SWENG568: Enterprise Integration

Lesson 2: Evolution of Integration Patterns (I): Communication, Data Sharing, and Remote Access

---

Introduction (1 of 8)

Learning Objectives

- Understand different methods of sharing data between applications
- Understand different methods of sharing functionality between applications
- Know the difference between sharing data and functionality
- Master basic concepts: socket communications, remote procedure calls, and remote invocation methods

---

By the end of this week, make sure you have completed the readings and activities found in the Lesson 2 Course Schedule.

Different Methods of Sharing Data Between Applications (2 of 8)

Different Methods of Sharing Data Between Applications

Business operations frequently requires for data sharing between applications which reside on the same computer or different computers. The three significant methods of sharing data between applications are:

1. file-based data sharing
2. the use of common database
3. sockets.

Do you know the main difference between these three (3) methods?

Without question, they are totally different means of data sharing in terms of technical implementations. But from a user's perspective, the first two (2) are not real-time. When a real-time connection to enable data sharing between applications, the third (i.e., sockets) should be used [1].

File-based sharing (3 of 8)

File-based Data Sharing

The easiest and most common means of sharing data between applications is the file-based method. As discussed in the textbook [1], this type of storage is allowed by applications running on all hardware systems and operating systems. A basic scenario is: one application writes data to a file while the other application reads data from the same file. Apparently, it is very simple if applications reside on the same machine. If they are running on different machines, a file-transfer mechanism between the machines should be used, for example, the file transfer protocol (i.e., commonly called FTP).

As files could have different formats accepted by different applications, a commonly understood file format by participating applications should be defined. This could become more complicated and challenges if you have at least one of these computing environments:

- Applications are running on different platforms (e.g., different operations systems).
- More than two applications will access (e.g., read and write) the shared data as the same time.
- If a new participating application has to join the party, this file-based data sharing approach would continue to increase its complexity. This is especially true when any two applications would have to share different set of data (Figure 2.1).

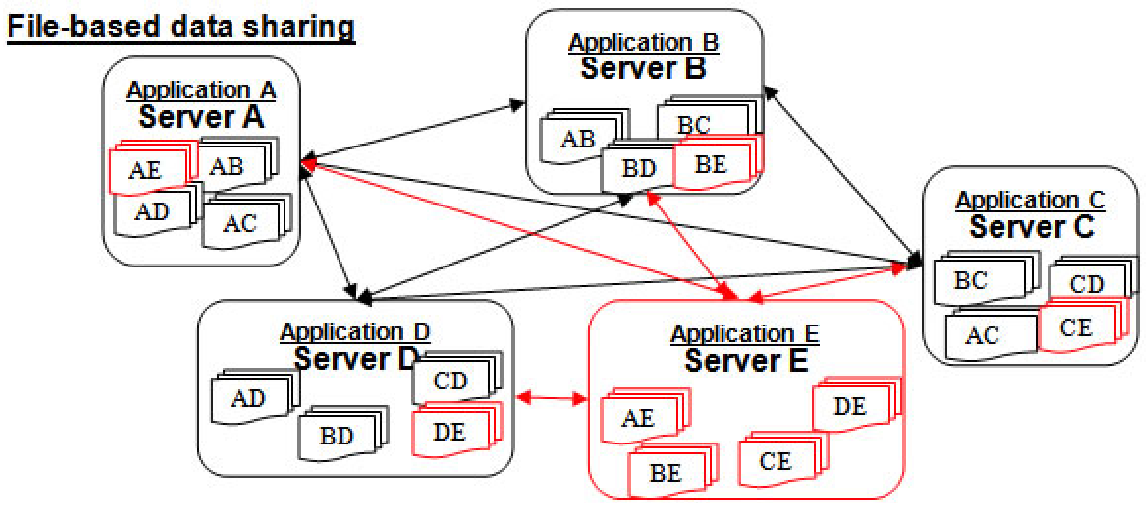

Figure 2.1: A Schematic View of File-based Data Sharing among 5 Different Applications

Figure 2.2 is an example we implemented when I was a software engineer. A company has 2 different applications, one residing on server A in California and the other on server B in Pennsylvania. The issue was how to make sure that two applications have the same user profiles as a user profile can be frequently changed on each side. We defined a common delimiter based text file accepted by both applications. Every midnight these two applications will be synchronized by sharing the updates on each side. Figure 2.1 shows just the updates from Application A. Application A exports all the updates during the day to a file. Then the file is transmitted via FTP to server B running application B. As soon as the file arrives on server B, it will be extracted to update the database of Application B. Once Application B finishes its updating, it will export a file which will be then transmitted to Server A.

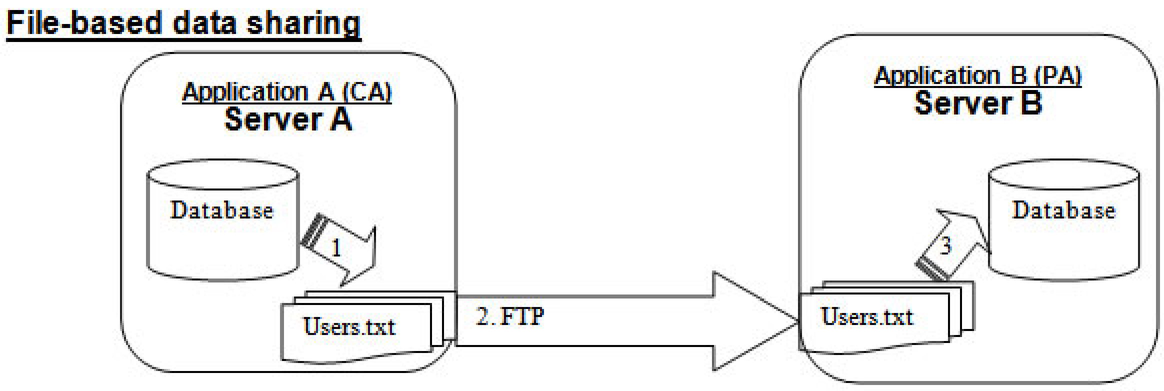

Figure 2.2: An Example of File-based Data Sharing

Common Database

As more and more applications rely on database systems to maintain data integrity and persistence, accessing data from database systems becomes pervasive. The widespread use of SQL-based relation databases results in much more shortened project development cycles in general due to the maturity of used database technologies. Different from the file-based data sharing approach, a substantive computational overhead is inevitable when a common database will be used for sharing data between applications (Figure 2.3). Application A or B has to communicate with the third program (i.e., database management system) to share their common data.

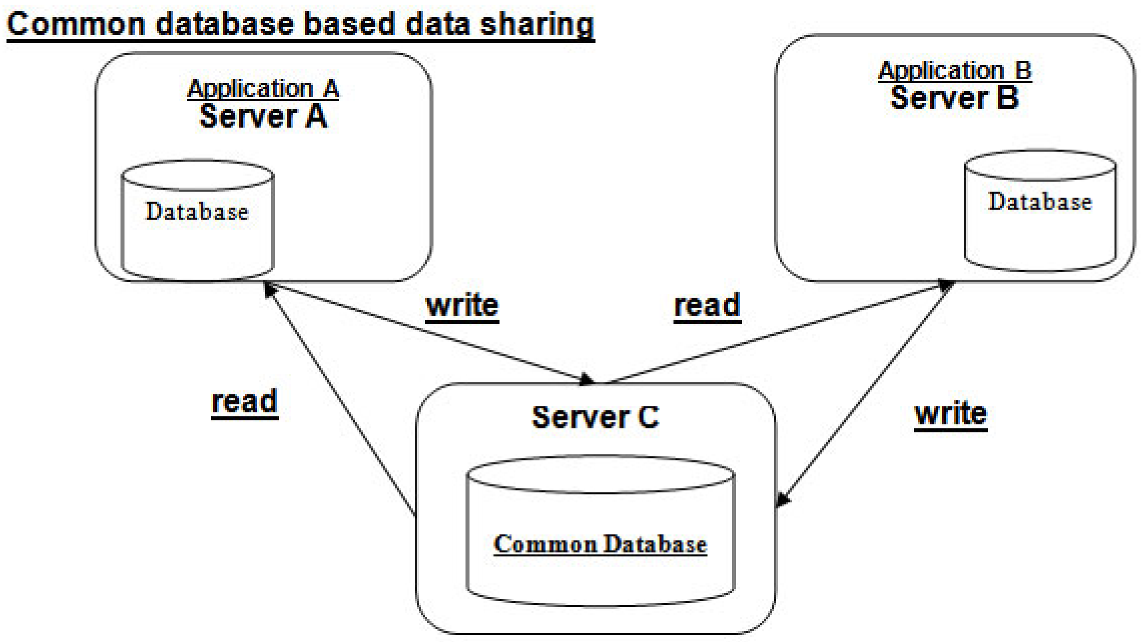

Figure 2.3: An Example of the Use of Common Database based Data Sharing

Sockets

A socket is a communication connection point that essentially has a combination of an IP address and a port number. Sockets create network connectivity, providing a real-time connection between applications. Please keep in mind, the applications that use a socket can reside on the same machine or on different networked machines. Figure 2.4 briefly shows how two processes residing on different networked machines get connected to share their common set of data using a socket.

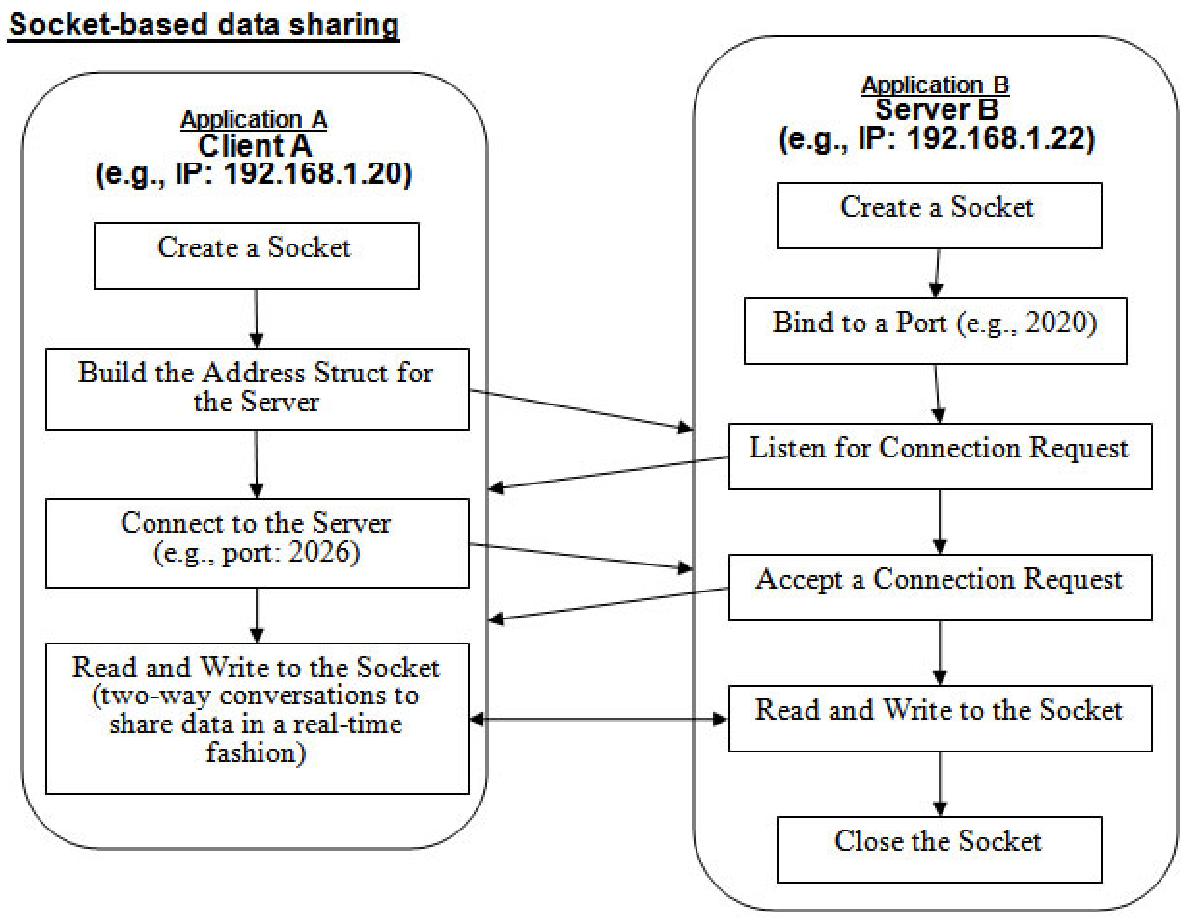
Socket-based data sharing
Application A
Client A
(e.g., IP: 192.168.1.20)
Create a Socket
Build the Address Struct for the Server
Connect to the Server (e.g., port: 2026)
Read and Write to the Socket (two-way conversations to share data in a real-time fashion)
Application B
Server B
(e.g., IP: 192.168.1.22)
Create a Socket
Bind to a Port (e.g., 2020)
Listen for Connection Request
Accept a Connection Request
Read and Write to the Socket
Close the Socket

Figure 2.4: Schematic View of a Socket for Sharing Data between Two Applications

In [2], an **Internet Protocol (IP) address** is defined as a numerical label assigned to a computing device that participates in a network. An IP address provides two main functions: device interface identification and location addressing. Its role has been characterized as follows: *"A name indicates what we seek. An address indicates where it is. A route indicates how to get there."*

According to the Internet Assigned Numbers Authority (IANA) [3], the port numbers are divided into three ranges: the Well Known Ports, the Registered Ports, and the Dynamic and/or Private Ports. Typically, the Well Known Ports are those from 0 through 1023, the Registered Ports are those from 1024 through 49151, and the Dynamic and/or Private Ports are those from 49152 through 65535. For some details of assignments, please check out http://www.iana.org (http://www.iana.org/) [3] or http://www.akerman.ca/port-table.html (http://www.chebucto.ns.ca/~rakerman/port-table.html) [4]

Online Discussion Assignment 2 is based on the following discussion: The following source codes in C and Java are examples of the use of sockets. Depends on your computer settings, they might not work. Please make one of them work, and then use the class ***Forum*** site to post your executable codes. You should also provide a simple description of your computing environment. Others can also revise/improve published codes due to enhancement or different computing environments. If you write your own or find other source codes to show a simple socket connection, please post yours on the ***Forum*** site.

*Note: These source codes were retrieved from the Internet. However, I lost the information where I had them.

Listing 2.1: Socket codes in C (server.c and client.c

```
/*server.c - code for example server program that uses TCP*/
#ifndef unix
#define WIN32
#include <windows.h>
#include <winsock.h>
#else
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#endif
#include <stdio.h>
#include <string.h>
#define PROTOPORT 5193      /*default protocol port number*/
#define QLEN 6              /*size of request queue*/
int visits = 0;              /*counts client connections*/
/*
* Program: server
*
* Purpose: allocate a socket and then repeatedly execute the following:
*         (1) wait for the next connection from a client
*         (2) send a short message to the client
*         (3) dose the connection
*         (4) go back to step (1)
* Syntax: Server [port]
*
*        port - protocol port number to use
*
* Note: The port argument is optional. If no port is specified,
* the server uses the default given by PROTOPORT.
*/
main(argc, argv)
int argc;
char *argv[];
{
    struct hostent *ptrh;          /*pointer to a host table entry*/
    struct protoent *ptrp;      /*pointer to a protocol table entry*/
    struct sockaddr_in sad;      /*structure to hold server’s address*/
    struct sockaddr_in cad;      /*structure to hold client’s address*/
    int sd,sd2;                  /*socket descriptors*/
    int port;                      /*protocol port number*/
    int alen;                      /*length of address*/
    char buf[1000];              /*buffer for string the server sends*/

#ifdef WIN32
    WSADATA wsaData;
    WSAStartup(0x0101,&wsaData);
#endif
    memset((char *)&sad,0,sizeof(sad));        /*clear sockaddr structure*/
    sad.sin_family = AF_INET;                  /*set family to Internet*/
    sad.sin_addr.s_addr = INADDR_ANY;          /*set the local IP address*/
```

```
/*Check command-line argument for protocol port and extract*/
/*port number if one is specified. Otherwise, use the default*/
/*port value given by constant PROTOPORT*/

if(argc> 1) {                /*if argument specified*/
    port = atoi(argv[1]);      /*convert argument to binary*/
} else {
    port = PROTOPORT;         /*use default port number*/
}

if (port > 0)                 /*test for illegal value*/
    sad.sin_port = htons((ushort)port);
else {                         /*print error message and exit*/
    fprintf(stderr,”bad port number %s\n”,argv[1]);
    exit(l);
}

/*Map TCP transport protocol name to protocol number*/

if (((int)(ptrp = getprotobyname("tcp”))) == 0) {
    fprintf(stderr, “cannot map \“tcp\” to protocol number”);
    exit(l);
}

/*Create a socket*/

sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if(sd<0) {
    fprintf(stderr, “socket creation failed\n”);
    exit(1);
}

/*Specify size of request queue*/

if (listen(sd, QLEN) <0) {
    fprintf(stderr,”listen failed\n”);
    exit(1);
}

/* Main server loop - accept and handle requests*/

while (1) {
    alen = sizeof(cad);
    if((sd2 = accept(sd,(struct sockaddr*)&cad, &alen))<0) {
        fprintf(stderr, "accept failed\n”);
        exit(1);
    }
visits++;
sprintf(buf,"This server has been contacted %d time%s\n”,
    visits,visits=1?“.“:“s.”);
send(sd2 ,buf,strlen(buf),0);
closesocket(sd2);
}
```

```
    /*client.c - code for example client program that uses TCP*/

#ifndefunix
#define WIN32
#include <windows.h>
#include <winsock.h>
#else
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#endif
#include <stdio.h>
#include <string.h>
#define PROTOPORT 5193            /*default protocol port number*/
extern int    errno;
char localhost[] = “locaihost”;    /*default host name*/
/* Program: client
*
*Purpose: allocate a socket, connect to a server, and print all output
*
* Syntax: client [host [port]]
*
*     host - name of a computer on which server is executing
*     port - protocol port number server is using
*
* Note: Both arguments are optional. If no host name is specified,
* the client uses ocalhost”; if no protocol port is
* specified, the client uses the default given by PROTOPORT.
*/
main(argc, argv)
int argc;
    struct hostent *ptrh;         /*pointer to a host table entry*/
    struct protoent *ptrp;      /*pointer to a protocol table entry*/
    struct sockaddr_in sad;     /*structure to hold an IP address*/
    int sd;                      /*socket descriptor*/
    int port;                     /*protocol port number*/
    char *host;                 /*pointer to host name*/
    int n;                          /*number of characters read*/
    char buf[l000];             /*buffer for data from the server*/#
#ifdefWIN32
    WSADATA wsaData;
    WSAStartup(0x0101,&wsaData);
#endif
    memset((char 5)&sad,0,sizeof(sad))      /*clear sockaddr structure*/
    sad.sin_family AF_INET;                  /*set family to Internet*/

    /*Check command-line argument for protocol port and extract*/
    /*port number if one is specified. Otherwise, use the default*/
    /*port value given by constant PROTOPORT*/

    if(argc>2) {                                  /*if protocol port specified*/
```

```
        port = atoi(argv[2]);                         /*convert to binary*/
    } else {
        port = PROTOPORT;                            /*use default port number*/
    }
    if (port > 0)                                    /*test for legal value*/
        sad.sin_port = htons((u_short)port);
    else {                                         /*print error message and exit*/
        fprintf(stderr,"bad port number %\n”,argv[2]);
        exit(l);
    }

    /*Check host argument and assign host name.*/

    if(argc>1){
        host = argv[1];                            /*if host argument specified*/
    }else{
        host = localhost;
    }

    /*Convert host name to equvalent IP address and copy to sad*/

    ptrh = gethostbyname(host);
    if (((char *)ptrh) = NULL) {
        fprintf(stderr”invalid host: %s\n”,host);
        exit(1);
    }
    memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);

    /*Map TCP transport protocol name to protocol number*/

    if (((int)(ptrp = getprotobyname("tcp”))) == 0) {
        fprintf(stderr, “cannot map \“tcp\” to protocol number”);
        exit(1);
    }
    /*Create a socket*/

    sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
    if(sd<O) {
        fprintf(stderr, “socket creation failed\n”);
        exit(l);
    }

    /*Connect the socket to the specified server*/

    if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) <0) {
        fprintf(stderr,”connect failed\n”).
        exit(1);
    }

    /*Repeatedly read data from socket and write to user’s saeen*/

    n = recv(sd, buf, sizeof(buf), 0);
    while (n> 0) {
        write(1 ,buf,n);
        n = recv(sd, buf, sizeof(buf), 0);
    }

    /*Close the socket*/

    closesocket(sd);

    /*Terminate the client program gracefully*/

    exit(0) ;
}
```

Listing 2.2: Socket codes in Java (server.java and client.java)

```
//Server.java
import java.net.*;
import java.io.*;
import java.util.Date;

public class Server {
    public final static int daytimePort = 6000;

    public static void main(String[] args) {

    ServerSocket theServer;
    Socket theConnection;
    Print Stream p;

    try{
        theServer = new ServerSocket(daytimePort);
            try{
                while (true) {
                    theConnection = theServer.accept();
                    p = new PrintStream(theConnection.getOutputStream());
                    p.println(new Date());
                    theConnection.close();
                }
            }


    catch (IOException e) {
        theServer.close();
        System.err.println(e);
    }
    }//end try
    catch(IOException e)  {
        System.err.println(e);
    }
    }
}
//Client.java

import java.net.*;
import java.io.*;

public class socketClient {
    public static void main(Stringfl args) {
        Socket theSocket;

        String hostname;
        Datainput Stream theTimeStream;
        Print Stream out;

        if (args.length > 0) {
        hostname = args[0];
        }
        else {
        hostname = “localhost”;
        }
            try {
                theSocket = new Socket(hostname, 6000);
                theTimeStream = new DataInputStream(theSocket.getInputStream());
                String theTime = theTimeStreamread.Line();
                System.out.println("It is “+ theTime +“ at “+ hostname);
                out = new PrintStream(theSocket.getOutputStream());
                out.println("Thanks”);
            }//end try

            catch(unknownHostException e) {
                System.err.println(e);
            }
            catch(IOException e) {
                System.err.println(e);
            }
    }//end main
}//end Client
```

The socket programming approach to addressing data sharing between applications has some obvious shortcomings:

- Similar to other two approaches, a common set of data will be shared when a pure socket is used. One application can’t directly invoke functions of the other application, and vice versa.
- A pure socket implementation is at a low level of programming models. The API for socket programming is difficult to use.
- Socket programming is not platform independent.
- Given that sockets create “point-to-point” connections, the data sharing becomes tightly coupled.

However, sockets have low overheads associated with applications, entailing a very efficient means of communication in real-time data sharing. Please keep in mind, in spite of certain shortcomings, sockets are fundamental to most of the modern methods of communications to share data and functionalities.

Different Methods of Sharing Functionality Between Applications (6 of 8)

Different Methods of Sharing Functionality Between Applications

Without question, sharing data is the first step for improving business operations by leveraging information technologies. However, it is extremely simple and lacks the necessary reusability and scalability for information systems to support a variety of business needs. Enterprise (application) integration is a challenging process as it typically involves many different types of distributed applications written in different programming languages and running on numerous different types of platforms.

Note that the terms methods, procedures, functions, and operations are exchangeable in this class. Depends on the technologies used at work, these terms essentially indicate the same logical and business support: functionalities enabled/provided by applications. Of course, they can be significantly different in their technical implementations. Different integration methods using different approaches should be applied to better meet the needs under different circumstances. This lesson starts with the simplest mechanism (i.e., remote procedure call) of sharing functionalities between applications, which will be the fundamentals of many other mechanisms conceptually and technically.

Remote procedure call (RPC) is a function-oriented interface based on socket programming by focusing on eliminating the need for network programming. It essentially introduces some basic concepts and features and lays out the necessary steps for sharing functionality between applications (Figure 2.5).

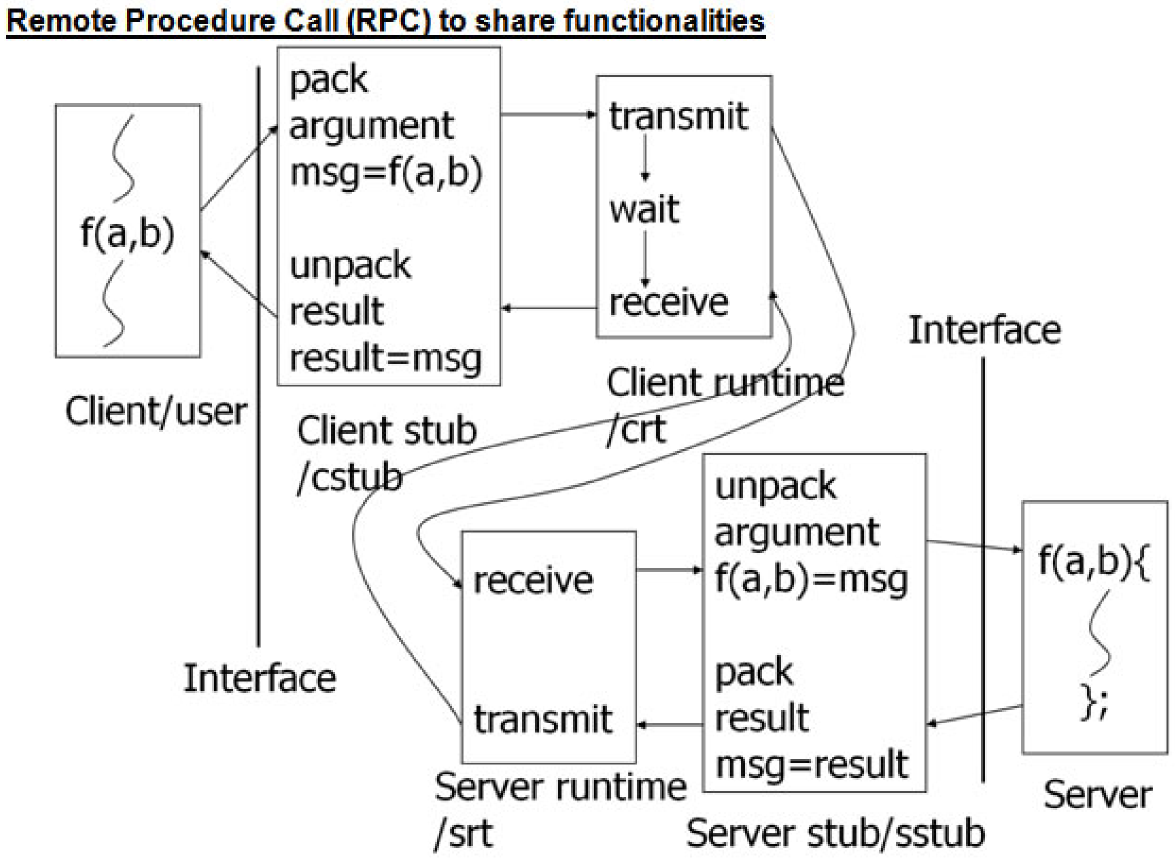

Figure 2.5: Schematic View of a RPC for Sharing Functionalities between Applications

Figure 2.5 introduces new concepts and features in order to make functionality sharing more computationally powerful and capable, ensuring the enabled mechanism scalable and reusable:

- The concept of interface
- The concept of a service provider (i.e., server) at the application level

- The introduction of stub/proxy/skeleton, which shield the programmer from system and network calls (Figure 2.6)
- The concept of marshalling/unmarshalling of arguments for transmission over the network
- The concept of platform independence via the use of external data representation (XDR), which encodes data in a machine-independent format

One can implement all the supports needed in Figure 2.5 by writing software modules in a given programming language. However, it will be extremely challenges to write supports in a heterogeneous computing environment, where you have different modules written in different languages, communicated using different networks, and running on different platforms.

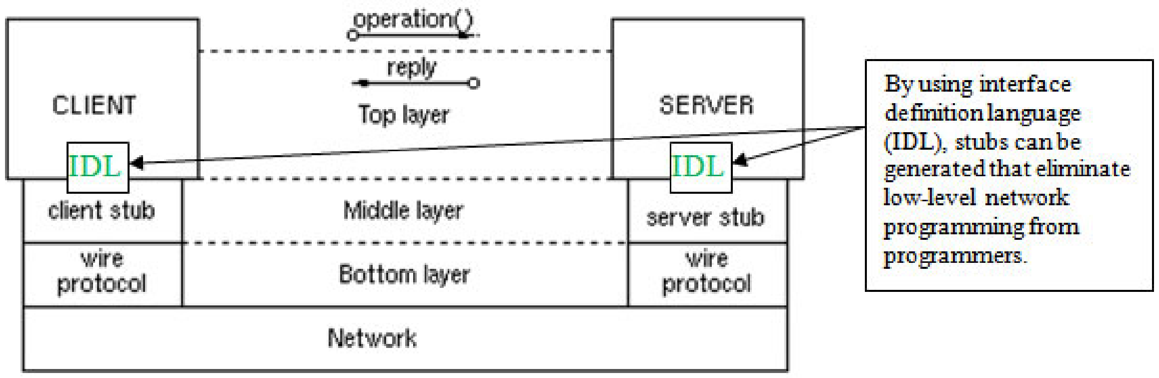
operation()
reply
Top layer
CLIENT
SERVER
IDL
IDL
client stub
Middle layer
server stub
wire protocol
Bottom layer
wire protocol
Network
By using interface definition language (IDL), stubs can be generated that eliminate low-level network programming from programmers.

Figure 2.6: Schematic Network View of a RPC for Sharing Functionalities between Applications

The textbook [1] introduces three possible types of function calls: local function calls, restricted remote function calls (restricted RPC) that involves two applications running on the same computer, and more general remote procedure calls between two applications running on two different network-connected computers (Figure 2.6). Fortunately, the IT industry started to “standardize” solutions by providing libraries/tools to the end users. RPC was one of them at the very beginning. We focus on the third type using Microsoft Developer Network (**MSDN**) library [5] as the third type is the origin of modern integration patterns for distributed enterprise IT systems.

By focusing on the need for programming language independence, an **interface definition language (IDL)** is essentially a specification language used to describe a software component's interface [2], aimed at enabling communication in a “standardized” manner between software components that are implemented using different programming language – for example, between components written in C++ and components written in Java. In Figure 2.6, the machines at either end of the "link" may be using different operating systems and computer languages. It is the IDLs that offer a bridge between the two different systems (Figure 2.7).

Listing 2.3 is the simplest example of an IDL, which defines the interface to be used by software components residing on two computers.

```
//fi1e hello.idl
[

uuid(7a98c250-6808-11cf-b73b-00aa00b677a7),
version(1 .0)

]
interface hello
{
void HelloProc([in. string] unsigned char*pszString);
void Shutdown(void);
```

Figure 2.7 shows how MSDN RPC works using the simplest example defined in Listing 2.3 [6]. “The MIDL design specifies two distinct files: the Interface Definition Language (IDL) file and the application configuration file (ACF). These files contain attributes that direct the generation of the C-language stub files that manage the remote procedure call (RPC). The IDL file contains a description of the interface between the client and the server programs. RPC applications use the ACF file to describe the characteristics of the interface that are specific to the hardware and operating system that make up a particular operating environment. The purpose of dividing this information into two files is to keep the software interface separate from characteristics that affect only the operating environment.” [7]

MIDL compiler will generate all the needed supports: a header file, a client side stub, and a server side stub. Apparently, the stubs provide the supports of all the network programming and data conversions needed in this RPC functionality sharing, which makes the end user possible to focus on business logic implementations rather than dealing with the complexity of low level network programming and data conversions.

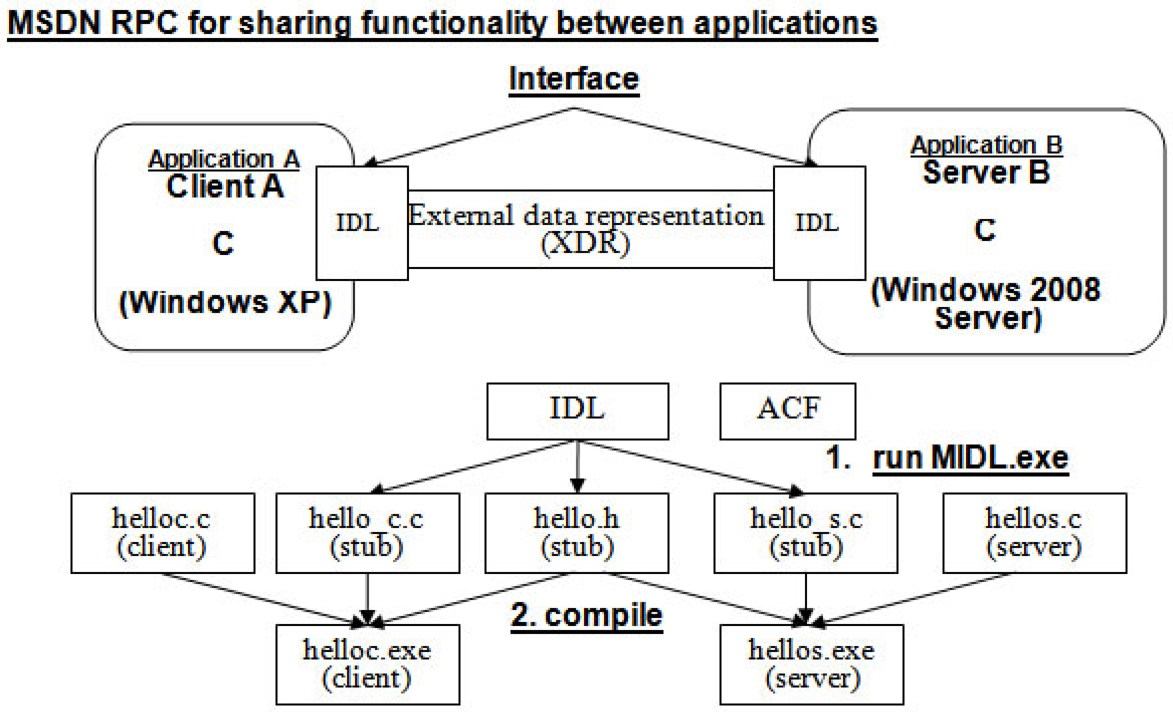

Figure 2.7: An Example of RPC for Sharing Functionality between Applications

Basic Concepts

Socket communications, remote procedure calls, and remote invocation methods [8] are the main discussions in this Lesson, which will lay out the foundation for our future Lessons on popular methods of enterprise integration.

References

[1] Roshen, W. 2009. SOA-based Enterprise Integration: A Step-by-Step Guide to Services-based Application Integration. McGraw-Hill, New York, USA.

[2] Wikipedia:

- IP Address: http://en.wikipedia.org/wiki/IP_address,
- RPC: http://en.wikipedia.org/wiki/Remote_procedure_call,
- IDL: http://en.wikipedia.org/wiki/Interface_description_language,
- Peer-to-peer: http://en.wikipedia.org/wiki/Peer-to-peer,

[3] The Internet Assigned Numbers Authority (IANA): http://www.iana.org.

[4] Port Table: http://www.akerman.ca/port-table.html,

[5] How RPC Works: http://msdn.microsoft.com/en-us/library/aa373935(VS.85).aspx

[6] MSDN RPC Tutorial: http://msdn.microsoft.com/en-us/library/aa379010(VS.85).aspx.

[7] IDL and ACF: http://msdn.microsoft.com/en-us/library/aa378708(VS.85).aspx.

[8] Remote Method Invocation: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp