initial call to `nwds_init`, and then have this thread create all other NetWeave threads. When the master thread exits, it must be the last thread to call `nwds_exit`.

There is another, technical sense of "thread" that refers to a different mechanism for handling simultaneous service requests. (This mechanism is supported on Windows NT and UNIX, but not on IBM/CICS or Tandem.) A thread is called a lightweight process because each thread has its own execution stack and therefore cannot affect the operation of another thread. It takes less processor time to create and destroy a thread than it does to start and stop a regular process.

In the classic UNIX client-server model, the server receives a connection and then starts a child process to service that connection. (In UNIX jargon, this is called *fork and exec*.) The child process is dedicated exclusively to that connection, and lasts only as long as the connection does. In UNIX, processes can share handles, and the handle for the connection is one of the parameters passed to the process. Because the functions of one thread cannot affect those of another, a blocking call in one thread does not block execution of other threads. Therefore the child process is created as a single-threaded service that makes only synchronous or blocking calls.

In contrast to the UNIX model, threaded implementations cannot share handles. To circumvent this restriction, NetWeave includes a Dispatcher to provide a client-server solution that closely resembles the UNIX model. The Dispatcher creates an environment in which each worker thread can use synchronous NetWeave function calls without affecting other worker threads.

To start and manage a Dispatcher, use the following functions:

Function	Description
<code>nwds_dispatcher_create</code>	Creates a NetWeave Dispatcher thread that will operate within the context of the calling application.
<code>nwds_dispatcher_stats</code>	Obtains current statistics on the specified Dispatcher, including messages and thread counts since startup.
<code>nwds_dispatcher_stop</code>	Terminates a Dispatcher.

As each new connection arrives, the Dispatcher creates a new thread (the worker) to service the requests delivered over the connection. The Dispatcher also solves the programming difficulties of associating handles associated with external connections and handles known to a particular worker. Because this is all performed in the background by another thread (the boss) created by the `nwds_dispatcher_create` function, the caller's program is free to do other things.

Any programming language that supports binding to a C function pointer may use the Dispatcher functions. NetWeave uses a C language function pointer to identify the worker routine to the Dispatcher. Each Dispatcher requires a unique set of three public names, called publish names:

- External publish name: provides connection for external clients.
- Internal publish name: provides connection for application-provided worker threads.
- Control publish name: retrieves statistics and shuts down the Dispatcher.

When the Dispatcher accepts an external connection, it also creates a worker thread to service the requests on that connection. This worker thread establishes an internal connection with the Dispatcher, which then links the external and internal connection to form a virtual connection. The worker thread executes the function that the application has requested. Once created, a Dispatcher will run indefinitely, even if it is not servicing any connections.

**NOTE:** When developing an application for the Dispatcher environment, it is a good idea to periodically use the `nwds_dispatcher_stats` function retrieve Dispatcher statistics such as the number of active service threads (connections) and the total number of messages that have passed through the Dispatcher.



## Topics for Programmers

The information in this section assumes that you have already chosen the tools for your system, that the design is adequate for configuring the NetWeave INI files (as described in the *NetWeave Configuration Guide*), and that you are looking for practical tips for coding and testing NetWeave applications.

Middleware facilitates the interoperability and portability of programs in a distributed computing environment. One of the major tradeoffs of making a particular function call is the desire to simplify the call versus the need to provide access to specific features of a particular hardware vendor's operating system.

NetWeave functions may be called either *synchronously* or *asynchronously*. A synchronous operation completes within the context of the call: when the function returns, the operation is completed. An asynchronous operation, on the other hand, completes some time after the initiating function call completes: when this call returns, the programmer knows only that things have started properly.

A NetWeave application may be built in any programming language that can call C functions. NetWeave itself is written in ANSI C because this is the only language with sufficient technical resources (communications and file system interfaces) that is implemented in the same way on every hardware platform. Although the "Sample NetWeave Programs" section on page 78 includes programming examples written in other common languages (COBOL85, ALGOL, Visual Basic), the information below assumes that you are familiar with C conventions and data types.

Keep in mind the following useful tips for application authoring:

- Prototype your first NetWeave application with synchronous calls.
- Do not mix synchronous and asynchronous calls in the same application.
- Where possible, unit test your distributed application in *loopback mode* on a single platform.
- Be generous when creating timeouts to your applications.

## The NetWeave Header File

To learn about the NetWeave API, the best place to start is the NetWeave header file, `netweave.h`. This file defines all enumerations, constants, typedefs, and prototypes in the public NetWeave API. Every NetWeave application must include the appropriate NetWeave header file.

Every release of the API includes the latest version of `netweave.h`. The following languages and header files are used for NetWeave application authoring:

Language	Header file
C	<code>netweave.h</code>
PowerBuilder	<code>netweave.pb</code>
16 bit Visual Basic	<code>netweave.vb</code>
32 bit Visual Basic	<code>netweave.vb5</code>

## Enumerations and Constants

In C, an enumeration is a set of comprehensive and mutually exclusive constants that cover a variable's range of values. The most important NetWeave enumerations are listed below:

Name	Description
NWDS_ERRNO_ENUM	<p>The list of return codes, most of which indicate fatal errors. Each NetWeave function returns one of the codes in this list. The “good” return codes are:</p> <ul style="list-style-type: none"> <li>• NWDS_SUCCESSFUL</li> <li>• NWDS_PENDING (returned when an asynchronous operation has started successfully)</li> </ul>
NWDS_SYSTEM_ENUM	<p>The list of systems that NetWeave supports. The system codes are returned by calls to <code>nwds_system_type</code> and <code>nwds_ping</code> and become input parameters to <code>nwds_convert_data</code> (the function that converts data from one system format to another).</p>
NWDS_DATA_ENUM	<p>The list of supported data types, which are especially important for constructing item lists and performing data conversion. The low-order nibble (4 bits) of each item type in an item list indicates the item type's data type. For more information about item lists, see page 44.</p> <p>Most runtime parameters (parameters that control operations in NetWeave and are determined when the program executes) are specified in the NetWeave configuration INI file. An important exception is <code>NWDS_MAX_USER_SIZE</code> (see below).</p>
NWDS_MAX_USER_SIZE	<p>The maximum size of a single application message (32567 bytes).</p>

## Typedefs

In C, a typedef is a data type constructed from more primitive data types. Typedefs help the compiler detect syntactical errors in your program. The table below lists the main `netweave.h` typedefs:

Data type	Description
NWDS_HANDLE	An arbitrary value that uniquely identifies an object for a series of related function calls. NetWeave generates handles and associates them with partner processes, open files, transactions, file triggers, and broadcasts.
NWDS_CONTEXT	One of the most important (and most abstract) concepts in NetWeave. Context means whatever information NetWeave needs to associate with a particular function call. For more information about context, see page 56.
NWDS_ITEM_LIST	A NetWeave application uses item lists to pass system-specific information between the application and NetWeave.
NWDS_CALL_BACK_PROC	<p>A callback procedure uses the following two arguments and does not return a return code:</p> <ul style="list-style-type: none"> <li>NWDS_CONTEXT is the information you asked NetWeave to remember when you initiated the function call that caused the event associated with the callback procedure.</li> <li>NWDS_ERRNO is the status of the event that initiates the callback.</li> </ul> <p>NWDS_CALL_BACK_PROC is an essential component of asynchronous (event-driven) programming. As NetWeave implements the asynchronous model, the programmer links functions (callback procedures) with events.</p>
NWDS_CALL_BACK	<p>The NWDS_CALL_BACK structure contains the following:</p> <ul style="list-style-type: none"> <li>The information (context) that NetWeave needs for this call.</li> <li>The name of the callback procedure NetWeave will call when the function completes.</li> </ul> <p>If NWDS_CALL_BACK is present when a NetWeave function is called, the call will be made asynchronously.</p>



## Item Lists

NetWeave uses item lists to extend the basic operation of its API. To use the NetWeave API's advanced features, you need to understand how lists work. One of the major tradeoffs for any function call is the desire to simplify the call versus the need to provide access to specific features of a particular hardware vendor's operating system. NetWeave uses item lists to specify system-specific parameters within the context of a simple API. There are two kinds of item lists:

List type	Description
Control item list	Contains the parameters and values that change how the function will be performed on the target system.
Return item list	Contains the parameters for which values will be returned when the call completes on the target system.

An item list is a variable-length array of parameters. A unique type, `NWDS_END_OF_LIST`, identifies the last element. Each element (item) in the array has three components:

Component	Description
Type	A constant from <code>netweave.h</code> that identifies a parameter (the name of the parameter).
Length	The length of the parameter value. Most parameters are either 16-bit integers ( <code>NWDS_SHORT</code> ) or 32-bit integers ( <code>NWDS_LONG</code> ). Variable-length parameters are considered to be of type <code>NWDS_CHAR</code> . For a return item list, the length is the maximum number of bytes that can be copied to the destination location.
Pointer to value	For a control item list, this is the address of the location in memory where the value of the parameter is stored. For a return item list, this is the address in which to store the returned value.

**NOTE:** Most NetWeave functions contain at least one parameter for an item list. Where possible, you should use the control parameter default values to avoid using an item list altogether. To indicate that there is no item list, set the associated `itemlist` parameter to `NULL`.

The bulk of `netweave.h` consists of definitions of item types for specific hardware vendors and/or NetWeave functions. Whereas the item types for queued messaging and data translation apply across all platforms, the item types for data server calls are unique by hardware vendor. The names of platform-specific item types and enumerated values are chosen to match a parameter and value on the target system. It is assumed that a programmer using data server calls has access to the system reference manuals for the target platform for obtaining more specific information.

For languages such as COBOL85 that cannot assign addresses in memory, NetWeave supplies several functions for loading item lists. At the other end of the spectrum, C programmers have a choice of several methods for loading an item list. The sample code below shows how to build a control item list to pass to `nwds_file_open` to open a FIFO for standard processing.

```

char          *fname;
NWDS_HANDLE   handle;
NWDS_ITEM_LIST  ilist[3];
short        file_type_fifo = NWDS_FILE_TYPE_FIFO;
short        fifo_read_new_pos = NWDS_FIFO_READ_NEW_POS;
NWDS_ERRNO    status = NWDS_SUCCESSFUL;

/* identify the file to open... */
/* load the item list */
status = nwds_item_load_short (ilist, 0, NWDS_FILE_TYPE,
                              &file_type_fifo);
if (status != NWDS_SUCCESSFUL) {
    /* error recovery... */
}
status = nwds_item_load_short (ilist, 1, NWDS_FIFO_SHARING,
                              &fifo_read_new_pos);
if (status != NWDS_SUCCESSFUL) {
    /* error recovery... */
}
status = nwds_item_load_short (ilist, 2, NWDS_END_OF_LIST, NULL);
if (status != NWDS_SUCCESSFUL) {
    /* error recovery... */
}

status = nwds_file_open (fname, &handle, ilist, NULL);
if (status != NWDS_SUCCESSFUL)
{

```

A more concise and intuitive version of the same snippet:

```

/* initialize the item list */
NWDS_ITEM_LIST  ilist[] =
    {NWDS_FILE_TYPE,      sizeof(short),    &file_type_fifo,
     NWDS_FIFO_SHARING,  sizeof(short),    &fifo_read_new_pos,
     NWDS_END_OF_LIST,   0,                NULL };
status = nwds_file_open (fname, &handle, ilist, NULL);
if (status != NWDS_SUCCESSFUL) {


```

The call to `nwds_file_info` below shows how to use a return item list to determine how many records are in a particular queue:

```


#define          fname "FIFO-1"
short          file_type_fifo = NWDS_FILE_TYPE_FIFO;
long           number_of_fifo_records = 0;
NWDS_ERRNO     status = NWDS_SUCCESSFUL;

```



```
/* define and initialize the control item list */
NWDS_ITEM_LIST control_list [] =
    {NWDS_FILE_TYPE,          sizeof(short),    &file_type_fifo,
      NWDS_END_OF_LIST,      0,          NULL};
/* define and initialize the return item list */
NWDS_ITEM_LIST return_list [] =
    {NWDS_FIFO_NUMBER_RECORDS, sizeof(long),    &number_of_fifo_records,
      NWDS_END_OF_LIST,      0,          NULL};

status = nwds_file_info (fname, control_list, return_list, NULL);
if (status != NWDS_SUCCESSFUL) {
```



## Programming Tips: Synchronous Calls

NetWeave functions may be called either *synchronously* or *asynchronously*, depending on how you want the function call to complete. An asynchronous operation does not complete upon return to the caller. Instead, it returns a status code of `NWDS_PENDING`, and will at another point invoke a caller-specified callback function with the completion status.

**NOTE:** It is possible that the callback function may have been called already when the original call returns `NWDS_PENDING`.

In contrast, a synchronous operation completes within the context of the call: when the function returns, the operation is completed. During the operations associated with the function call (some of which occur on a different computer), your program is suspended. It has to wait for each synchronous call to complete before it continues with the next operation.

A synchronous program uses only synchronous function calls. Synchronous programs are easier to read, maintain, and debug because the sequence of operations tends to match the steps in the design algorithm (the pseudo-code). Most NetWeave function calls contain a parameter for at least one callback function. To perform an operation synchronously, set the callback parameter to `NULL`.

**NOTE:** In the previous section, the examples of how to specify item lists use synchronous calls.

The main disadvantage of a synchronous server application is poor scalability. Because a synchronous server processes each request serially, it can service only one external connection at a time. To overcome the scalability limitation while preserving the simplicity of synchronous coding, NetWeave created the Dispatcher. For more information about the Dispatcher, see “Threaded Services” on page 35.

### Example #1: Synchronous Calls in a Client-Server Environment

This example shows how a synchronous client application connects to a synchronous, single-threaded server application on another computer. (A synchronous client may have simultaneous connections with more than one server; a single-threaded server services only one client at a time.)


The server begins the connection sequence by calling `nwds_ipc_publish`. One of the call parameters is the public name to which the client will connect. This public name identifies the INI file group that must match the corresponding name and INI group parameters in the client's call to `nwds_ipc_connect`. When the `nwds_ipc_publish` call returns, the server program receives a handle for its public connection.

Next, the server uses the public handle to call `nwds_ipc_accept` and waits for a connection from a remote client. When `nwds_ipc_accept` completes, the server application receives another “private” handle that uniquely identifies the conversation with this remote client.

```
#define          server          "SERVER"

NWDS_ITEM_LIST      ilist[] =  { NWDS_END_OF_LIST,  0,  NULL };
NWDS_HANDLE         public_handle;
NWDS_HANDLE         private_handle;

status = nwds_ipc_publish( server,
                          &public_handle,
```



```
        ilist,
        NULL,
        NULL );

if ( status != NWDS_SUCCESSFUL ) {
    /* error recovery... */
}

status = nwds_ipc_accept( public_handle,
                          &private_handle,
                          ilist,
                          NULL,
                          NULL );

if ( status != NWDS_SUCCESSFUL ) {
    /* error recovery... */
}
```

When the server application reaches the point where it is waiting on its `nwds_ipc_accept` call, the remote client can connect to it. The client issues `nwds_ipc_connect` with the public name of the server. When the connection is successfully established, the client receives a private handle that identifies the conversation with this particular server.

```
#define          server          "SERVER"
NWDS_HANDLE ipc_handle;

status = nwds_ipc_connect( server,
                          &ipc_handle,
                          NULL,
                          NULL,
                          NULL );

if( status != NWDS_SUCCESSFUL ) {
```

## Example #2: Infrastructure

This example illustrates several of the activities that support the NetWeave API:

Activity	Description
Initialization	Calls <code>nwds_init</code> before any other functions in the API.
Cleanup	Calls <code>nwds_exit</code> to terminate all use of NetWeave resources.
Suspending operation	Calls <code>nwds_sleep</code> to suspend for a specific interval.
Decoding errors	Calls <code>nwds_error_text</code> to translate return codes to text strings.
Recovering from network failures	Puts it all together.

The numbers in parentheses to the right refer to additional comments, which appear at the end of this code example.

```
#include "netweave.h" (1)

#define server "SERVER"
#define ini_file "netweave.ini" (2)
#define ini_group "CLIENT" (2)
static NWDS_HANDLE ipc_handle;
NWDS_MILLISECONDS wait_interval = 5000; (7)

void oops (NWDS_ERRNO, char *); (3)

int main( void )
{
    NWDS_ERRNO status;

    if( NWDS_SUCCESSFUL != nwds_init(ini_file, ini_group)) { (2)
        printf("can't open ini,group (%s,%s)\n", ini_file, ini_group);
        return 0;
    }

    for (;;) { (4)
        printf("before nwds_ipc_connect\n");
        status = nwds_ipc_connect( server,
                                &ipc_handle,
                                NULL,
                                NULL,
                                NULL );

        if( status != NWDS_SUCCESSFUL && (5)
            status != NWDS_LINK_DOWN ) {
            oops(status, "nwds_ipc_connect failed");
        }
    }
}
```

```

        (void)nwds_exit();
        return 0;
    }

    if ( status == NWDS_LINK_DOWN ){
        oops( status, "trying again...");
        (void)nwds_sleep ( wait_interval, NULL, NULL);
        continue;
    }
    printf("nwds_ipc_connect successful\n");

/* All preliminaries are complete; now get to work. */

/* Work is all done; shut 'er down. */

    status = nwds_ipc_shutdown( ipc_handle,
                                NULL,
                                NULL );
    if( status != NWDS_SUCCESSFUL ) {
        oops(status, "nwds_ipc_shutdown failed");
    }
    break;
}
printf("success\n");
(void)nwds_exit();
return 0;
}

void oops (NWDS_ERRNO status, char *msg)
{
    #define    MAX_STRING  255
    char    buffer[MAX_STRING];
    int    offset;

    memset( buffer, ' ', MAX_STRING );
    offset = strlen(msg);
    memcpy (buffer, msg, offset++);
    buffer[ offset ] = ':';
    offset += 2;
    nwds_error_text( status, &buffer[offset] );
    printf( "%s\n",buffer );
    return;
}

```

#### Notes for example #2:

1. Every NetWeave application must include `netweave.h`.
2. `Nwds_init` tells NetWeave where to find the runtime configuration parameters.

3. `Oops ( )` is a simple error handler that accepts a message and displays the message with the text of the error message.
4. This loop controls the client's attempts to connect to the server. If the connection fails, the program pauses for five seconds and retries the connection.
5. A test for fatal errors.
6. A test for a failed connection.
7. There are four ways to use `nwds_sleep`. Usually the program pauses for a specified interval (measured in milliseconds).
8. To release NetWeave resources, call `nwds_exit`.
9. To convert a return code to a text message, use `nwds_error_text`.

### Example #3: A Simple Distributor for Queued Messages

This example shows a simple synchronous process called a Distributor. For the purposes of this example, let's assume that:

- Multiple producer processes on remote platforms all write to a common queue on your system.
- There are multiple consumer processes on your system, each with its own queue for input.

The Distributor's job is to consume the common input queue and then assign each message to the right consumer.

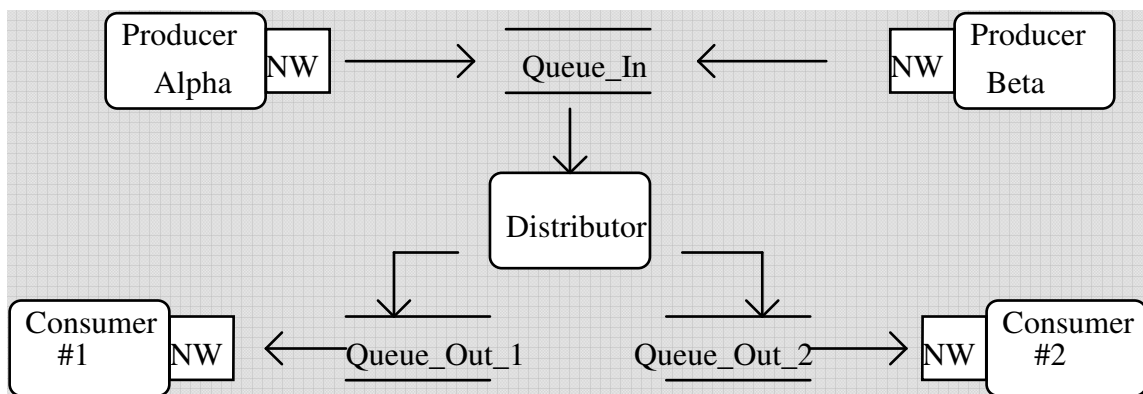


Figure 17. Synchronous message distribution example program

In the main loop, the Distributor reads the input queue, calls the application-specific routine `which_one` to decide where to write the message, and then writes the message to the appropriate queue. The main routine handles housekeeping details such as calling `nwds_init` and `nwds_exit`.

The function `fifo_read` checks the input queue. If the queue is empty, `fifo_read` sleeps. It does not return until it either reads a message or encounters a fatal error.



The routine `open_all_queues` defines which queues should be opened. In this simple example the queue names are hard-coded, but in a more robust example they would be externally defined and passed into the application.

```
#include <stdlib.h>
#include <stdio.h>
#include netweave.h

#define FIFO_OPEN_FOR_WRITE 0
#define FIFO_OPEN_FOR_READ 1

#define ini_group      "DEMO"
#define ini_file       "nwds.ini"
#define                max_message_size      500

NWDS_HANDLE          queue_in;
NWDS_HANDLE          queue_out_1;
NWDS_HANDLE          queue_out_2;

NWDS_SIZE            message_size;
char                 fifo_message[ max_message_size ];
char                 buffer[ NWDS_MAX_STRING ];

NWDS_ERRNO fifo_open (char *fname, NWDS_HANDLE *handle, int how)
{
    NWDS_ERRNO        status = NWDS_SUCCESSFUL;
    NWDS_ITEM_LIST    ilist[ 4 ];
    short             file_type_fifo    = NWDS_FILE_TYPE_FIFO;
    short             fifo_append_only  = NWDS_FIFO_APPEND_ONLY;
    short             fifo_read_new_pos = NWDS_FIFO_READ_NEW_POS;

    status = nwds_item_load_short (ilist, 0, NWDS_FILE_TYPE,
    &file_type_fifo);
    if (status != NWDS_SUCCESSFUL) {
        (void)nwds_error_text(status, buffer);
        printf("%s\n", buffer);
        return status;
    }

    if (how == FIFO_OPEN_FOR_WRITE) {
        status = nwds_item_load_short (ilist, 1, NWDS_FIFO_SHARING,
    &fifo_append_only);
    } else {
        status = nwds_item_load_short (ilist, 1, NWDS_FIFO_SHARING,
    &fifo_read_new_pos);
    }
    if (status != NWDS_SUCCESSFUL) {
        (void)nwds_error_text(status, buffer);
    }
}
```

```

        printf("%s\n", buffer);
        return status;
    }

    status = nwds_item_load_short (ilist, 2, NWDS_END_OF_LIST, NULL);
    if (status != NWDS_SUCCESSFUL) {
        (void)nwds_error_text(status, buffer);
        printf("%s\n", buffer);
        return status;
    }

    status = nwds_file_open (fname, handle, ilist, NULL);
    if (status != NWDS_SUCCESSFUL) {
        (void)nwds_error_text(status, buffer);
        printf("%s\n", buffer);
    }
    return status;
}

void open_all_queues (void)
{
    NWDS_ERRNO    status;
    char          *q1 = {"QUEUE_IN"};
    char          *q2 = {"QUEUE_OUT_1"};
    char          *q3 = {"QUEUE_OUT_2"};

    status = fifo_open (q1, &queue_in, FIFO_OPEN_FOR_READ);
    if (status != NWDS_SUCCESSFUL) {
        printf ("main failed to open %s\n", q1);
        exit(-1);
    }

    status = fifo_open (q2, &queue_out_1, FIFO_OPEN_FOR_WRITE);
    if (status != NWDS_SUCCESSFUL) {
        printf ("main failed to open %s\n", q2);
        exit(-1);
    }

    status = fifo_open (q3, &queue_out_2, FIFO_OPEN_FOR_WRITE);
    if (status != NWDS_SUCCESSFUL) {
        printf ("main failed to open %s\n", q3);
        exit(-1);
    }
}

```

```

NWDS_ERRNO fifo_write (NWDS_HANDLE handle)
{
    NWDS_ERRNO    status;

```



```

status = nwds_file_write (handle, message_size, fifo_message,
                          NULL, NULL);
if (status != NWDS_SUCCESSFUL) {
    (void)nwds_error_text(status, buffer);
    printf("%s\n", buffer);
}
return status;
}

```

```

NWDS_ERRNO fifo_read (NWDS_HANDLE handle)
{
NWDS_ERRNO status;
NWDS_MILLISECONDS      wait_time = 60000;      /* 1 minute */

    memset(fifo_message, 0, max_message_size);

    for (;;) {

        status = nwds_file_read (handle, max_message_size,
                                fifo_message, &message_size, NULL,
                                NULL, NULL);
        if (status != NWDS_SUCCESSFUL &&
            status != NWDS_EOF) {
            (void)nwds_error_text(status, buffer);
            printf("%s\n", buffer);
            return status;
        }
        if (status == NWDS_SUCCESSFUL)
            return status;

        (void) nwds_sleep( wait_time, NULL, NULL);
    }
}

/**
 *   application logic goes here:
 *   return TRUE  if you want the message to go to queue_out_1
 *   return FALSE if you want the message to go to queue_out_2
 */
int which_one( void )
{
    return TRUE;
}

```

```

void close_all_queues (void)
{

```

```
NWDS_ERRNO status;
```

```

status = nwds_file_close (queue_in, NULL, NULL);
if (status != NWDS_SUCCESSFUL) {
    (void)nwds_error_text(status, buffer);
    printf("Queue_IN: %s\n", buffer);
}
status = nwds_file_close (queue_out_1, NULL, NULL);
if (status != NWDS_SUCCESSFUL) {
    (void)nwds_error_text(status, buffer);
    printf("Queue_Out_1: %s\n", buffer);
}
status = nwds_file_close (queue_out_2, NULL, NULL);
if (status != NWDS_SUCCESSFUL) {
    (void)nwds_error_text(status, buffer);
    printf("Queue_Out_2: %s\n", buffer);
}
}

```

```
int main (void)
```

```

{
NWDS_ERRNO status;

status = nwds_init(ini_file, ini_group);
if (status != NWDS_SUCCESSFUL){
    (void)nwds_error_text(status, buffer);
    printf("nwds_init failed: %s\n", buffer);
    exit(-1);
}

open_all_queues();

for (;;) {
    if( NWDS_SUCCESSFUL != fifo_read (queue_in) )
        break;
    if( which_one ){
        if( NWDS_SUCCESSFUL != fifo_write (queue_out_1) )
            break;
    } else {

        if( NWDS_SUCCESSFUL != fifo_write (queue_out_2) )
            break;
    }
}
close_all_queues();
(void)nwds_exit();
exit(0);
} /* main */

```

## Programming Tips: Asynchronous Calls

A NetWeave call is asynchronous if it uses a callback structure. An asynchronous call consists of two parts:

- An initiating call, which passes the application's request and parameters to an asynchronous service. When the initiating call returns, the programmer knows only that the called function has started properly.
- An event, such as the receipt of a message from another process. This event causes NetWeave to call your callback procedure. Thus, the callback function completes sometime during or after the initiating function call has completed.

### Context

Context is all the information required to continue the logical thread of a process. Because context is what ties together the two parts of the asynchronous call, the context information must be provided in a form that will survive from the completion of the initiating procedure to the calling of the callback procedure. To ensure the integrity of the context information, do one of the following:

- Malloc a structure from the heap and populate it before the initiating call (the preferred method).
- Keep the information in static global variables.

The `netweave.h` file includes the following typedefs:

```
typedef void      (NWDS_CALL_BACK_PROC) (NWDS_CONTEXT, NWDS_ERRNO);
typedef struct {
    NWDS_CONTEXT      context;
    NWDS_CALL_BACK_PROC *procedure;
} NWDS_CALL_BACK;
```

A callback structure consists of two parts:

- A pointer to context (a single pointer to the information that holds the logic of your call together). This pointer should point to a structure malloced from the heap.
- A function pointer of type `NWDS_CALL_BACK_PROC`.

You pass context to NetWeave using the callback structure, and NetWeave passes the context back to you as the first parameter of the callback function.



## Waiting for Events to Happen

`Nwds_sleep` is one of the most powerful and complex functions within the NetWeave API. This function allows event-driven asynchronous applications to wait in a variety of ways for new events to occur, and/or allows the application to wake up at specified intervals to determine whether processing is proceeding as it should. For more information about using the sleep functions to wait for events, see page 16.

## Associating Events with Callbacks

A program design is called *event-driven* when certain events initiate and then determine the flow of the program. NetWeave can use either of the following to associate events and callbacks:

- The API's callback parameter
- Platform-specific (kernel) functions

Each NetWeave function represents a NetWeave event, usually either the delivery of a message, or a service such as a database update. In an asynchronous program, several events may be occurring simultaneously. By pointing to the callback structure when you make a call, you associate the event that the call causes with your callback procedure. When an event finishes, NetWeave passes your original context and the completion status to your callback procedure, and you continue processing.

Sometimes you may want to wait not only on NetWeave events, but also on external events about which NetWeave knows nothing – typically, asynchronous events such as keyboard or mouse inputs. These types of external events affect only the client applications that interact with human users. But asynchronous operations such as database updates, I/O calls to other devices, or status events from the operating system can occur in the server environment as well.

To register these types of external events with the NetWeave kernel, use the platform-specific function `nwds_<platform_id>_define_event`. To remove events from NetWeave's kernel, use the analogous function `nwds_<platform_id>_clear_event`. Removing an event means that NetWeave will no longer wait for that event.

The following platforms have specific functions for defining events and their associated actions:

- Tandem
- Unix
- DEC/VMS/OpenVMS
- IBM MVS/CICS

## TANDEM

On the Tandem, you can associate an event with a callback either once (the default) or permanently. Because the syntax of the Tandem kernel callback procedure matches the parameters of the Guardian OS function `awaitiox`, NetWeave can transfer control directly from the completion of `awaitiox` to a user's procedure without loss of information.

NetWeave for Tandem also has a mechanism for notifying the application about certain system events. The function `define_system_event` associates a kernel callback procedure with receipt of a particular system message (the event), usually a system open or close message.

```
#define NWDS_PERMANENT          0x0010
#define NWDS_NOT_PERMANENT     0x0000

typedef void (NWDS_KERNEL_CALL_BACK_PROC) (short,
                                           void *,
                                           short,
                                           short,
                                           long,
                                           void * );

typedef struct {
    NWDS_CONTEXT                user_context;
    NWDS_KERNEL_CALL_BACK_PROC *procedure;
} NWDS_KERNEL_CALL_BACK;

extern __CPP NWDS_ERRNO NWDS_TANDEM_DEFINE_EVENT (
    short          filenumber,
    long           tag,
    long           timeout,
    short          permanent,
    NWDS_ITEM_LIST *item_list,
    NWDS_KERNEL_CALL_BACK *call_back );

extern __CPP NWDS_ERRNO NWDS_TANDEM_DEFINE_SYSTEM_EVENT (
    short          system_message_number,
    short          permanent,
    NWDS_ITEM_LIST *item_list,
    NWDS_KERNEL_RECV_CALL_BACK *call_back);

extern __CPP NWDS_ERRNO NWDS_TANDEM_CLEAR_EVENTS (short          filenumber,
                                                  long           tag,
                                                  short          error
);

extern __CPP NWDS_ERRNO NWDS_TANDEM_CLEAR_SYSTEM_EVENTS (short
system_number,
                  NWDS_KERNEL_RECV_CALL_BACK *call_back);
```

## UNIX

For UNIX operating systems, NetWeave associates a callback function with a file handle such as a raw socket. This allows the application to integrate NetWeave with private mechanisms that use asynchronous events on file handles to indicate when to return control to the application. Because NetWeave avoids using signals, an application may use signals to return control from asynchronous events without interfering with NetWeave's operation.

```
typedef short (NWDS_UX_KERNEL_CALL_BACK_PROC) (NWDS_CONTEXT, IO_TYPE);
typedef struct {
    NWDS_CONTEXT          context;
    NWDS_UX_KERNEL_CALL_BACK_PROC *procedure;
} NWDS_UX_KERNEL_CALL_BACK;

extern __CPP NWDS_ERRNO nwds_ux_clear_event (
    int          socket_descr,
    int          read_mask,
    int          write_mask    );
extern __CPP NWDS_ERRNO nwds_ux_define_event (
    int          socket_descr,
    int          permanent,
    NWDS_UX_KERNEL_CALL_BACK *read_completion,
    NWDS_UX_KERNEL_CALL_BACK *write_completion );
```

## Digital Equipment Corp., VMS and OpenVMS

On DEC Alpha and VAX/VMS platforms, NetWeave lets you associate a callback with a VMS event flag. An event flag can be either clear or set (the event is assumed to have occurred). NetWeave waits only for local event flags in the range 32 to 63.

```
NWDS_ERRNO nwds_vms_define_event (int efn, NWDS_CALL_BACK *completion);
NWDS_ERRNO nwds_vms_clear_event (int efn);
```

## Microsoft Windows NT, Windows Millennium

On Windows platforms, NetWeave let you associate a callback with a Win32 handle such as a raw socket descriptor. NetWeave uses the Windows function `waitForMultipleObjects` to coordinate asynchronous operations.

```
extern __CPP NWDS_ERRNO nwds_nt_clear_event (long hEvent);
extern __CPP NWDS_ERRNO nwds_nt_define_event (
    long          hEvent,
    NWDS_CALL_BACK *completion);
```



## Asynchronous Server: Case Study

This example shows the implementation of a simple server program using NetWeave API functions in asynchronous mode. On the Tandem platform, this file is known as `aserverc`. All other platforms refer to it as `aserver.c`. You can find this and other useful sample programs discussed in this document at <ftp://www.vertexinteractive.com/middleware/code/CodeExamples.zip>.

The `aserver` program is implemented using callbacks (i.e., it is asynchronous), and it is somewhat limited because it can process only one connection at a time. To enable `aserver` to handle multiple concurrent sessions, unique `MESSAGE_CONTEXT` structures have to be allocated from dynamic memory instead of the single static data structure in effect now. Because `aserver` uses only NetWeave functions to associate events with callbacks, events in this program are NetWeave events, as opposed to being external O/S-specific events described in previous sections. The events describe a cascade of actions that form a connection to a remote client, and then wait for messages from the client.

There are two callback structures associated with each of the three calls (`nwds_ipc_publish`, `nwds_ipc_accept`, and `nwds_ipc_connect`) required to set up a new call. Each NetWeave function has a completion callback, while a few functions (notably `nwds_ipc_publish` and `nwds_ipc_accept`) have both completion and notification callbacks.

The completion callback is similar to that used in other NetWeave functions. NetWeave calls this when the event associated with the function has completed. Thus, after NetWeave creates the public name in the communications network, the completion callback reports that the server program is ready to receive calls.

The notification callback is crucial for effective IPC messaging. For `nwds_ipc_publish`, the notification callback is called when NetWeave detects a new call. For `nwds_ipc_accept` and `nwds_ipc_connect`, the notification callback is called when the `aserver` application receives a new message.

### Road Map for ASERVER

This section describes each `aserver` section, beginning with `main()`. Like most asynchronous programs, `aserver` performs its initialization steps and then calls `nwds_sleep()`, which allows all subsequent processing to occur in the series of callback procedures that were set up during initialization.

1. The main routine calls `start_publish` and then waits for events to occur by calling `nwds_sleep()`. All subsequent events flow from `start_publish`.
2. `start_publish` calls `nwds_ipc_publish` using a completion callback and a notification callback called `connect_received`. This callback will be called when the program connecting to `aserver` issues `nwds_ipc_connect` to establish a connection with the corresponding NetWeave connection group.
3. During the final call setup step, `connect_received` calls `nwds_ipc_accept` using both a completion callback and a notification callback. The status that is returned to `connect_received` indicates whether the call setup was concluded successfully. When data is available for `nwds_ipc_read()` to read, the notification callback `process_request()` is called.
4. Whenever `aserver` receives a new message from the client, the NetWeave kernel notifies `process_request`. Within `process_request`, a call to `nwds_ipc_options` returns (among

other things) the size of the incoming message and the sender. The `nwds_ipc_options` call is synchronous because data is already available to return to the program. The same is true for the next call, `nwds_ipc_read()`, which dequeues the message from one of NetWeave's internal queues.

5. If the destination application has already read the message, a write completion (in an asynchronous program, this means when the NetWeave kernel calls the `write_callback`) is returned.

**NOTE:** The aserver code is reproduced in its entirety. Additional editorial comments appear in **bold text**.

```

/**
 *
 * This server is asynchronous. It receives a message, processes it
 * and writes the reply to the originator.
 *
 * This example uses the following NWDS functions:
 *     nwds_init
 *     nwds_publish
 *     nwds_accept
 *     nwds_ipc_write
 *     nwds_ipc_read
 *     nwds_ipc_shutdown
 *     nwds_exit
 *     nwds_error_text
 *
 */

#include "netweave.h"

#define MAX_MSG_SIZE  10000
#define INITIAL      10
#define WAITING_FOR_REQUEST  11
#define PROCESSING_REQUEST  12
#define REQUEST_PROCESSED    13

typedef struct {
    short      state;
    char       *buffer;
    NWDS_HANDLE  publish_handle;
    NWDS_HANDLE  accept_handle;
} MESSAGE_CONTEXT;           the "context" that links together the initial
                             phase of a call and its completion phase

void accept_complete ( NWDS_CONTEXT context, NWDS_ERRNO error );
void connect_received ( NWDS_CONTEXT context, NWDS_ERRNO error );

```

```

void publish_complete ( NWDS_CONTEXT context, NWDS_ERRNO error );
void start_publish ( char *publish_name, MESSAGE_CONTEXT *msg_context );

void client_write_complete ( NWDS_CONTEXT context, NWDS_ERRNO error );
void process_request ( NWDS_CONTEXT context, NWDS_ERRNO error );

void shutdown_gracefully (MESSAGE_CONTEXT *);

int main ( int argc, char *argv[] )
{
    NWDS_ERRNO      status;
    char            ini_file[MAX_STRING];
    char            ini_group[MAX_STRING];
    char            server_name[MAX_STRING];
    char            publish_name[MAX_STRING];
    int             error;
    MESSAGE_CONTEXT *msg_context;

    if (argc < 4) {
        printf("Error:: Improper parameters passed at runtime\n");
        printf("Syntax:: SERVER <ini_file> <group_name> <publish_name>\n");
        exit(0);
    }

    strcpy( ini_file, argv[1] );
    strcpy( ini_group, argv[2] );
    strcpy( publish_name, argv[3] );

    printf("%s initializing\n", ini_group);

    msg_context = (MESSAGE_CONTEXT *)malloc(sizeof(MESSAGE_CONTEXT));
    if (msg_context == NULL) {
        printf("Server unable to malloc MESSAGE_CONTEXT");
        return TRUE;
    }

    msg_context->buffer = malloc(MAX_MSG_SIZE);
    if (msg_context->buffer == NULL) {
        printf("Server unable to malloc MESSAGE_CONTEXT->BUFFER");
        return TRUE;
    }

    msg_context->publish_handle = NULL;
    msg_context->accept_handle = NULL;
    msg_context->state = INITIAL;

    /**

```

```

*
* We first initialize the NetWeave library
*
***/

if (NWDS_SUCCESSFUL != nwds_init(ini_file, ini_group)) {
    printf("Server init failed with <ini_file>,<ini_group> (%s,%s)\n",
ini_file, ini_group);
    exit(0);
}

printf("Server Issuing NWDS_IPC_PUBLISH for %s\n", ini_group);

start_publish( publish_name, msg_context ); everything starts here

nwds_sleep(0xFFFFFFFF, NULL, NULL); sleep a very long time
}

    accept_complete marks the end of the call setup phase
    it changes the context's "state" to WAITING_FOR_REQUEST
void accept_complete ( NWDS_CONTEXT context, NWDS_ERRNO error )
{
    MESSAGE_CONTEXT *msg_context = (MESSAGE_CONTEXT *)context;

    if (error != NWDS_SUCCESSFUL) {
        printf("FATAL Server: NWDS_IPC_ACCEPT error, returned error (%d)\n",
error);
        shutdown_gracefully(msg_context);
        return;
    }

    msg_context->state = WAITING_FOR_REQUEST;
}

void connect_received ( NWDS_CONTEXT context, NWDS_ERRNO error )
{
    NWDS_ERRNO      status;
    NWDS_ITEM_LIST  ilist[1];
    NWDS_CALL_BACK  completion_callback;
    NWDS_CALL_BACK  data_received_callback;
    MESSAGE_CONTEXT *msg_context = (MESSAGE_CONTEXT *)context;

    if (error != NWDS_SUCCESSFUL) {
        printf("FATAL Server: Error (%d) on publish handle\n", error);
        shutdown_gracefully(msg_context);
        return;
    }
}

```

```

}
/**
 *
 * We now wait for someone to connect to us
 *
 */
ilist[0].type = NWDS_END_OF_LIST;
completion_callback.context      = msg_context;
completion_callback.procedure    = accept_complete;
data_received_callback.context   = msg_context;      notification callback
data_received_callback.procedure = process_request;

printf("Server Issuing NWDS_IPC_ACCEPT\n");
status = nwds_ipc_accept( msg_context->publish_handle,
                          &msg_context->accept_handle,
                          ilist,
                          &completion_callback,
                          &data_received_callback );

if ((status != NWDS_SUCCESSFUL) && (status != NWDS_PENDING)) {
    printf("FATAL Server: NWDS_IPC_ACCEPT error, returned status (%d)\n",
status);
    shutdown_gracefully(msg_context);
    return;
}
printf("Server NWDS_IPC_ACCEPT issued successfully\n");
}

void publish_complete ( NWDS_CONTEXT context, NWDS_ERRNO error )
{
    MESSAGE_CONTEXT *msg_context = (MESSAGE_CONTEXT *)context;

    if (error != NWDS_SUCCESSFUL) {
        printf("FATAL Server: NWDS_IPC_ACCEPT error, returned error (%d)\n",
error);
        shutdown_gracefully(msg_context);
        return;
    }
}

void start_publish ( char *publish_name, MESSAGE_CONTEXT *msg_context )
{
    NWDS_ERRNO      status;
    NWDS_ITEM_LIST  ilist[1];
    NWDS_CALL_BACK  completion_callback;
    NWDS_CALL_BACK  call_received_callback;

```

```

/**
 *
 * We publish so that requests can be received
 *
 */

ilist[0].type = NWDS_END_OF_LIST;

completion_callback.context      = msg_context;
completion_callback.procedure    = publish_complete;
call_received_callback.context  = msg_context;      notification callback
call_received_callback.procedure = connect_received;defined here

status = nwds_ipc_publish( publish_name,
                           &msg_context->publish_handle,
                           ilist,
                           &completion_callback,      completion CB
                           &call_received_callback );  notification CB

if ((status != NWDS_SUCCESSFUL) && (status != NWDS_PENDING)) {
    printf("Server NWDS_IPC_PUBLISH error, returned status (%d)\n", status);
    shutdown_gracefully(msg_context);
    return;
}

printf("Server NWDS_IPC_PUBLISH issued successfully\n");

}

void client_write_complete ( NWDS_CONTEXT context, NWDS_ERRNO error )
{
    MESSAGE_CONTEXT *msg_context = (MESSAGE_CONTEXT *)context;

    if (error != NWDS_SUCCESSFUL) {
        printf("Server NWDS_IPC_WRITE to client error, returned error (%d)\n",
error);
    }

    msg_context->state = WAITING_FOR_REQUEST;

}

void process_request ( NWDS_CONTEXT context, NWDS_ERRNO error )
{
    NWDS_ERRNO      status;
    NWDS_ITEM_LIST  ilist[4];
    char            request[MAX_STRING];
    NWDS_SIZE       return_size;

```

```

long          queue_count;          for ipc_options
long          message_size;
char          address[MAX_STRING];
NWDS_CALL_BACK completion_callback;
MESSAGE_CONTEXT *msg_context = (MESSAGE_CONTEXT *)context;

if (error != NWDS_SUCCESSFUL) {
    if (error == NWDS_LINK_DOWN) {
        printf("Server Client shutdown connection - awaiting new
connection\n");
    } else {
        printf("Server NWDS_IPC_READ on accept_handle error, returned status
(%d)\n", status);
    }
    status = nwds_ipc_shutdown( msg_context->accept_handle,
                                NULL,
                                NULL
                                );

    if (status != NWDS_SUCCESSFUL) {
        printf("FATAL Server: NWDS_IPC_SHUTDOWN on accept_handle failed,
returned status (%d)\n",
            status);
        shutdown_gracefully( msg_context );
        return;
    }
    msg_context->accept_handle = NULL;
    return;
}

/**
 *
 * Call NWDS_IPC_OPTIONS to retrieve info
 *
 */
status = nwds_item_load_long( ilist,
                              0,
                              NWDS_IPC_QUEUE_COUNT,
                              &queue_count
                              );

if (status != NWDS_SUCCESSFUL) {
    shutdown_gracefully( msg_context );
    return;
}

status = nwds_item_load_long( ilist,
                              1,
                              NWDS_IPC_MESSAGE_SIZE,
                              &message_size
                              );

if (status != NWDS_SUCCESSFUL) {

```

```

        shutdown_gracefully( msg_context );
        return;
    }

    status = nwds_item_load_char( ilist,
                                  2,
                                  NWDS_IPC_ADDRESS,
                                  MAX_STRING,
                                  address
                                  );

    if (status != NWDS_SUCCESSFUL) {
        shutdown_gracefully( msg_context );
        return;
    }

    ilist[3].type = NWDS_END_OF_LIST;

    status = nwds_ipc_options( msg_context->accept_handle,
                               ilist
                               );

    printf("Message from Client\n");
    printf("  QUEUE_COUNT      = %ld\n", queue_count );
    printf("  MESSAGE_SIZE      = %ld\n", message_size );
    printf("  ADDRESS              = %s\n", address );

    /**
     *
     * We read the request from the client
     *
     */

    ilist[0].type = NWDS_END_OF_LIST;

    status = nwds_ipc_read( msg_context->accept_handle,
                            MAX_MSG_SIZE,
                            msg_context->buffer,
                            &return_size,
                            ilist
                            );

    if (status != NWDS_SUCCESSFUL) {
        printf("Server NWDS_IPC_READ on accept_handle error, returned status
(%d)\n", status);
        shutdown_gracefully( msg_context );
        return;
    } else {
        printf("\nServer - Request of %d bytes\n", return_size);

        msg_context->state = PROCESSING_REQUEST;

```





```

    /***
    * The call to the procedure to process the request goes here
    ***/

    ilist[0].type = NWDS_END_OF_LIST;

    completion_callback.context      = msg_context;
    completion_callback.procedure    = client_write_complete;

    status = nwds_ipc_write( msg_context->accept_handle,
                            return_size,
                            msg_context->buffer,
                            ilist,
                            &completion_callback      );

    if (status != NWDS_SUCCESSFUL) {
        printf("FATAL Server: NWDS_IPC_WRITE on accept_handle error, returned
status (%d)\n", status);
        shutdown_gracefully( msg_context );
        return;
    }

    msg_context->state = REQUEST_PROCESSED;

    printf("Server write successful\n");
}

}

void shutdown_gracefully (MESSAGE_CONTEXT *msg_context)
{


    NWDS_ERRNO      status;
    NWDS_ITEM_LIST  ilist[1];

    if (msg_context == NULL) {
        nwds_exit();
        exit(0);
    }

    ilist[0].type = NWDS_END_OF_LIST;

    if (msg_context->publish_handle != NULL) {

```



```
status = nwds_ipc_shutdown( msg_context->publish_handle,
                           ilist,
                           NULL                               );
}

if (msg_context->accept_handle != NULL) {
    status = nwds_ipc_shutdown( msg_context->accept_handle,
                               ilist,
                               NULL                               );
}

if (msg_context->buffer != NULL) {
    free(msg_context->buffer);
}

free(msg_context);

nwds_exit();
exit(0);
}
```





## Prototyping with Synchronous Calls

Objectives of a prototype:

- Demonstrate one call to each of the NetWeave functions you want to use.
- Confirm that you understand which item types and values are required.
- Test your INI file.

Always start with a simple, synchronous prototype of each new application. NetWeave functions are intended to simplify programming. If your prototype takes more than a day or two to construct, it is too complicated. Simplify and start over.

### Mixing Synchronous and Asynchronous Calls

As a rule, the way you define a handle also indicates how you expect NetWeave to use it. For example, if you build a server that performs call setup synchronously, the underlying communications infrastructure expects that your messages will be read by blocking calls to `nwds_ipc_read`. The same is true for file operations. If your call to open a FIFO is synchronous, subsequent reads or writes of the FIFO should be synchronous too.

Because it is so tedious to write asynchronous code, you may be tempted to take shortcuts by calling some of the NetWeave functions synchronously. Although this may work for simple, low volume situations, once applications are ramped up to production volumes, you may start to notice some performance oddities. If you are serious about writing asynchronous applications, be thorough!

### Calls with Timeouts

There is no direct support for timeouts on individual function calls. Whereas in a closed, proprietary environment you can limit the consequences of a timeout, usually the only thing you can do in an open, distributed environment is close a remote file or shut down a conversation. Recovery of either of these can be expensive.

Lack of timeout protection can lead to situations where a synchronous client application may have to wait an unacceptably long time for a response. To avoid this, use either of the following strategies for your synchronous program:

- If your platform supports timers, use system routines to cause a timer event, and define this event and callback to NetWeave.
- Use `nwds_sleep` to poll for the completion of an event.

A timer is a system-specific external function that will cause a particular event (timer interrupt) in your program. You can use a system routine to define a timer, and then use the NetWeave kernel calls to define this timer event to NetWeave. For data server operations that time out, the callback associated with the timer must close the remote file. If a timeout occurs during IPC messaging, the callback must shut down the circuit.

**NOTE:** If you have started a system timer and the event for which you are waiting occurs, remember to stop the timer (if your system supports this), and remove the event from the list that NetWeave monitors.



To *poll for a completion* means to monitor a flag that the completion function sets up when an asynchronous NetWeave function finishes. `Nwds_sleep` is called repeatedly until the flag is set or the timeout interval expires – usually a small number of poll cycles. If the call times out, the program may continue to operate and ignore the eventual completion, or it may force a shutdown to stop the server from further processing.

## Tips on Testing NetWeave Applications

For messaging applications, unit test locally. Most applications (except data server) should be constructed to run on a single platform. For example, if you are doing IPC messaging, construct a simple client or server to interact with your application. Such a test bed is said to run in loopback mode.

Don't skip on error routines. Test the return code of every NetWeave function, and if it returns `NWDS_NOT_SUCCESSFUL` or `NWDS_PENDING`, call `nwds_error_text` to interpret the error.

When you get unexpected results, remember to check the NetWeave log for errors. NetWeave's trace feature can log and print the values of many of the parameters passed in NetWeave calls. If you can duplicate a problem using a short sequence of code, try running the short program with full traces.

## Threaded Dispatcher `dp_pong`: Case Study

Calling the `nwds_dispatcher_create` function starts a threaded Dispatcher. This function call starts the controlling thread called the boss thread, whose primary function is to detect any incoming external calls. The boss thread also starts additional service threads, called worker threads, which house the function that the application supplied to `nwds_create_dispatcher()`.

When a boss thread is started, it publishes the internal and external connection group names that were provided as parameters in the API. It then waits for an incoming call on the external name. When an external connection is established, the boss thread starts a worker thread, which in turn connects to the internal publish name. Once this connection is established, the boss thread links the external and internal connections into a single virtual connection.

The worker thread is a user-provided function that must satisfy the following requirements:

- `Nwds_init` must be the first NetWeave API function called in the worker thread. It should be passed `NULL` for both the INI file and INI group parameter. These are process-wide parameters that the main application supplied before it called `nwds_create_dispatcher()`.
- All NetWeave calls within the `worker_thread` function should be synchronous.
- The worker thread must issue a `nwds_ipc_connect` call in a timely manner. The boss thread has its own internal timer that it starts when it creates a worker thread. If the worker thread doesn't connect before this timer expires, the boss thread considers the connection stale and disconnects it.
- Once connected, the worker thread must issue a `nwds_ipc_read` call and wait to receive a message before it can do anything else.
- If a worker thread detects a broken connection or is finished processing, it must issue a `nwds_ipc_shutdown` call and promptly exit, or simply return from the function call.
- Before exiting, the worker thread must issue the `nwds_exit` call.

### A Dispatcher Example

In the example below, the main processing is relatively short. `Nwds_init` is followed by the `nwds_dispatcher_create` call, which provides the external, internal, and control publish names. (These names must be defined in the INI file that is passed as a parameter to the `nwds_init` API function.) When `nwds_dispatcher_create` returns to the caller, the example program enters a loop and spends the next hour gathering statistics from the boss thread. When the program completes this cycle of reporting statistics, it stops the Dispatcher and terminates the program.

**NOTE:** You can find the sample program `dp_pong`, as well as the others mentioned in this document, at <ftp://www.vertexinteractive.com/middleware/code/CodeExamples.zip>. On the Tandem platform, `dp_pong` is known as `dppongc`. All other platforms refer to it as `dp_pong.c`.

```

#include "netweave.h"

/* The main processing routine has two parameters, the INI file and the
Start Group. */
int main(int argc, char *argv[])
{
NWDS_ERRNO   retstat;
NWDS_HANDLE  DHandle;
NWDS_ITEM_LIST  items[2];
NWDS_APPLTHREAD_PROC  worker_thread;

char  *ext_pub_nam = "EXTERNAL";
char  *int_pub_nam = "INTERNAL";
char  *ctrl_pub_nam = "CONTROL";

int   i,  num_thrds = 0, num_msgs = 0;

/* Initialize NetWeave */
retstat = nwds_init(argv[1], argv[2]);

if (retstat != NWDS_SUCCESSFUL)
err_exit("Error doing init", retstat);

/* In this example only the int_pub_nam is passed to the worker thread. In
a typical application other information must also be provided to the worker.
*/
retstat = nwds_dispatcher_create(ext_pub_nam,
int_pub_nam,
ctrl_pub_nam,
int_pub_nam,          /* Information for the worker */
&DHandle,
worker_thread);

if (retstat != NWDS_SUCCESSFUL)
err_exit("Error doing create_dispatcher", retstat);

items[0].type = NWDS_KERNEL_SUSPEND;
items[1].type = NWDS_END_OF_LIST;

for (i = 1; i <= 3600; i++)
{
    retstat = nwds_sleep(1, items, NULL);

    nwds_dispatcher_stats(DHandle, &num_thrds, &num_msgs);
    printf("Threads = %d, Messages = %d\n", num_thrds, num_msgs);
}

```

```

nwds_dispatcher_stop(DHandle);
return 0;
}
/* end of main */

```

Nwds\_init is the first NetWeave function called in the worker thread. Each of its parameters must be NULL. Next, nwds\_ipc\_connect is called to form the internal connection to the boss thread. After the worker thread is connected, it repeatedly receives a request message, processes it, and returns a response. This cycle continues until the external client disconnects.

**NOTE:** When the dialogue is complete, nwds\_exit is the final NetWeave API function called. Nwds\_exit is also called before the application exits because of an error.

```

/* worker thread processing routine */
void worker_thread( void *anypointer )
{
char *int_pub_nam = (char*)anypointer;
NWDS_ERRNO retstat;
NWDS_HANDLE IPCHandle = (NWDS_HANDLE)0;
NWDS_SIZE Retlen;
int j;
char MsgBuffer[MAX_SIZE];
char *SampleReply = {"Here is some text...."};

/* params can be NULL after 1st call to nwds_init in main() */
retstat = nwds_init( NULL, NULL);
if( retstat != NWDS_SUCCESSFUL) return;
printf("new thread started...\n");

/* make connection with boss thread */
nwds_ipc_connect( int_pub_nam, &IPCHandle, NULL, NULL, NULL);
if( retstat != NWDS_SUCCESSFUL)
thrd_err_exit("failed connect",retstat);
printf("Connected...\n");

for (;;) {
/* wait for message */
retstat = nwds_ipc_read( IPCHandle, sizeof(MsgBuffer), MsgBuffer ,&Retlen,
NULL);
if (retstat != NWDS_SUCCESSFUL) {
/* other side shutdown, go out of loop */
if (retstat == NWDS_LINK_DOWN)
break;
else
thrd_err_exit("failed to read request: ",retstat);
}
}

```

```

/* process the request from the remote client application */

/* prepare response message */
strcpy( MsgBuffer, , SampleReply, sizeof(SampleReply) );
retstat = nwds_ipc_write(IPCHandle, MsgBuffer, sizeof(SampleReply) ,NULL,
NULL);
if (retstat != NWDS_SUCCESSFUL) {
/* other side shutdown, go out of loop */
if (retstat == NWDS_LINK_DOWN)
break;
else
thrd_err_exit("failed to write reply: ",retstat);
}
} /* end worker's processing loop */
/* disconnect gracefully */
retstat = nwds_ipc_shutdown( IPCHandle, NULL, NULL);
if( retstat != NWDS_SUCCESSFUL)
    thrd_err_exit("trouble shutting down IPC Handle",retstat);
}
printf(" thread leaving....\n");
nwds_exit();
}

/* error processing routines */
void err_exit(char * text, NWDS_ERRNO errnum)
{
printf("%s : %d\n",text, errnum);
nwds_exit();
}

void thrd_err_exit(char * text, NWDS_ERRNO errnum)
{
printf("%s : %d\n",text, errnum);
nwds_exit();
}

```





## The Windows Versions of the NetWeave DLL

There are two versions of NetWeave for the Win9x/NT/2000 platforms. Both versions are intended for 32-bit operating systems.

### W95

NetWeave W95 is the version of NetWeave for Windows 95/98 that uses the Microsoft Winsock 1.1 library. Existing NetWeave programs for Win9x platforms will work with NetWeave W95, which may be used with any of the Win9x/NT/2000 platforms subject to the conditions described below.

W95 is used for client-side GUI applications based on the traditional Windows Messaging architecture. W95 uses the Winsock 1.1 DLL for TCP/IP sockets. All inputs to an application built with the W95 version must be received via the Windows Message queue. The principal timing function is `SetTime`, and all timing operations are controlled by messages sent to the Windows Message queue.

### NT/2000

NetWeave WinNT/2000 is based on the Windows NT/2000 kernel, and exploits some of the powers unique to the Windows/NT operating system. The NT version is used for multi-threaded, server-side applications. You can use it to build GUI applications as long as you don't use a thread for both the GUI and for NetWeave functions.

The NT version of NetWeave is "thread-safe" and works best with an event-driven programming model. The NT version uses the enhanced interface to sockets, Winsock 2.0, that is compatible with the event-driven models. All inputs to a NetWeave thread are controlled by calls to `WaitForMultipleObjects`, which has a hard limit of 64 concurrent objects. Timers are implemented by a special set of events initiated by `SetWaitableTimer`.

## Contents of the NetWeave for Windows 95 and NT Releases

In addition to the division by platform and operating system between Windows 95 and NT, each customer's licensing agreement determines which files the customer receives. You may purchase a NetWeave license in three different configurations:

- Client-only license: connects applications on a workstation with applications on another NetWeave system.
- Standard (base) server license: adds the NetWeave Agent to the client license components.
- Broadcast server license: adds support for broadcast messaging to the basic Agent.

For more information, please see the table on next page.



Version	License type	File	Description
W95	Client-only	netweave.h	NetWeave header file
		nwds95.dll	Win95/98 dll
		nwds95i.lib	Import library
		nwds95s.lib	Static library
		tst.exe	TST test program (uses nwds95.dll)
	Base server	basesrv.exe	Console base server
		wbasesrv.exe	GUI base server
	Broadcast server	bcastsrv.exe	Console broadcast server
		wcastsrv.exe	GUI broadcast program
	NT	Client only	netweave.h
nwdsnt.dll			NT dll
nwdsnti.lib			Import library
nwdsnts.lib			Static library
tst.exe			TST test program (uses nwdsnt.dll)
Base server		basesrv.exe	NetWeave Agent as a console program
		basesvc.exe	NetWeave Agent as NT service
Broadcast server		bcastsrv.exe	Console broadcast server
		bcastsvc.exe	Broadcast program as NT service



## Sample NetWeave Programs

This section describes the sample programs available on the NetWeave ftp site. The text or zip files on the site are organized by hardware platform, language, and NetWeave service (IPC messages, queued messages, local database, etc.). The program archive is constantly expanding as more examples are added for the vast range of applications possible with the NetWeave API for all the major hardware vendors. To see what's new in the sample program archive, please visit the NetWeave website [ftp.netweave.com/middleware](http://ftp.netweave.com/middleware).

All files permanently located on the ftp site are compressed using the PKZIP utility from PKWARE, Inc. If needed, you can make special arrangements to obtain software examples that are uncompressed, or compressed using another common utility (such as uuencode for UNIX).

In this section, the term *make file* refers generically to the procedures for compiling and linking modules into runnable programs (executables). For each hardware vendor, you will find examples of the commands you need on that platform to compile and link the sample programs.

## Accessing the NetWeave ftp Site

Because ftp is a TCP/IP utility, you must have a computer with an Internet connection to access and retrieve the files on the ftp site.

From an FTP Client program:

1. Attach to [ftp.netweave.com](http://ftp.netweave.com) and log on as anonymous.
2. For the password, enter your email address.
3. Select a file transfer mode: binary for zip files, ascii for text files.
4. Change the directory to middleware/code.
5. Look for the file SampleCodeArchive.zip.
6. Using a Web browser, download the following file:  
<http://www.vertexinteractive.com/middleware/SampleCodeArchive.zip>

**NOTE:** To find the sample files that you need, check the location list for the hardware platform and language that you are using. If there are no examples in a particular language for the platform you are using, check the other platforms for samples in this language.



## Code Samples

### General Purpose Examples

These programs demonstrate many basic concepts in a variety of programming languages:

Program	Language	Description
ACLIENTC	C	An asynchronous program that can communicate with either ASERVERC or SSERVERC.
ASERVERC	C	An asynchronous server program. It is one of the examples on page 60.
SCLIENTC	C	A synchronous program that can communicate with either ASERVERC or SERVERC. It is a more complex version of XIPCCC.
SSERVERC	C	A synchronous server program that echoes the request message as the reply.
TST2S	COBOL85	A synchronous program that demonstrates queued messaging services.
TSTIPCS	COBOL85	This program demonstrates IPC messaging. It connects to the server TSTSERSV, sends it a message, and waits for a reply.
TSTSERSV	COBOL85	The companion to TSTIPCS.
XIPCCC	C	A client program that demonstrates IPC messaging and message translation services.
XIPCSC	C	A server program, the companion to XIPCCC.
XPATHCC	C	A client program that sends a message to a Tandem Pathway serverclass and waits for a reply.
XPATHSC	C	A companion for XPATHCC.
XREADC	C	This program illustrates data server services and message translation.
XWRITEC	C	A companion for XREADC, it writes the text records read by XREADC.

**WinNT****NTPongSamp.zip**

The zip file `NTpongsamp.zip` contains a Microsoft Visual C++ 6.0 workspace with three projects, which create the following executables:

Program	Description
anwpong	An asynchronous messaging program with both client and server operations, which pongs messages back and forth across one or more concurrent sessions. Anwpong demonstrates how to use asynchronous function calls.
snwpong	A synchronous version of anwpong, which supports only a single session. Snwpong demonstrates how to use synchronous function calls.
amtpong	A multi-threaded version of anwpong. Amtpong is an example of how to use synchronous function calls with multiple threads.

These simple programs serve as both the client and server component of a ping-pong messaging program. The two components may run either on two different machines, or on the same box in loopback mode. They send a message buffer back and forth, each time adding two bytes to the end of the message to represent the number of loops. Each time the client or server receives a message from its counterpart, the array element at position 'n' is compared with the loop counter to test NetWeave's byte ordering component (the DDL).

By typing the program name, you can make each program return a text display of the command line parameter list. `Snwpong` uses the groups `PONG_CLIENT`, `PONG_SERVER`, depending on whether you are running a client or server. Example test groups `TST1`, `TST2`, `TST3` are provided for use with the other two executables.

The `NTPongSamp.zip` file contains the following components:

File	Description
Testprograms.dsw	Workspace file
anwpong.dsp	anwpong project file
snwpong.dsp	snwpong project file
amtpong.dsp	amtpong project file
netweave.h	Netweave API header
amtpong.h	Header for amtpong.c
anwpong.h-	Header for anwpong.c
anwpong.c	Asynchronous version of pong program
snwpong.c	Synchronous version of pong program
amtpong.c	Multithreaded version of pong program
nwds.ini	Sample INI file



In project settings under C/C++ in the preprocessor category you should set additional include directories to be “..\netweave” for each project. Regardless where you unpack the zip file, make sure you remap the paths to the source files in the workspace.

## NTWindemo.zip

The `NTWindemo.zip` file contains a Microsoft Visual C++ 6.0 workspace with one project that creates the executable `windemo`. `Windemo` illustrates how to write a client application in C that calls the NetWeave API. The `NTWindemo.zip` file contains the following components:

File	Description
<code>windemo.dsw</code>	Workspace file
<code>windemo.dsp</code>	Project file
<code>itemdef.c</code>	Source code files
<code>tables.c</code>	
<code>w_draw.c</code>	
<code>w_files.c</code>	
<code>w_ipc.c</code>	
<code>w_items.c</code>	
<code>w_sql.c</code>	
<code>w_tp.c</code>	
<code>w_util.c</code>	
<code>windemo.c</code>	
<code>windemo.dlg</code>	
<code>windemo.h</code>	Header file
<code>windemo.rc</code>	Resource file
<code>itemdef.h</code>	Header files
<code>netweave.h</code>	
<code>nwds95i.lib</code>	Netweave library
<code>tables.h</code>	Header files
<code>windraw.h</code>	
<code>windlib.h</code>	
<code>windemo.ico</code>	Icon file

To build this project on a local machine, do the following:

1. In project settings under C/C++ in the preprocessor category, add a path to the NetWeave directory that contains `netweave.h`.

2. Under Preprocessor Definitions, define `windemo` to any value.
3. In the Link category, put the library `netweave/nwds95i.lib` under Object/Library modules, assuming that the `netweave` directory is located in a subdirectory called `netweave`.
4. Remap the paths to the source files in the workspace (this will depend on where you unpack the zip file).

## UNIX

This directory contains a makefile (`UnixSampleCode.zip`) suitable for making the following executables:

- `anwpong`
- `snwpong`
- `amtpong`

These files perform the same functions that we described for the WINNT platform. You can mix and match platforms as you experiment with these sample programs. The directory contains the following sample files:

File	Description
Makefile	Unix makefile for <code>anwpong</code> , <code>snwpong</code> , <code>amtpong</code>
<code>amtpong.c</code>	Multithreaded version of pong program
<code>amtpong.h</code>	Header for <code>amtpong.c</code>
<code>anwpong.c</code>	Asynchronous version of pong program
<code>anwpong.h</code>	Header for <code>anwpong.c</code>
<code>netweave.h</code>	Netweave API header
<code>snwpong.c</code>	Synchronous version of pong program
<code>nwds.ini</code>	Sample INI file



## IBM\_CICS\COBOL

File	Description
sampcode.zip	
ca.cob	
camaster.cob	
client.cob	
netweave.cob	
server.cob	
sql.cob	

## Win31

### POWERBUILDER

The NW\_PB4.ZIP archive contains a sample application written for the 16-bit version of Power Builder software, and includes examples of IPC messaging and queued messaging.

### VisualBasic (vb30)

Use the sample files in VBDEMO.EXE to build a simple client application in the 16-bit Windows environment using Visual Basic.





## TANDEM

TANDEM\cfile\T1.ZIP Data server

This collection of files shows how to build and run a simple C program that reads and writes C stream files (flat files) on any platform with a NetWeave Agent.

File	Description
BINDIN	Input to the Binder; called from MAKE1IN.
MAKE1IN	Input to TACL; called from MAKECF1.
MAKECF1	A makefile for Tandem that compiles and binds TANCC.
NWDSINI	An example that shows how to use the NetWeave API to read and write flat files directly from the application without the Agent. Use this for local unit testing.
NWDSINI2	A typical INI file that shows how to address an Agent to read and write remote flat files. (This Agent happens to be the loopback Agent.)
RC1	An obey file to execute the program.
RDC1	An obey file for debugging.
TANCC	The sample C program. It opens a file, writes one record, and closes it. Then it reopens the file for reading, reads the record, and closes it again.
TANCH	TANCC's header file.
TTE.CAP	Sample output from TANCC during a successful run.



## TANDEM\fifo\T2.ZIP Queued Messages

This collection of files shows how to build and run a simple C program that reads and writes queued messages to a FIFO that may be located on any platform with a NetWeave Agent.

File	Description
BINDIN	Input to the Binder; called from MAKE1IN.
MAKE1IN	Input to TACL; called from MAKECF1.
MAKEFF1	A "make" file for Tandem that compiles and binds TANFC.
NWDSINI	A typical INI file that shows how to address an Agent to read and write remote queue files. (This Agent happens to be the loopback Agent.)
RF1	An obey file for executing the program.
RDF1	An obey file for debugging.
TANFC	The sample C program. It opens a file, writes one record, and closes it. Then it reopens the file for reading, reads the record, and closes it again.
TANFH	TANFC's header file.
TTE.CAP	Sample output from TANFC during a successful run.



## TANDEM\ipc\T3.ZIP IPC Messaging

File	Description
BINDINC	Input to BINDER; called from MAKE1CIN.
BINDINS	Input to BINDER; called from MAKE1SIN.
MAKE1C	"Makes" the client.
MAKE1CIN	TACL input; called by MAKE1C.
MAKE1S	"Makes" the server.
MAKE1SIN	TACL input; called by MAKE1S.
NWDSINI	Sample configuration file.
RDIC1	Run the client in debug mode.
RDIS1	Run the server in debug mode.
RIC1	Run the client. (But start the server first.)
RIS1	Start the server.
TANICC	The client connects to the server, writes a message to it, and waits for the reply.
TANICH	Header file for TANICC.
TANISC	The server publishes as "ECHO_SERVER" and issues <code>nwds_ipc_accept</code> to wait for a connection. Once connected, it waits for a message from the client.
TANISH	Header file for TANISC.
TTEC.CAP	Sample output from the client program.
TTES.CAP	Sample output from the server.

## TANDEM\COBOL85\T4.ZIP Flat Files and Kernel Functions

The files below illustrate how to compile and link COBOL85 programs that include C language calls. There are four simple COBOL85 programs that show how to use file calls and Guardian calls.

**NOTE:** If you plan to build a COBOL85 application that makes both Guardian calls and NetWeave calls, study the sample file COB5 below.

File	Description
BINDCOB1	Binder input for building COB1.
BINDCOB2	Ditto COB2.
BINDCOB3	And COB3.
BINDCOB5	And COB5. (No, you are not missing COB4; there is none.)
COB1	Sample program that makes synchronous calls to write a record to a flat file.
COB2	Sample program that makes Guardian calls to write a record to a flat file. This program does not call any NetWeave functions.
COB3	This sample program shows that synchronous file calls can occur in the same program with Guardian calls, but only if carefully segregated.
COB5	Sample program that demonstrates the correct way to integrate Guardian functions with NetWeave calls. To allow NetWeave to manage the calls to awaitiox, you must use the NetWeave kernel functions.
COBH	The standard NetWeave copybook. Use the most current copy that is included with the NetWeave installation materials.
MAKECOB1	Compile and bind COB1.
MAKECOB2	Compile and bind COB2.
MAKECOB3	Compile and bind COB3.
MAKECOB5	Compile and bind COB5.
NWDSINI	Sample INI file; remember to modify it for your system.
RCOB1	Run command for COB1.
RCOB2	Run command for COB2.
RCOB3	Run command for COB3.
RCOB5	Run command for COB5.



## TANDEM\CRE\T9.ZIP

This example shows how to create TAL programs that use the CRE to incorporate C language calls. The point is to illustrate how these modules fit together, not to build a NetWeave application.

File	Description
BT	The BIND command that calls BTIN.
BTIN	Input to the binder to link TAL and C modules.
CC	Compile the C function.
CS	The source code for the C function.
CT	Compile the TAL main program.
TS	The source code for the TAL main program.

## TANDEM\CRE\T10.ZIP

This example shows how to build a TAL main program that can call a C function, which in turn calls a TAL function.

File	Description
BT	The BIND command that calls BTIN.
BTIN	Input to the binder to link TAL and C modules.
CC	Compile the C function.
CS	The source code for the C function.
CT	Compile the TAL main program.
TS	The source code for the TAL main program and the called function.

## UNISYS

The UNISYS/cobol74 directory contains the following sample files:

- cfile
- dms
- fifo



## Glossary

Agent	The NetWeave process that controls all input and output to queues, sends notifications to clients when data base changes have occurred, and is responsible for all aspects of security and data conversion.
Asynchronous	An operation in which the applications program is allowed to continue execution while the operation is performed. The access method informs the application program when the operation is completed.
Broadcast services	Simultaneous transmission of data to more than one destination: one sender, unlimited receivers. Message deliveries are connectionless and unacknowledged.
Data server services	Allows all other computers in the network, regardless of platform type, to access one computer's file system.
Client-server model	A client application sends a request message to a server program. The server program retrieves information or updates a local database on behalf of the (remote) client application.
Client-transaction services	Applications where programs communicate and synchronize operations by exchanging messages (IPC). They are used to implement on-line transaction processing and high-speed, real-time process control applications.
Consumer process	An asynchronous procedure that is responsible for processing the data in a message queue.
Dispatcher	In a distributor-based threaded server, the Dispatcher (provided by NetWeave as part of the <code>nwds_dispatcher_</code> function set) is responsible for creating application threads and passing messages to them once started.
Distributor	A NetWeave-provided facility for multi-threaded server processes. The Distributor starts and manages simple application threads for processing messages.
Event-driven design	A non-procedural methodology of software development that is asynchronous in nature, and is fundamentally multi-threaded because it allows you to maintain multiple concurrent sessions.



## Interprocess communication (IPC)

The process by which programs communicate data to each other and synchronize their activities.

### Item list

A variable-length array of parameters whose last element is a unique type, `NWDS_END_OF_LIST`. Each element (item) in the array has three components:

- *Type*: a constant from `netweave.h` that identifies a parameter (parameter name).
- *Length*: the length of the parameter value. Most parameters are either 16-bit integers (`NWDS_SHORT`) or 32-bit integers (`NWDS_LONG`). Variable-length parameters are considered to be of type `NWDS_CHAR`. For return item lists, the length is the maximum number of bytes that can be copied to the destination location.
- *Pointer to value*: for a control item list, this is the address of the location in memory where you have stored the value you want to assign to the parameter. For a return item list, this is the address in which to store the returned value.

### Legacy application

The vast collection of commercial and scientific applications written since the late 70s that share one or more of these features:

- The application resides on a single hardware platform.
- The user interface is the traditional character-oriented terminal.
- Access to related application functions is via menus and function keys.
- Application data are stored in record-oriented files.
- Access to these records is typically through keys and indices.

### Loopback testing mode

Used for unit testing locally. Most applications, except data server, can (and should) be constructed to run on a single platform. For example, if you are doing IPC messaging, construct a simple client or server to interact with your application. Such a test bed is said to run in loopback mode.

### Netweave.h

NetWeave header file. Contains the official definition of the API.

### On-line transaction processing (OLTP)

A system that processes multiple transactions concurrently and where the data flows to/from the computer directly from the point of origin.

### Peer-to-peer model

Data communications between two nodes(processes) that have equal status in the interchange. Any peer node can both generate messages to other processes as well as receive (*unsolicited*) messages from other processes.



Polling for a completion	Monitoring a flag that the completion function sets up when the (asynchronous) NetWeave function finishes.
Producer application	In FIFO message queuing, a producer puts messages at the tail of the queue, and a consumer gets messages from the head of the queue.
Queuing services	NetWeave services that store messages awaiting delivery. Queuing services are often the core of store and forward applications.
Receiver application	A process that reads and reacts to broadcast messages.
To scale (growth of application)	To enlarge or expand either a process, or the number of messages that a process can handle.
Sender application	An application program that generates a message to broadcast.
Synchronous function call	Initiated by a process that requests a specific event. All other processing is suspended until a response is received for the request.
Thread, boss thread, worker thread	The boss/worker thread model is a thread-based mechanism for work distribution between threads. A unit of work is delivered to the boss, which chooses a worker thread to perform the task and then return the result to either the boss or the originator.
UDP datagram	User Datagram Protocol (UDP) is an IP protocol. Datagrams are ideal for broadcasts because they are delivered to the IP network layer regardless how many nodes in the network may consume the information. A datagram is the basic unit of information passed across the Internet environment. It contains a source and destination address along with the data. An Internet Protocol (IP) datagram consists of an IP header followed by the data.
Unsolicited message	A message that a process receives without any prior prompting.
Workflow model	The automobile assembly line is a paradigm for the workflow model in manufacturing. Each cell accepts the outputs of its predecessors as its inputs, modifies the assemblage and passes its output to its successors.