V1.0 WNT Client/Server SDK

CS699 Project Report

Author: Tony Vaccaro
Date: 01-APR-2000
Rev: 1.0

TABLE OF CONTENTS

TABLE OF CONTENTS

Figures

Appendices

1.0 INTRODUCTION

1.1 Objective

The primary objective of this document is to describe the functional capabilities and basic structural design of the server (Agent), client (AgentTest) and installation programs that comprise the WNT Client/Server software development kit (SDK).

1.2 Brief Project Description

This WNT Information Agent Software Development Kit (SDK) is intended to be used by developers as a design center for implementation of client/server based applications. The SDK provides the primitives required to create a basic application. The developer must supply the additional application specific logic necessary to complete the total solution. This kit will be particularly suitable for use in the development of solutions where simple remote management of a system or resource is required. For instance, an application could be developed to issue commands and obtain system specific information from remote hosts for desktop management applications. Commands and information could include any that are supported by the Win32 API. Using custom designed driver software that interfaces with the server it would also be possible to produce an application that manipulates kernel mode resources.

1.3 Overall Goal and Reason for Development

The overall goal of the project was to design and implement specialized client and server software components that could provide a solid foundation for development of a complete multi-user, high-performance client/server solution. It is assumed that if the development process starts with these basic pre-designed and implemented client and server components, the overall product development cycle should be considerably simplified and reduced. Using this SDK will permit developers to focus and concentrate on application specific details and program logic.

1.4 SDK Software Components

Physically the SDK will be comprised of the following three basic software components:

- ***WNT Information Agent*** – The Agent is the server component of the kit. Installed as a service program on a Windows NT host system, it will load and run at boot time. The Agent is capable of issuing functions on the local host and of communicating with remote client programs. The project and binary are called WntInfoAgent.
- ***WNT Agent Test*** – The Agent Test program is the client component of the kit. It was implemented as a Win32 console based application. It may be installed and run from any WNT based host. The Agent test program is capable of issuing commands to and receiving responses from remote WntInfoAgents. The project and binary are called SPAgentTest.
- ***WNT Client/Server SDK Setup*** – The setup program is responsible for installing and setting up all required components of the SDK. It is also capable of uninstalling all components. The setup was created using the InstallShield development environment and has the capability to guide a user through the entire installation process.

1.5 Related Documentation

The documents listed below should be referenced for additional detail regarding the V1.0 WNT Client/Server SDK product.

- **A Software Development Kit for Windows NT Client/Server Computing** - Project Proposal for Professional Seminar Course CS699A, by Tony Vaccaro, v1.0, 1/24/00

2.0 SDK COMPONENT FUNCTIONALITY

The individual components of the Client/Server SDK were designed to provide the following functional capabilities.

2.1 WNT Information Agent

- The WNT Information Agent has been implemented as a Windows NT service application that is started automatically at system boot time and is controlled through use of the WNT Service Control Manager (SCM).
- The Agent interfaces with a host server using Win32 function calls and with remote clients using the Windows Sockets (WinSock) API. The Agent implements a set of custom commands that are used by clients requesting service.
- The Agent is capable of servicing multiple concurrent client connections and requests.
- The Agent implements a security mechanism that both identifies (the client is actually who it says it is) and authenticates (password verification) clients requesting connection.
- The Agent implements a common messaging protocol that is used when transmitting and receiving data with clients. The protocol allows for the transfer of simple ASCII character data as well as complex data structures.
- The Agent's messaging protocol allows for the return of status messages regarding the success or failure of the requested service.
- The Agent implements an event logging facility that facilitates troubleshooting and maintenance activities.
- The Agent is designed to be extensible, allowing the addition of new commands and service routines.

2.1.2 Basic Agent Service Command Set

The Agent implements the following basic command set:

- Request for the current version of the Agent: returning major and minor build
- Request for host information: returning host name, machine type, and OS version
- Request for file version information: returning the version info for a client specified file
- Request to shutdown the Agent service: a client commanded graceful service shutdown
- Request to test multithreaded operation: for test and verification purposes
- Request to return complex data structure: for test and verification purposes

2.2 Agent Test (Client) Program

- The Agent Test program is implemented as a command line (non-GUI) based program.
- The Agent Test program is capable of issuing all commands supported by the basic Agent service (see 2.1.2).
- The Agent Test program communicates with the Agent service program using the TCP/IP transport protocol.
- The Agent Test program uses the messaging protocol supported by the Agent service when transferring data and commands.
- The Agent Test program is able to display the constructed command syntax sent to the Agent service as well as the resulting response data.
- The Agent Test program is capable of establishing connection with the Agent service through use of the supported security mechanism.

2.3 SDK Installation and Setup Script

- The SDK Setup script is capable of installing the Agent service. This includes copying all Agent service required files to a user selectable target directory and modifying required WNT registry parameters.
- The SDK Setup script is capable of installing the Agent Test program. This includes copying all Agent Test program required files to the user selectable target directory and modifying required WNT registry parameters.
- The SDK Setup is capable of completely uninstalling the SDK. This includes removing all files and deleting any associated WNT registry entries.

3.0 SYSTEM AND PLATFORM SUPPORT

Architecture	Intel or Alpha server with 32 MB of memory min.
Operating System (Client)	Windows NT 4.0 Server, Window NT 4.0 Workstation, or Windows 95/98
Operating System (Server)	Windows NT 4.0 Server, Service Pack 4 or greater
Networking	Local Area Network and TCP/IP protocol stack on Client and Server hosts with static IP addresses
Disk Space	All software component files ~630KB
File System	NTFS or FAT partitions
Monitor	VGA
Input Devices	Keyboard and mouse

4.0 SDK PRODUCT LIMITATIONS AND CONSTRAINTS

4.1 No Support for DHCP

Due to DNS (Domain Name Service) limitations peculiar to the Windows NT 4.0 DNS service and the security system implemented in the SDK, both the client and server must be installed on systems with static IP addresses. Dynamic Host Configuration Protocol (DHCP) will not be supported.

4.2 No Graphical User Interface

Due to the application specific nature of a user interface the SDK only provides a rudimentary client design. The supplied SDK client logic may be coupled with any GUI to perform the backend processing required for a given application.

4.3 No Provision for Transmission of Complex Data Types

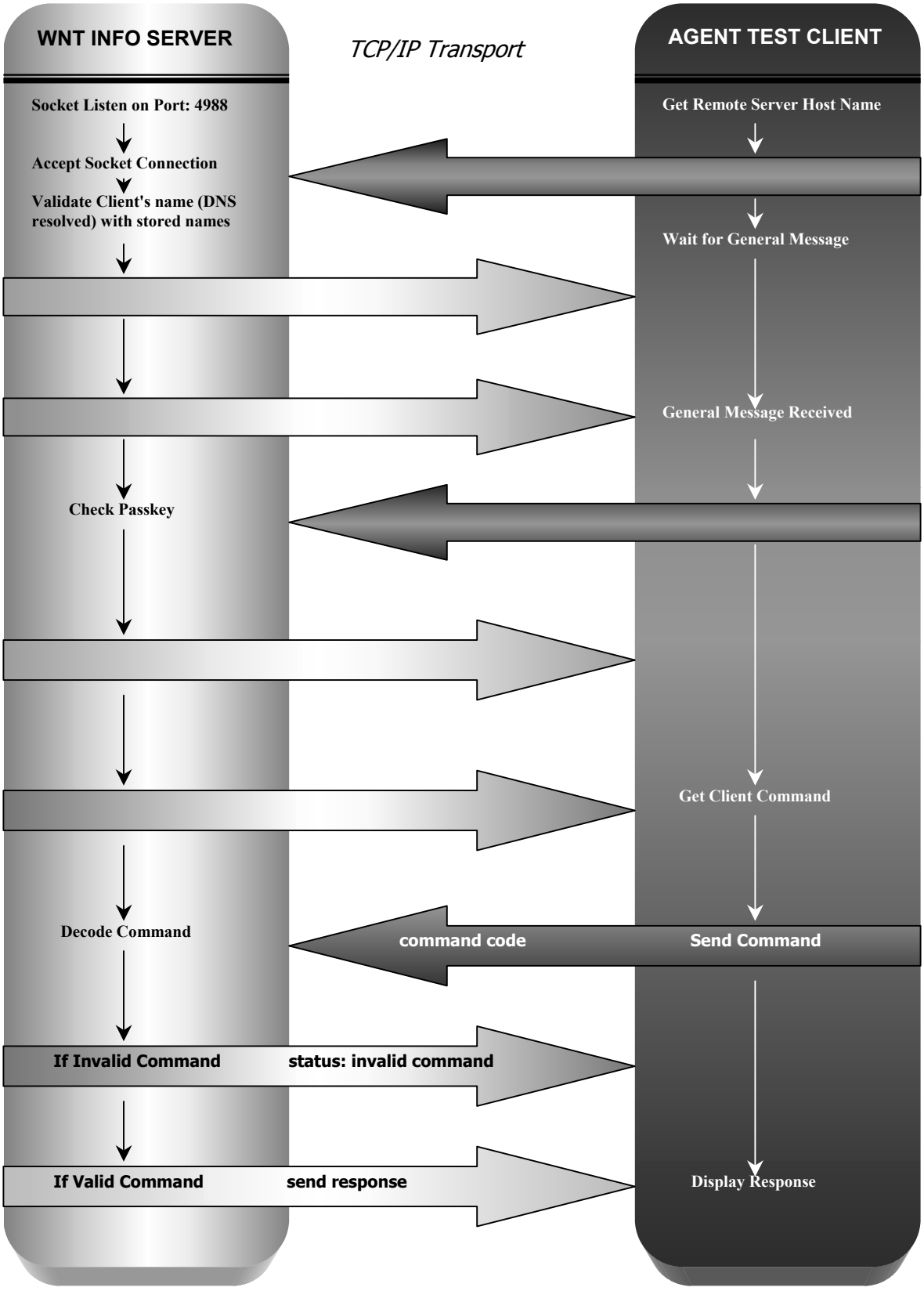
The basic client/server implementation in the SDK does not provide a mechanism for the transmission of complex data types (i.e. wav, mpg, or jpg files). This functionality is left as an exercise for the application developer.

5.0 SYSTEM ARCHITECTURE

5.1 Basic Client/Server Communication Architecture

The basic client/server security and communication architecture is illustrated in Figure 1. Inter-process communication is via sockets as implemented in Windows NT by the WinSock32 API.

Figure 1
Client/Server Communication Architecture



5.2 Server System Architecture

The SDK server is comprised of 10 modules organized according to specific function. This section describes the basic operation of the server and inter-relation between the component modules. Refer to the Server Architecture Diagram shown in Figure 3 for this section.

5.2.1 Starting the Agent Service

The Agent starts at system boot time when routines located in the Agent.c module invoke functions located in the WinServ.c module that issue the required series of Windows NT system service calls necessary to install the program as a service. The Agent module then calls functions located in the Config.c module which get platform specific information and locate the directory paths for the configuration and client files used to store client authorization information. Finally, the Agent module calls functions located in the Thread.c module to initialize critical data sections used by other modules, and to create the client communication thread. Once the client communication thread is launched, program execution is passed to the clientConnect() function located in the Client.c module.

5.2.2 Connecting with the Client

All client communication is via sockets using the WinSock32 API. When program execution is passed to the client communication thread in the Client.c module the clientConnect() function creates a socket and listens for client connections on port: 4988. When a connection request arrives, the function accepts the connection, which creates a new socket and returns a handle to the new socket. The original socket is returned to the listen state and the new socket is used for subsequent communication with connected client. The function next validates the client using authorization information located in the client and configuration files which are created by the user during the server installation and setup process. Refer to section 5.2.3 below for a detailed description of the client validation process.

5.2.3 Client Validation

The server's validation scheme is based on an identification and authorization model. The identification process determines that the connected client is who it says it is while the authorization process determines that the client can produce the correct password to gain access.

Validation begins when the server first connects and the socket is accepted. As part of the acceptance routine the server attempts to identify the client and makes a request of the Domain Name Service (DNS) to resolve (IP-to-Host name resolution) the IP address of the connecting client. The routine then attempts to match the returned DNS host name with a list of authorized clients supplied by the user in the client.ini file. If there is a match then the client will be authorized if the correct password is produced and validation will proceed. If not, the client connection is rejected.

To prevent reading the password by "snooping" the internetwork connection, the server next generates a random number and sends it to the connected client. This seed value is then used by the client to encrypt the password before it is sent back to the server for verification. The encrypted password (seed+password) is known as the PassKey. When the server receives the client's PassKey it generates its own version using the seed it sent across and an encrypted copy of the password that it reads from its local configuration file. If the client version of the PassKey matches the one generated by the server then the validation is complete and the client is granted access rights. If not, the client connection is rejected.

5.2.4 Process Request Thread Management

Process request threads perform all aspects of client command service including returning response data if required. Following client validation the function prepares a structure of thread parameters, creates the process request thread, and passes the parameter structure to the new thread. The parameter structure includes the connected client's name and other socket specific information that will be used by the process request thread. It is important to mention that because this program is designed for multi-threaded operation it is possible for worker threads to be executed out-of-order, on heavily loaded multiprocessor-based systems. Since it was necessary to define the argument structure as a global data type, it would be possible for this structure to be overwritten with data for newer requests. So, threads executing concurrently and out-of-order could read data meant for other threads. To resolve this potential problem the routine, which creates the argument data structure passed to the worker threads, is implemented using a 16 deep circular queue and an array of pointers to the queue is defined. This method ensures that each process request thread passed its own unique copy of arguments.

5.2.5 Command and Status Syntax

The protocol used for communication between the server and client application uses ASCII data exclusively. The use of ASCII allows the client and server applications to embed and parse delimitating ("|") and terminating ("<EOM>\n") character sequences through use of simple string and character functions. Figure 2 lists the specific command and return syntax used for communication between the client and server.

Figure 2
SDK Supported Command and Return Syntax

COMMAND	***COMMAND SYNTAX***	***RETURN***	***EXAMPLE***
Get Product Information	"007|ProcessId"	Preamble|Completion Status| (string)<EoM>\n"	"0|Agent_Status_Command_Success |Product Info (string) <EoM>\n""
Get Agent Information	"100|ProcessId"	Preamble|Completion Status| (string)<EoM>\n"	"0|Agent_Status_Command_Success |Agent Info (string)<EoM>\n""
Get Agent's Host Information	"101|ProcessId"	Preamble| Completion Status(string)| Agent Host Info<EoM>\n"	"0| Agent_Status_Command_Success |Agent Host Info (string)<EoM>\n""
Launch Beeper Thread	"102|ProcessId"	Preamble| Completion Status (string) <EoM>\n"	"0| Agent_Status_Command_Success <EoM>\n""
Get a File Version	"103|ProcessId| Dir\FileName"	Preamble| Completion Status (string)|File Version (string)<EoM>\n"	"0| Agent_Status_Command_Success |File Version(string)<EoM>\n""
Test Complex Structure Transfer	"104|ProcessId"	Preamble|Completion Status|hexified data structure (ASCII)<EoM>\n"	"0|Agent_Status_Command_Success| hexified data structure(ASCII)<EoM>\n"
ShutDown Agent	"900|ProcessId"	No Data Return	No Data Return

5.2.6 Data Structure Transfer

To pass data structures between the server and Agent the binary contents of the structure are encoded as hexadecimal characters (ASCII) prior to transmission. When the data is received from the server it is re-converted back to binary data and packed into a corresponding structure on the client side. The Hexify function in the Clientcmd.c module implements an engine that first casts a pointer to the structure to be transmitted as an unsigned char (byte) and then uses the ANSI sprintf() function to convert each byte into two hexadecimal characters (two bytes). The code fragment shown below is the actual data Hexify engine.

```
char* Hexify(char* bData, int iLength)
{
        int i;
        char* cHexBuf;

        cHexBuf = (char*) malloc((iLength*2) + 1);      // Grab a new buffer
        memset(cHexBuf, '\0', (iLength*2) + 1);         // Clear it

        // Convert each "char" of binary data to two hex chars
        for (i = 0; i < iLength; i++)
        {
                sprintf (cHexBuf + (i*2), "%02x", (UCHAR) bData[i]);
        }

        return cHexBuf;
}
```

5.2.7 Command Decoding and Execution

When the process request thread is launched it immediately reads the command string from the client socket using the handle passed from the thread parameter structure. The command string is first partially parsed to extract the command code, which is used to select the correct service routine from a switch statement. The specific service routine completes parsing the command string and extracts the client's process id, and any other arguments that may be required for the command. It then calls any auxiliary service functions in the Clientcmd.c module to complete processing of the command if necessary.

5.2.8 Supported Commands

The SDK supports the following basic set of commands:

- **Command Code 100 – Get Current Agent Version**

When the server receives the 100 level command code it retrieves information concerning the current Agent's version and build from stored global data. It then sends the version string back to the client and appends an end-of-message ("<EOM>\n") termination sequence to complete the message.

- **Command Code 101 – Get Host Server Information**

When the server receives the 101 level command code it retrieves information concerning the Agent's host server from stored global data. It then sends the string back to the client and appends an end-of-message ("<EOM>\n") termination sequence to complete the message.

- **Command Code 102 – Test Server Multiple Thread Capability**

This is a test command code used to evaluate the server's capability to dispatch and manage multiple client worker threads. When the server receives the 102 level command code it calls the BeepThread() function located in the Clientcmd.c module. This function creates a thread, which "beeps" every 4 seconds and displays a Windows MessageBox on the server host system requesting the user to push the OK button to dismiss the thread. It is possble for the Agent dispatch up to 16 concurrent threads using this command. When the user pushes the OK button the thread is destroyed and the communication socket is closed.

- **Command Code 104 – Return Test Structure**

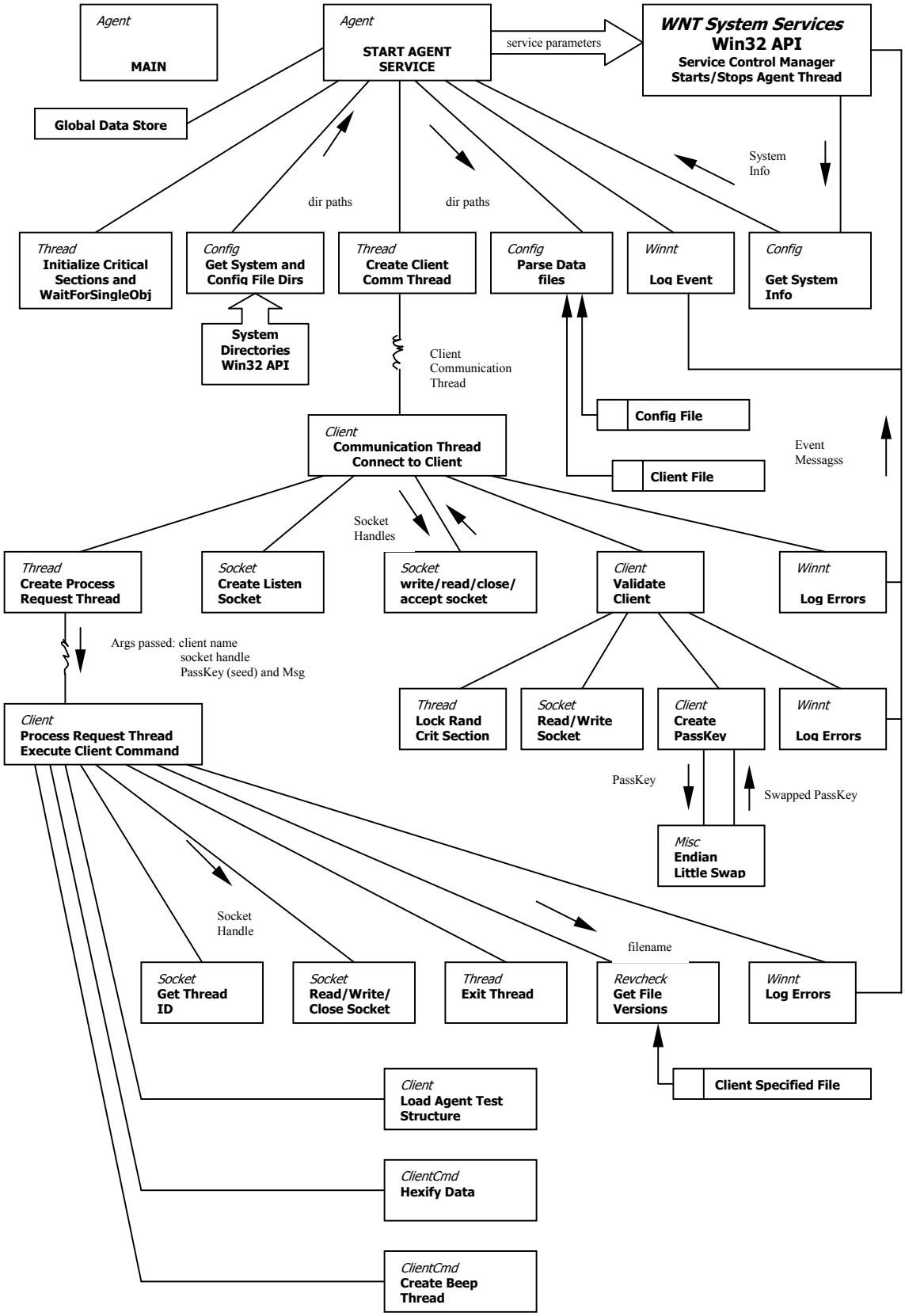
This is a test command code used to evaluate the server and Agent's capability to transmit and receive complex data structures. When the server receives the 104 level command it initializes a test structure with data, converts the data to hex characters, and transmits the contents to the client for display.

- **Command Code 900 – Shutdown Agent**

When the Agent receives the 900 command code it makes the required system service calls to gracefully shutdown the Agent service.

Figure 3

SERVER ARCHITECTURE



5.3 Client System Architecture

All client functions are implemented in a single module. This section describes the basic architecture of the client. Refer to the Client Architecture Diagram shown in Figure 4 for this section.

5.3.1 Client Startup

When the client program starts it requests input for the remote server's name and password. It then makes a Win32 call for the process Id and starts the socket interface before displaying a menu of the supported commands.

5.3.2 Connecting with the Server

To connect with the remote server (ClntConnect())the client calls DNS (Domain Name Service) to resolve the remote host's name (Name-to-IP) provided by the user on the command line. The resulting IP address is used to create a connection socket, which the client binds to the local IP and Port:4988. Once connected the client blocks and waits for a General reply message from the remote server.

5.3.3 Access Authorization

When the client receives the General message from the server it extracts the seed value, which it uses along with the supplied password to encrypt a PassKey (CreatePasskey()). The PassKey is then transmitted to the server and the client blocks waiting for the authorization reply. If the reply is successful the client is granted access to the server.

5.3.4 Building Commands

The client uses the command code input by the user to form (BuildCommand()) the proper syntax of the command message to be sent to the server (refer to Figure 2). Once built the client sends the command string to the client and blocks for the return message. The client decodes the return by first extracting the status string and then extracts the rest of the message depending upon the type of command it sent. For instance, if it sent a 104 command (Return Test Structure) it executes the routine to decode and pack the test data structure. If it sent a simple command (Return Agent Version) it simply prints the string returned in the message.

5.3.5 Receiving Data Structures

When receiving a complex data structure the client must convert the hexified data sent from the server to binary and pack the bits into the data structure. This is the opposite process from the one described in section 2.6.2 above. The code fragment below illustrates how this process is performed.

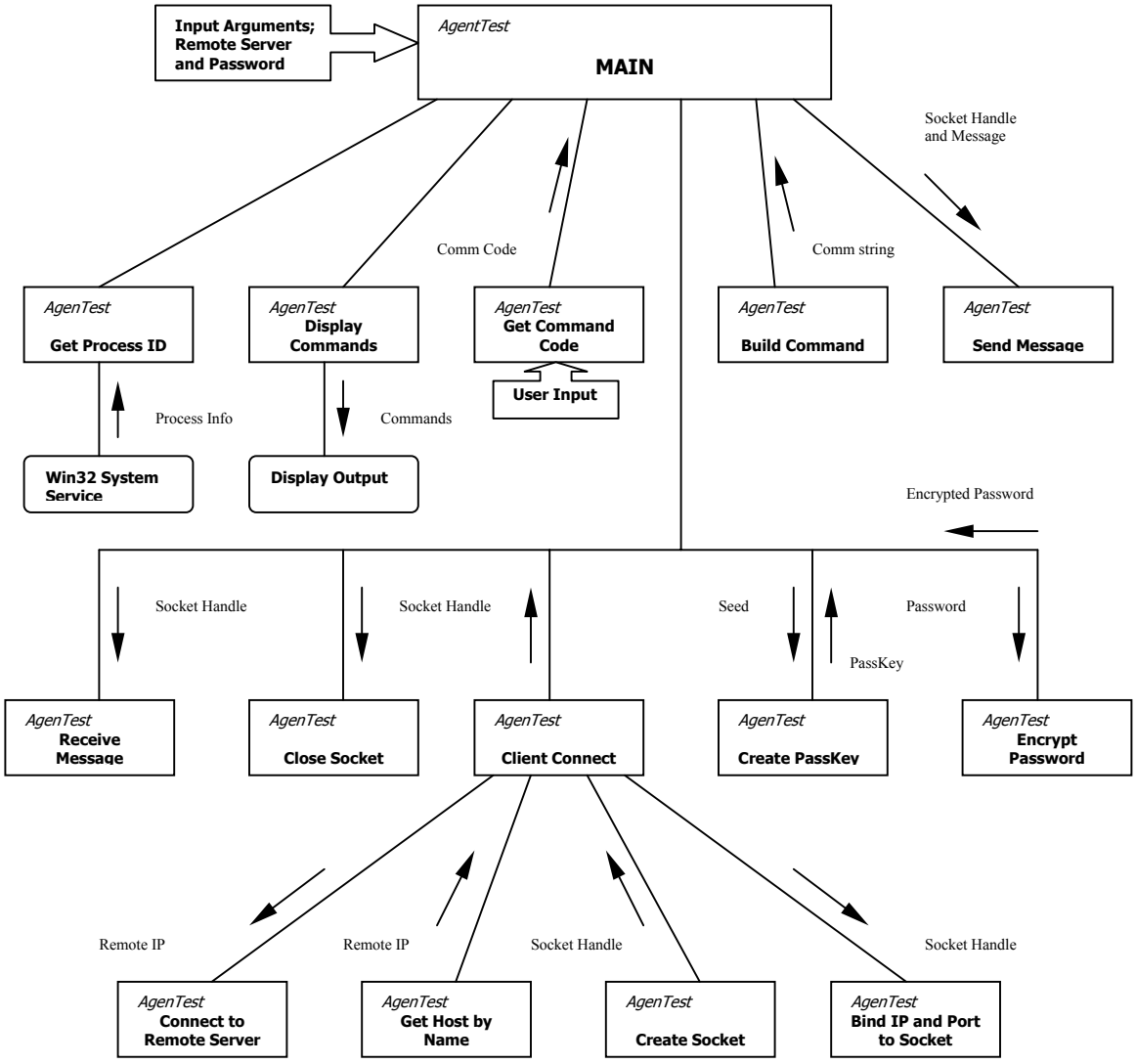
```
// cast a char pointer to the TestInfo structure
pTestInfo = (char *) &sAgentTestInfo;

// Loop till End - De-Hexify the data
while (*pHexData != '\0')
{
        // Get 2 characters
        memcpy(cTemp, pHexData, 2);

        // Save the byte in TestInfo structure
        *pTestInfo = (UCHAR) strtol(cTemp, NULL, 16);

        // Bump the pointers
        pHexData += 2;
        pTestInfo ++;
}
```

Figure 4
CLIENT ARCHITECTURE



6.0 DATA-FLOW DESIGNS

Data-Flow diagrams illustrate the basic flow of data between processes in the system. The Data-Flow diagrams for the server component are illustrated in Appendix A. The Data-Flow diagrams for the client component are illustrated in Appendix B.

7.0 DATA DICTIONARIES

Data Dictionaries catalog data items used in the system and functions that manipulate that data. The Data Dictionary for the server component is shown in Appendix C. The Data Dictionary for the client component is shown in Appendix D.

8.0 SOURCE CONTROL, DEBUG, AND SYSTEM TESTING

8.1 Source Control

Program source control was maintained through use of Microsoft's Visual SourceSafe V6.0, which was integrated into the Microsoft Visual C++ 6.0 development environment. SourceSafe was used to backup code source and track code modifications during program development.

8.2 Integrated Debugging

Integrated debugging was performed through use of Numega's BoundsChecker V6.0. BoundsChecker was used during program development to track thread execution, find memory leaks, and identify dangling pointers.

8.3 System Testing

System testing was accomplished primarily through the use of a set of specialized client commands and a debug version of the server. The debug server could be run as a Win32 Console application allowing program execution to be traced using printf() output statements. It was also possible to execute the debug server from the Visual C++ debugger, which allowed detailed examination of variable and structure contents from the program runtime environment. A system test matrix is presented in Appendix D, which identifies all test cases used during the test phase.

8.4 Software Metrics

Software quantity and quality metrics were measured using a tool called SourceMonitor. SourceMonitor is a Win32 program developed by James F. Wanner, a programmer and writer for Dr. Dobb's Journal (James F. Wanner, "SourceMonitor: Expose Your Code", Dr. Dobb's Journal, Vol. 25, Issue 3, March 2000, Pg. 92). Using this tool, which integrates well with the Visual C++ development environment, metrics were measured for both the server and client programs. According to the article the following values should be considered acceptable for C code:

- Percent comments: 10 to 50 percent
- Percent branches: 10 to 30 percent
- Average block depth: below 1.8

As shown in Figures 5 and 6 all metrics for both programs fall well within the acceptable range of values.

Figure 5
Monitor Metric Details – Server Program

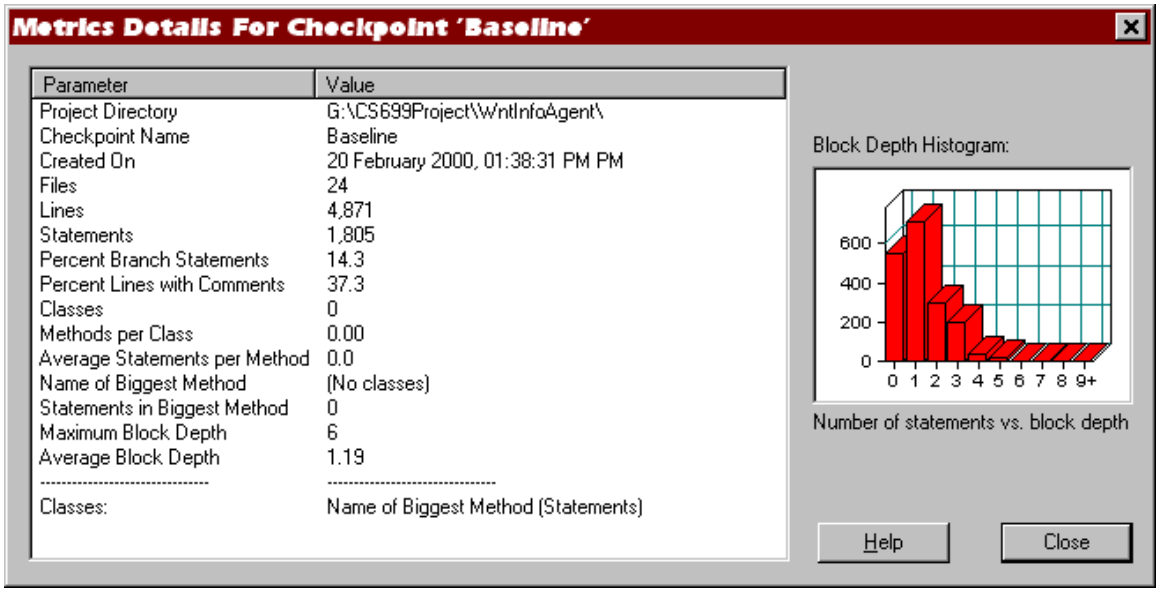
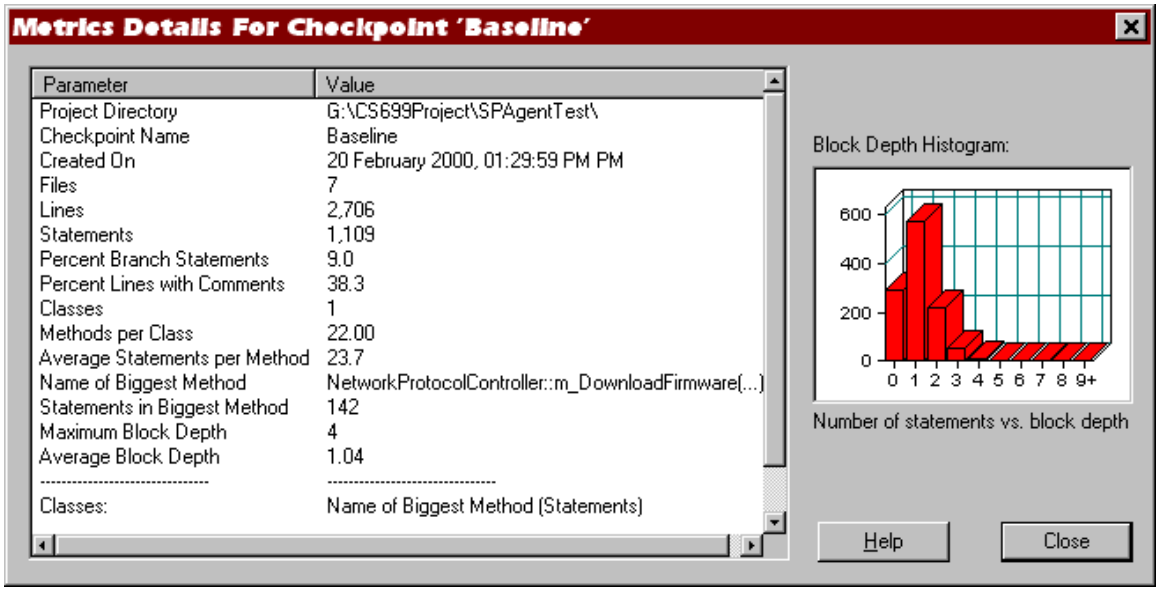
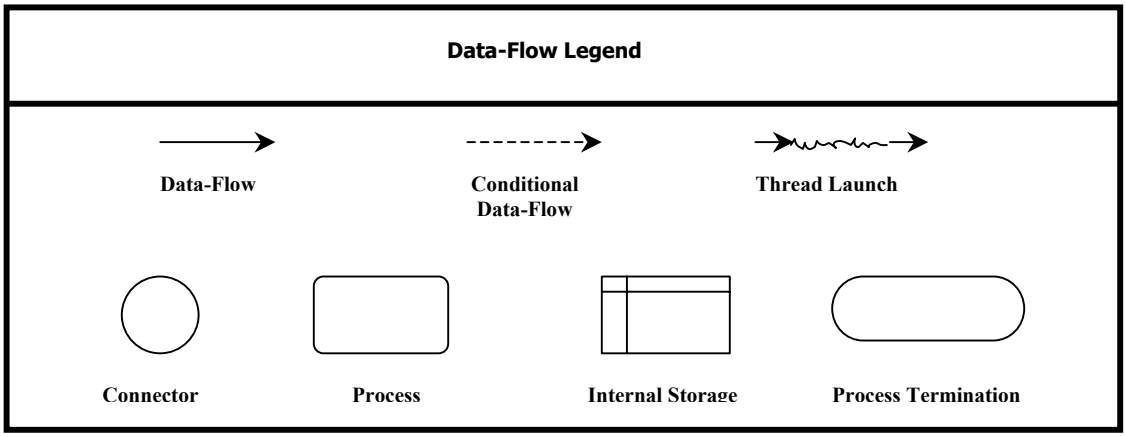
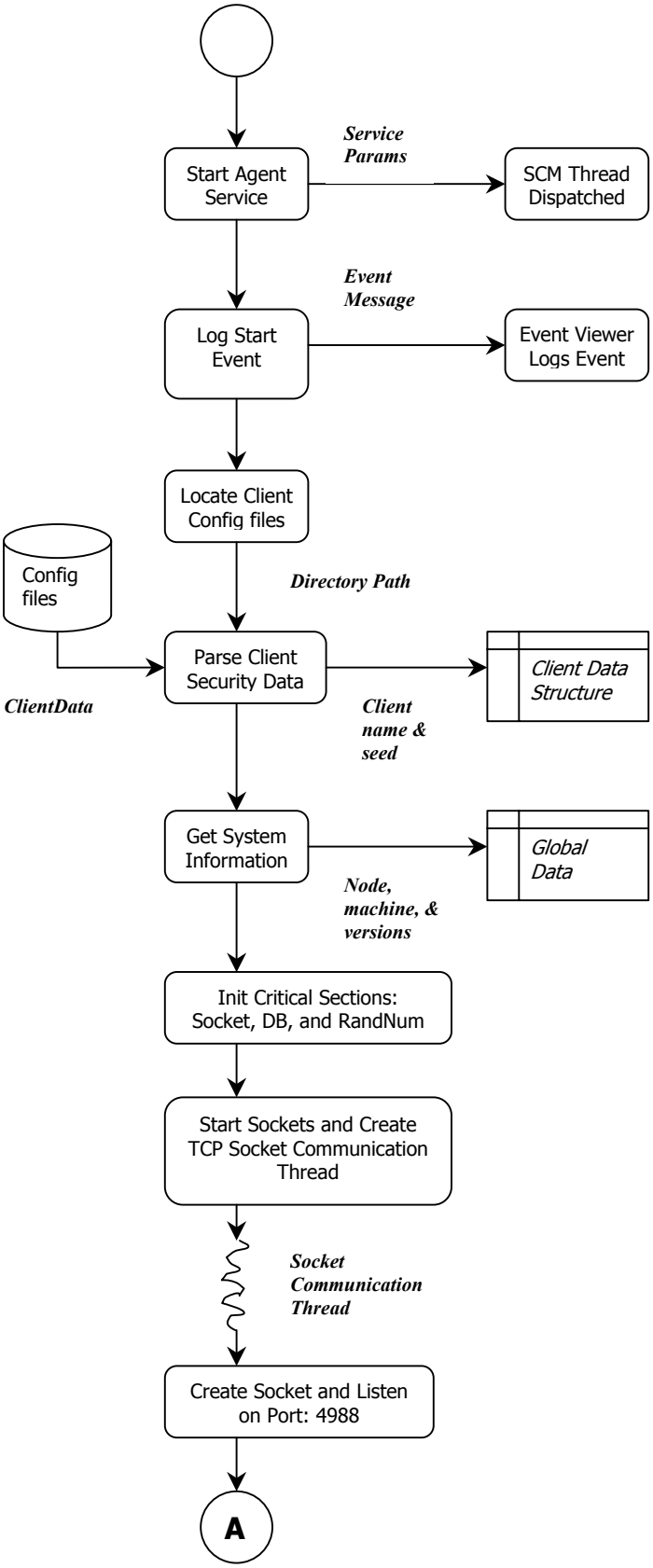


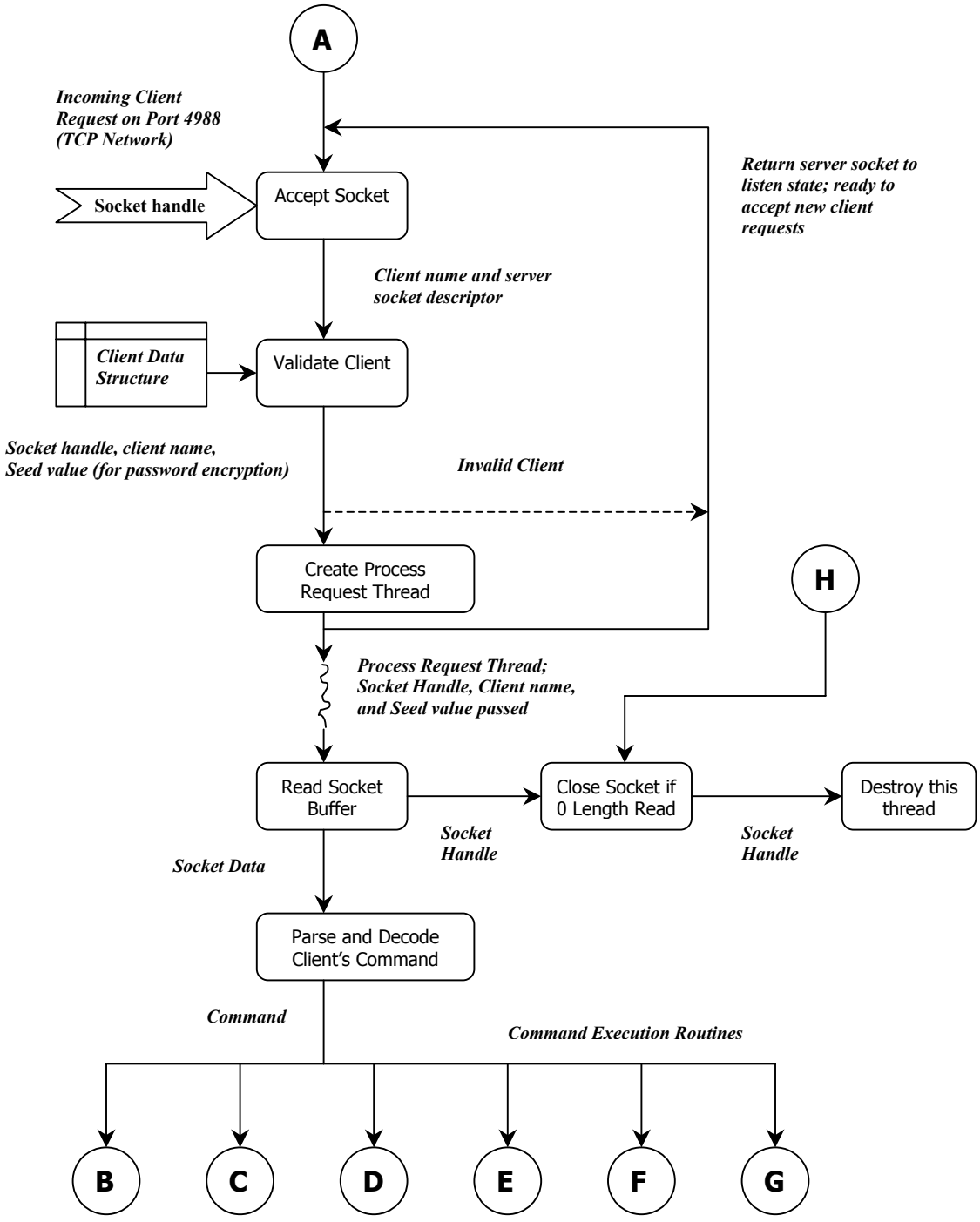
Figure 6
Monitor Metric Details – Client Program Component

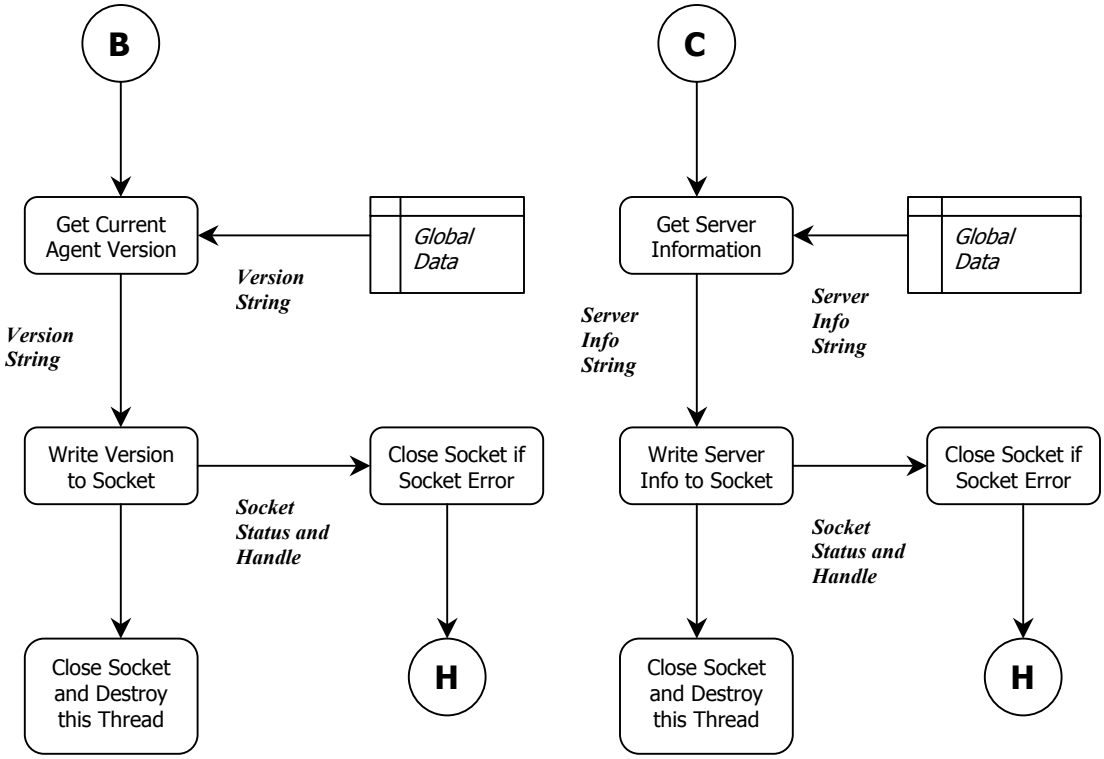


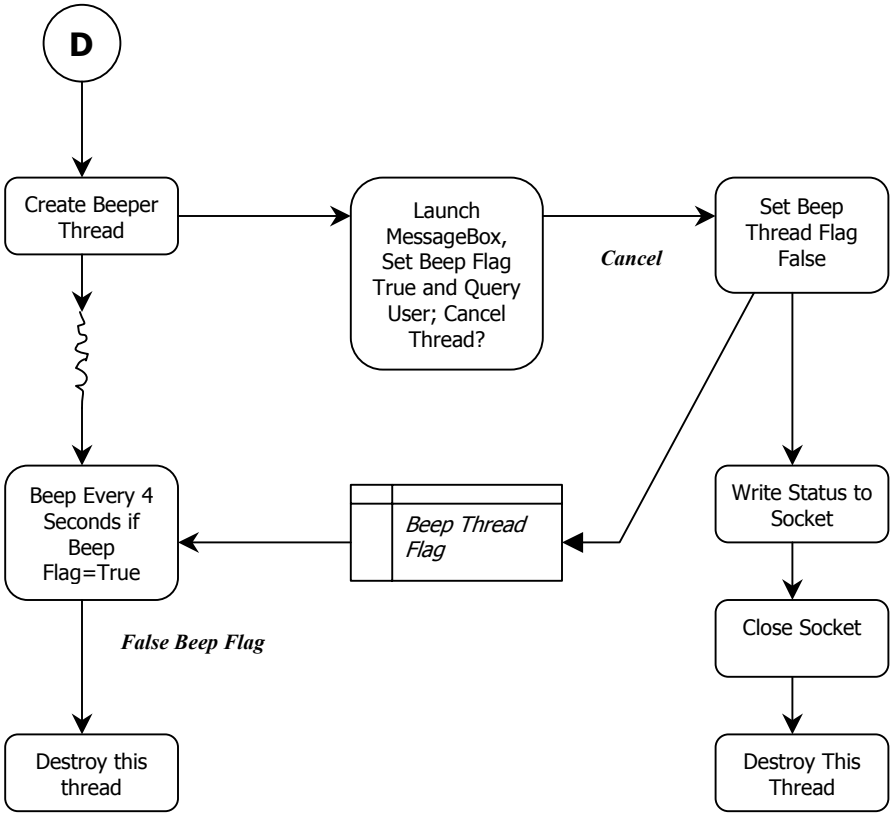
APPENDIX A
Server Data-Flow Diagrams

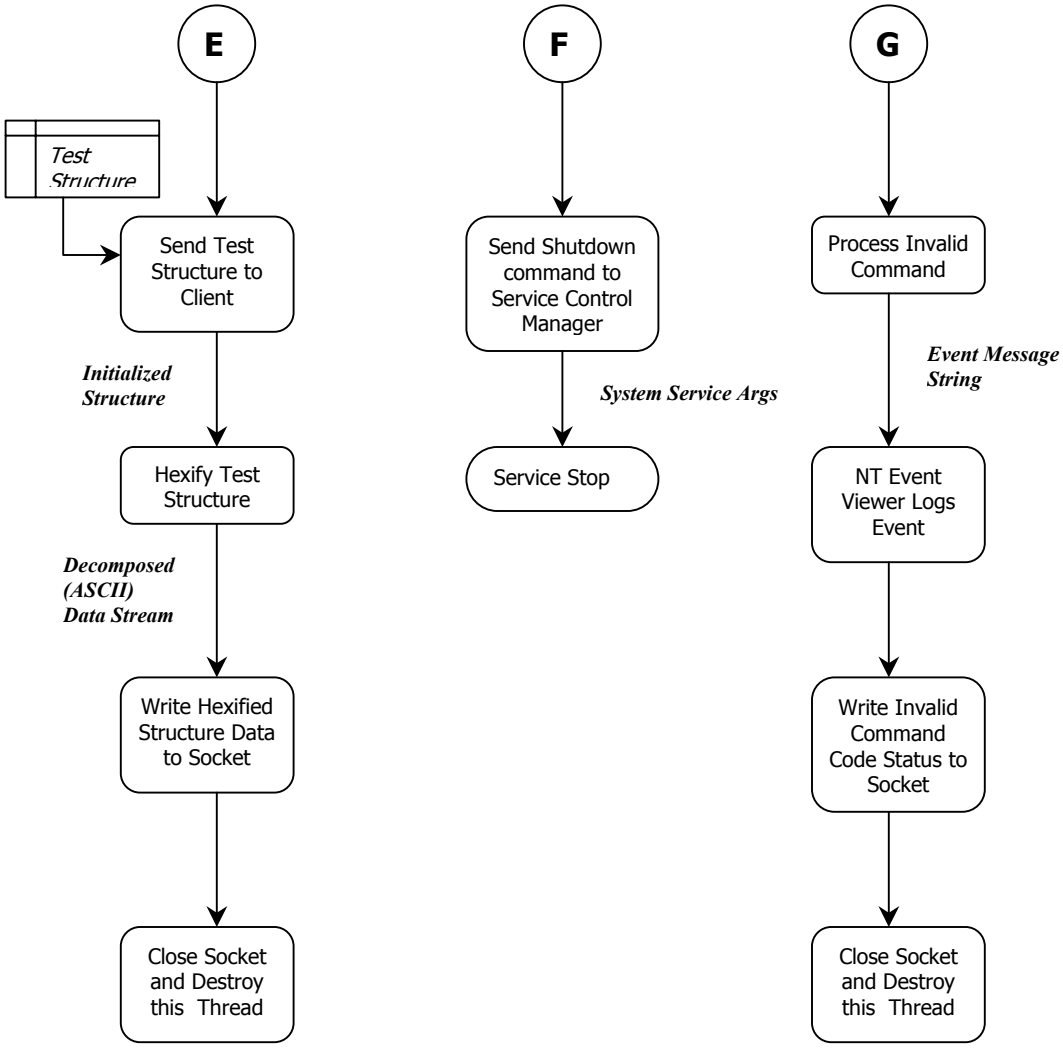



Service Params
Start Agent Service
SCM Thread Dispatched
Event Message
Log Start Event
Event Viewer Logs Event
Locate Client Config files
Config files
Directory Path
Parse Client Security Data
Client Data Structure
ClientData
Client name & seed
Get System Information
Global Data
Node, machine, & versions
Init Critical Sections: Socket, DB, and RandNum
Start Sockets and Create TCP Socket Communication Thread
Socket Communication Thread
Create Socket and Listen on Port: 4988
A

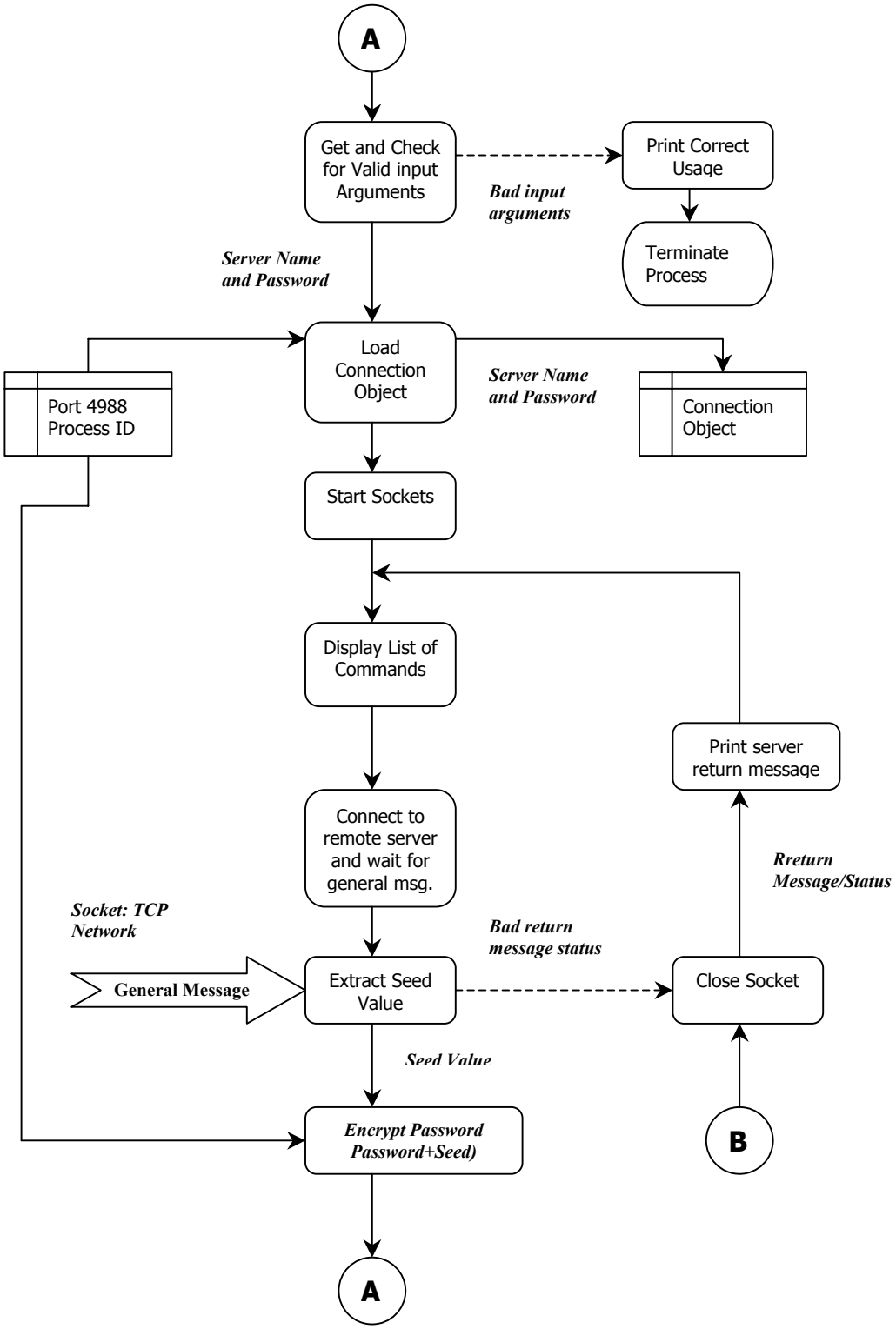

A
Incoming Client Request on Port 4988 (TCP Network)
Socket handle
Accept Socket
Return server socket to listen state; ready to accept new client requests
Client name and server socket descriptor
Client Data Structure
Validate Client
Socket handle, client name, Seed value (for password encryption)
Invalid Client
Create Process Request Thread
H
Process Request Thread; Socket Handle, Client name, and Seed value passed
Read Socket Buffer
Close Socket if 0 Length Read
Destroy this thread
Socket Handle
Socket Handle
Socket Data
Parse and Decode Client's Command
Command
Command Execution Routines
B
C
D
E
F
G

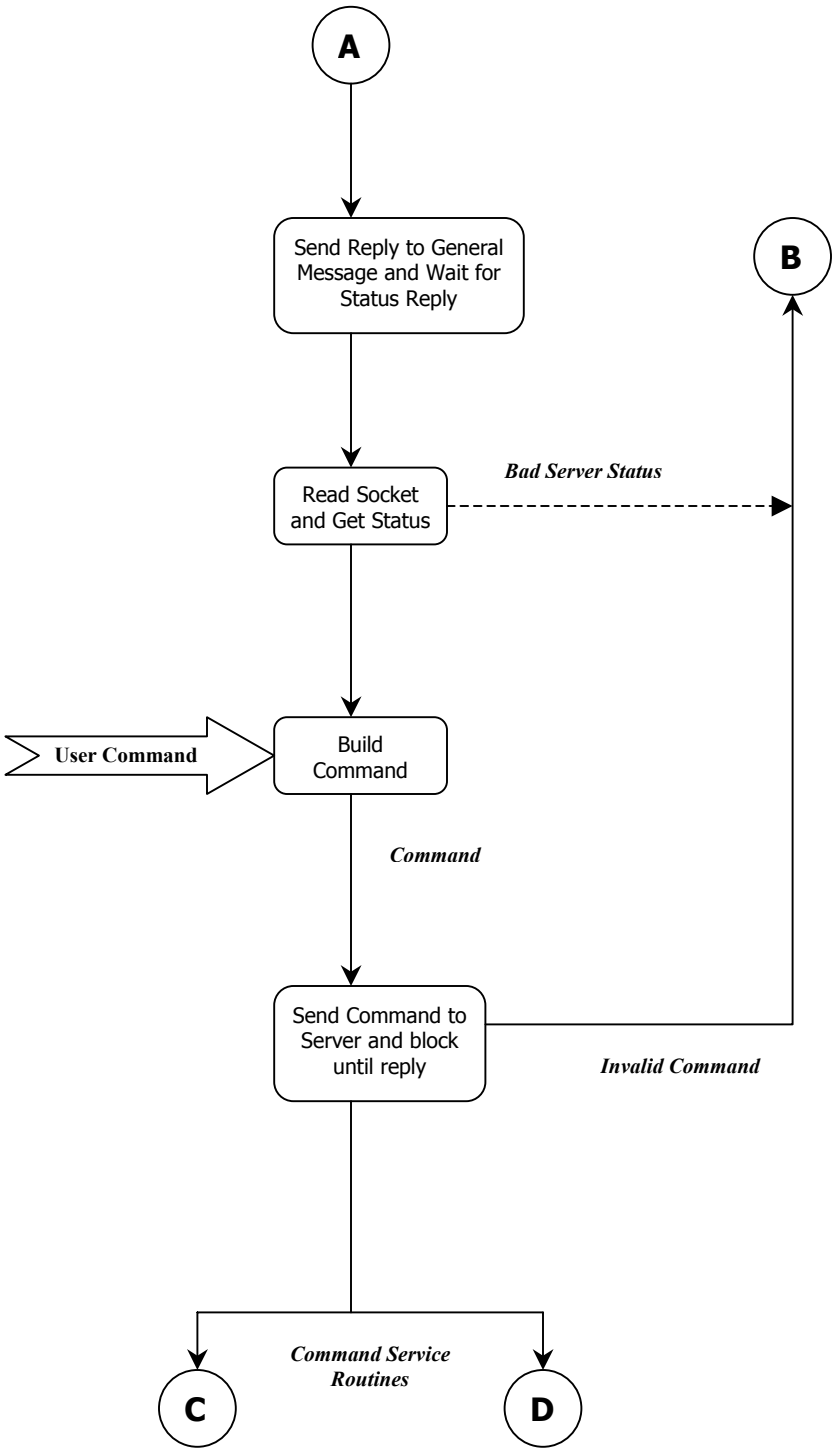

B
Get Current Agent Version
Global Data
Version String
Version String
Write Version to Socket
Close Socket if Socket Error
Socket Status and Handle
Close Socket and Destroy this Thread
H
C
Get Server Information
Global Data
Server Info String
Server Info String
Write Server Info to Socket
Close Socket if Socket Error
Socket Status and Handle
Close Socket and Destroy this Thread
H

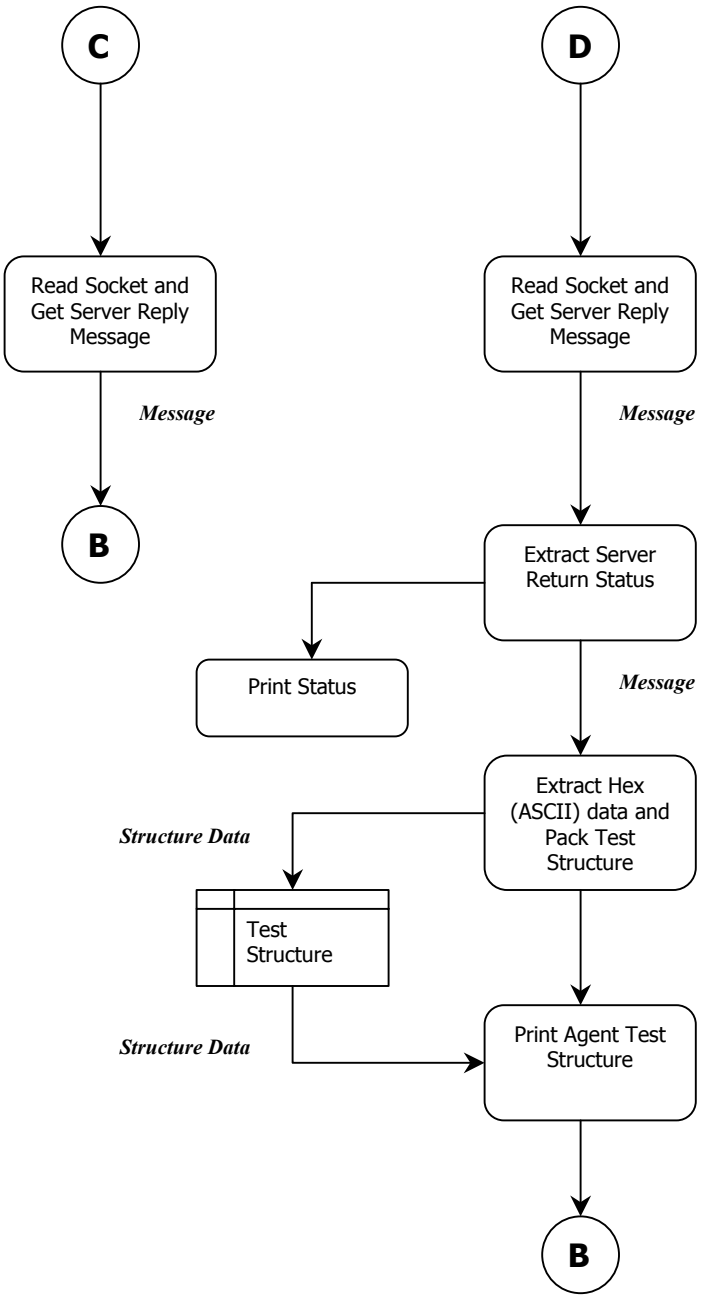

D
Create Beeper Thread
Launch MessageBox, Set Beep Flag True and Query User; Cancel Thread?
Cancel
Set Beep Thread Flag False
Beep Every 4 Seconds if Beep Flag=True
Beep Thread Flag
Write Status to Socket
Close Socket
False Beep Flag
Destroy this thread
Destroy This Thread


E
Test Structure
Send Test Structure to Client
Initialized Structure
Hexify Test Structure
Decomposed (ASCII) Data Stream
Write Hexified Structure Data to Socket
Close Socket and Destroy this Thread
F
Send Shutdown command to Service Control Manager
System Service Args
Service Stop
G
Process Invalid Command
Event Message String
NT Event Viewer Logs Event
Write Invalid Command Code Status to Socket
Close Socket and Destroy this Thread

APPENDIX B
Client Data-Flow Diagrams




A
Send Reply to General Message and Wait for Status Reply
B
Read Socket and Get Status
Bad Server Status
User Command
Build Command
Command
Send Command to Server and block until reply
Invalid Command
Command Service Routines
C
D


C
D
Read Socket and Get Server Reply Message
Read Socket and Get Server Reply Message
Message
Message
B
Extract Server Return Status
Print Status
Message
Extract Hex (ASCII) data and Pack Test Structure
Structure Data
Test Structure
Structure Data
Print Agent Test Structure
B

APPENDIX C
SERVER DATA-DICTIONARIES

CLIENT Module Abstract and Data Dictionary

The Client module contains the Agent's main service routine, which listens for and services client requests. The main service routine establishes a listening socket, bound to the local server's IP address and Secure Path port, and waits for a client connection. Following connection, the routine validates the client against a list of known clients stored in the client.ini file. If the client successfully passes validation the Agent parses the command request, creates a thread and passes the command along with arguments to the ClientCmd module for service. When the ClientCmd module completes servicing the command it returns and the main service routine closes the client's socket. The Client module completes the transaction by destroying the worker thread.

Entity Name	Type	Description
AGENTESTINFO	Structure	(UINT) uAnInteger - just an integer; (double) dbAFloater - just a float (CHAR) strString_1[100] - String #1 (CHAR) strString_2[100] - String #2 (CHAR) strString_3[100] - String #3 (CHAR) strString_4[100] - String #4
CLIENTINFO	Array	infoArray[MAX_THREADS] – array of info structures
ClientConnect	Function	Sets up socket and client state variables, and waits for a client process to request service. Requested services are passed to ProcessRequest for disposition. This thread never ends under normal circumstances ***Inputs:*** Pointer to an 'array' of passed arguments ***Outputs:*** None
ProcessRequest	Function	Parse a client's message string and act on it. ***Inputs:*** Pointer to an argument list including the socket handle, client's name, and authorization level ***Outputs:*** Returns 0 for normal completion, -1 if an EOF from the client is encountered (i.e. the client closes the socket).
ValidateClient	Function	Validate a connecting client and its access privileges. ***Inputs:*** (string) Name of the client host system, socket handle, and (int) thread_seed. ***Outputs:*** Returns the client's access level if client access is authorized (thread_seed is also loaded). Returns -1 if it is not.
Client.ini	File	User supplied list of authorized clients.

CLIENTCMD Module Abstract and Data Dictionary

ClientCmd includes a set of functions that perform command specific processing for client requests received by the Agent.

Entity Name	Type	Description
THREADPARAMS	Structure	(int) iThreadCount – max number of threads PTHREADCOUNTINFO pThreadCountInfo - Pointer to global threadcountinfo struct
ThreadCountInfo	Structure	(int) iThreadCount - counts active Beep threads (BOOL) ar_bBeepStop[MAX_THREADS] – stores stop flags
Hexify	Function	Converts a buffer of binary data to ASCII Hex characters. ***Inputs:*** Pointer to binary buffer and length of buffer. ***Outputs:*** Pointer to ASCII Hex buffer
BeepThread	Function	Prepares thread parameters and creates a worker thread which continually beeps until a system modal MessageBox is acknowledged. ***Inputs:*** PTHREADCOUNTINFO, a pointer to a thread info structure. ***Outputs:*** Pointer to a status string to return to the client.
Beeper	Function	Runs as a worker thread and continually beeps to indicate the existence of the thread. ***Inputs:*** A pointer the ThreadParams structure ***Outputs:*** None

AGENT Module Abstract and Data Dictionary

The purpose of the Agent module is to request the WNT Service Control Manager (SCM) to start the Agent service, prepare configuration data, initialize critical sections, and launch the client communication thread to wait for incoming service requests.

Entity Name	Type	Description
MAIN	Function	The main routine. First called at program startup. ***Inputs:*** argc - The number of arguments in 'argv'. argv - a pointer to an 'array' of passed arguments. ***Outputs:*** None
StartAgent	Function	This routine starts the Agent functions ***Inputs:*** None ***Outputs:*** None

REVCHECK Module Abstract and Data Dictionary

The purpose of the Revcheck module is to retrieve the file version from a specified file's GetFileVersionInfo structure.

Entity Name	Type	Description
GetFileVer	Function	Retrieves the file version from the GetFileVersionInfo structure. ***Inputs:*** LPSTR strFileName - name of file LPSTR strFileVersion - gets loaded with the version ***Outputs:*** Returns TRUE if a version is found

CONFIG Module Abstract and Data Dictionary

The purpose of the Config module is to prepare platform specific information. For instance the module locates the directory paths and parses information from required initialization files.

Entity Name	Type	Description
GetConfig	Function	Reads in data from the configuration file containing the encrypted password. ***Inputs:*** None ***Outputs:*** Returns 0 for normal completion, -1 for an error condition.
GetClientData	Function	Reads authorized client data from client.ini file and stores host system name, notification mode, and the security authorization code. ***Inputs:*** None ***Outputs:*** Returns 0 for normal completion, a negative value for an error condition.
GetSysInfo	Function	Gathers information about this application and its host system and loads global data. ***Inputs:*** None ***Outputs:*** Returns 0 for normal completion, -1 for an error condition.
sysDir	Function	Loads the SYS_DIR character array with the name of the system directory. ***Inputs:*** None ***Outputs:*** None
steamDir	Function	Pull the subdirectory from the string which is actually 'argv0' from main(). If a directory name exists, changes to that directory. For Debug Only. ***Inputs:*** cstring - a string containing the subdirectory as a subset. ***Outputs:*** None

MISC Module Abstract and Data Dictionary

The purpose of the Misc module is to handle miscellaneous functions required by other modules. These include endian little conversions, ASCII-to-binary, and binary-to-ASCII routines.

Entity Name	Type	Description
endianLittle	Function	This function returns '1' if little endian (IBM) machine, otherwise returns '0' if big endian (IEEE) machine. ***Inputs:*** Value "1" ***Outputs:*** returns '1' if little endian (IBM) machine, otherwise returns '0' if big endian (IEEE) machine
endianSwap	Function	Function to swap bytes for little endian machine to convert to big endian machine ***Inputs:*** array - array of words to swap. (int) - The word length in bytes. (int)- The number of words to swap. ***Outputs:*** None
ascii_to_bin	Function	Converts a string of 2 digit acsii hex numbers to a binary value in a buffer. ***Inputs:*** ascii_ptr - pointer to char string to be converted ***Outputs:*** bin_value - the binary value buffer
bin_to_ascii	Function	Converts a binary buffer to a ASCII hex string of chars. ***Inputs:*** bin_ptr - binary buffer to be converted ascii_ptr - pointer to char sting count - number of bytes to convert ***Outputs:*** None

SOCKET Module Abstract and Data Dictionary

The SOCKET module provides a set of WinSock services that are used by other modules in the Agent to communicate directly with a connected client.

Entity Name	Type	Description
socketCreateByService	Function	Create a listen socket for Secure Path using the TCP protocol with information from the \etc\services file. ***Inputs:*** The services file name entry (securepath) and the type of connection to create (tcp). ***Outputs:*** The listen socket file descriptor (int).
socketAccept	Function	Accept a socket connection on a listen socket. ***Inputs:*** The listen socket file descriptor (int). ***Outputs:*** The active socket file descriptor (int).
socketClose	Function	Close an existing socket. ***Inputs:*** the socket file descriptor (int). ***Outputs:*** The close error.
socketRead	Function	Read from a socket. ***Inputs:*** The socket file descriptor (int), the data buffer to fill (void), and the max length of the read operation (int). ***Outputs:*** The number of bytes read from socket (int).
socketWrite	Function	Write to a socket. ***Inputs:*** The socket file descriptor (int), the data buffer to send (void), and the max length of the read operation (int). ***Outputs:*** The number of bytes written to socket (int).
HOSTENT	Structure	Used to store information about a host. Windows Sockets allocates the **HOSTENT** structure.
SOCKETADDR_IN	Structure	This structure is used by WinSock to specify a local or remote endpoint address to which to connect a socket.

THREAD Module Abstract and Data Dictionary

The Thread module provides a set of thread creation and management functions to other program modules. These include thread creation, termination and initialization of critical sections.

Entity Name	Type	Description
dmThreadCreate	Function	Creates threads. ***Inputs:*** start_routine - The procedure that the thread calls. args - The parameters for the above procedure. idetach - =1 for detached thread, =0 for attached thread. ibound - =1 for bound thread, =0 for unbound thread. ipriority - The scheduling priority. ***Outputs:*** thread - The thread id.
dmThreadJoin	Function	Causes the calling thread to wait for termination of specified thread. ***Inputs:*** thread - The thread to wait on. value_ptr - Return value of the terminating thread. ***Outputs:*** error status.
dmThreadExit	Function	Terminates the calling thread. ***Inputs:*** value_ptr - value returned to calling thread. ***Outputs:*** None
dmThreadSelf	Function	Obtains the identifier of the current thread. ***Inputs:*** None ***Outputs:*** The thread identifier.
dmThreadMutexInit	Function	Initialize a critical Section ***Inputs:*** lock - The mutex lock. ***Outputs:*** error status.
dmThreadMutexLock	Function	Enter (join) a critical Section ***Inputs:*** lock - The mutex lock. ***Outputs:*** error status.
dmThreadMutexUnlock	Function	Leave a critical Section ***Inputs:*** lock - The mutex to unlock. ***Outputs:*** error status.

WINNT Module Abstract and Data Dictionary

The Winnt module provides a set of Windows event logging and messaging routines used by other program modules. These include logging informational, warning, and error events.

Entity Name	Type	Description
systemLogOpen	Function	Opens a handle to the WNT Event Viewer. ***Inputs:*** None ***Outputs:*** handle
systemLogClose	Function	Closes a handle to the WNT Event Viewer. ***Inputs:*** handle ***Outputs:*** None
systemLogWarn	Function	Log a Warning Type message. ***Inputs:*** char * pointer to message string ***Outputs:*** None
systemLogAlert	Function	Log an Error Type message. ***Inputs:*** char * pointer to message string ***Outputs:*** None
systemLogInfo	Function	Log an Informational Type message. ***Inputs:*** char * pointer to message string ***Outputs:*** None
WinDebugEvent	Function	This routine creates an event and then waits for the event signal. ***Inputs:*** None. ***Outputs:*** None.
winMessage	Function	This routine creates a message from an error value. The error value must come from GetLastError() or one of its deviates. ***Inputs:*** ierr - the error value from GetLastError(). ***Outputs:*** cmessage - the string container to hold the message.
winSocketStart	Function	Initialize WINSOCK API ***Inputs:*** None ***Outputs:*** 1=success, 0=failure.

WINSERV Module Abstract and Data Dictionary

The Winserv module provides a set of functions that facilitate WNT Service Management Control.

Entity Name	Type	Description
ResumeService	Function	Resumes a paused service. ***Input arg:*** This thread handle ***Outputs:*** NONE
PauseService	Function	Pauses the service. ***Inputs:*** This thread handle ***Outputs:*** None
StopService	Function	Stops the service. ***Inputs:*** This thread handle ***Outputs:*** NONE
SendStatusToSCM	Function	This function consolidates the activities of updating the service status with SetServiceStatus. ***Inputs:*** Win32 Service Args ***Outputs:*** BOOL success/failure
ServiceCtrlHandler	Function	Dispatches events received from the service control manager ***Inputs:*** DWORD controlCode ***Outputs:*** None
terminate	Function	Handle an error from ServiceMain by cleaning up and telling SCM that the service did not start. ***Inputs:*** DWORD error ***Outputs:*** None.
ServiceMain	Function	Registers the service and tracks progress. ***Inputs:*** Argc and Argv ***Outputs:*** None
startWINService	Function	Starts the service ***Inputs:*** None ***Outputs:*** None.

Appendix D
System Test Matrix

TEST CASE	DESCRIPTION	SUCCESS CRITERA*	PASS	FAIL
Kit Installation and Setup	Invoke Setup wizard to install all components of the kit.	Agent and Client programs are installed in user selected folders, correct registry parameters are entered, Agent service starts, and Client program executes. Startup event is logged in the event viewer	X	
Kit UnInstallation	Invoke WNTInfoAgent uninstall from Add/Remove Programs applet.	Uninstall can be invoked from Add/Remove Programs applet. All files and folders are removed from target system.	X	
Agent and Client Security	Connect with Agent using proper client authorization and password. Attempt to connect with Agent using improper client authorization and password.	Client is granted access to the server using the correct password and/or client name authorization. Client is denied access without the correct password and/or client name authorization.	X	
100 Level Command	Issue 100 Level Command to Agent from Client.	Agent returns the correct current version of the running agent to the client and the client outputs the version string.	X	
101 Level Command	Issue 101 Level Command to Agent from Client.	Agent returns the correct server information to the client and the client outputs the information string	X	
102 Level Command	Issue 102 Level Command to Agent from Client.	Agent launches Beeper worker thread on host server and presents MessageBox. Agent cancels worker when user clicks OK.	X	
103 Level Command	Issue 103 Level Command to Agent from Client with valid file path and name.	Agent returns the correct file version for the specified file and the Client properly displays the output.	X	
104 Level Command	Issue 103 Level Command to Agent from Client.	Agent returns test structure contents to Client and Client properly displays output.	X	
900 Level Command	Issue the 900 Level Command to the Agent from the Client.	Agent shuts-down. Shutdown event is logged in the event viewer.	X	
Multi-thread and multi-user Tests	Use 16 clients to concurrently connect with the Agent and issue 102 Level Commands.	Agent establishes connection with each client and launches a corresponding beep thread for each. Also, threads are destroyed and connections closed when each thread is cancelled.	X	

*For all Test Cases the following general success criteria apply:

- No Application failures
- No System bug checks