Metering SDK User's Guide

Version 1.1
January 2000

Contents

Chapter 1 — Introduction

Overview

The Metering Software Development Kit (SDK) is an object-oriented, cross-platform toolkit that enables independent software vendors (ISVs), systems integrators, and hardware manufacturers to integrate their applications seamlessly with a network service provider's (NSP) billing and management systems. The SDK's small memory footprint and optimized performance make it suitable for use in the most demanding and resource-constrained environments. The SDK sends metering information to a scaleable, highly-distributable Metering Server that persistently stores the data. This enables you to focus on your core technology rather than expending valuable resources on custom development.

The sections in this chapter are:

- Metering SDK Features
- MetraTech Concepts
- System Requirements for both NT and UNIX
- Installing and uninstalling the Metering SDK for both NT and UNIX
- Testing the Metering SDK After Installation

Metering SDK Features

- **Easy to Integrate:** The SDK supports interfaces for all popular development environments and has a straightforward API that enables service usage, such as a voice call, to be metered in as few as 20 lines of code. The SDK is provided royalty-free and at no charge. In addition, MetraTech hosts an Internet accessible reference server that enables developers to test and debug SDK integration against a stable reference environment.

- **Standards-Based:** The SDK uses the **M**etered **S**ystems **I**nformation e**X**change (**MSIX**) standard to communicate with MetraTech's Metering Server. MSIX is an open protocol that provides a common interface for applications to exchange detailed usage information with network billing and management systems. MetraTech's MSIX implementation enables plug-and-play billing capabilities between ISVs and NSPs. For more information on MSIX, see www.msix.org. The site contains the protocol specification, a white paper, and a FAQ.

- **Secure:** The SDK uses RSA™ technology to encrypt all communication between the SDK and the Metering Server. As the SDK can be configured to use HTTP/HTTPS as its transport protocol, metered information can be sent through firewalls and/or proxy servers.

- **Robust:** The SDK supports a robust set of transaction types that support metering a wide range of services from voice calls, to fax broadcasts with tens of thousands of recipients, to quality of service (QOS) bandwidth usage, to multicast video broadcasts.

- **Failover and Local Mode Support:** In the event that communication with the primary Metering Server is lost, the SDK has the ability to failover to alternate backup Metering Servers. Using the SDK, a developer can specify a prioritized list of Metering Servers to support failover situations. Furthermore, if no Metering Servers are available due to a network outage, the SDK can be configured to automatically store the data in a local file and forward it as soon as a Metering Server becomes available.

- **Global:** All MetraTech technologies support localization for international applications. The SDK supports the metering of information in any character set via Unicode.

MetraTech Concepts

This section provides a high-level overview of fundamental MetraTech concepts including:

- Communications

- MSIX
  - Clients and Servers
  - Services
  - Properties
  - Sessions
  - Unique Ids

Communications

The SDK communicates with the Metering Server via the MSIX protocol and, by default, uses HTTP tunneling. Tunneling enables the SDK to leverage SSL for encryption and to pass information through most firewall configurations. A subsequent release of the SDK will allow HTTP tunneling to be optionally disabled in situations where tunneling is not required.

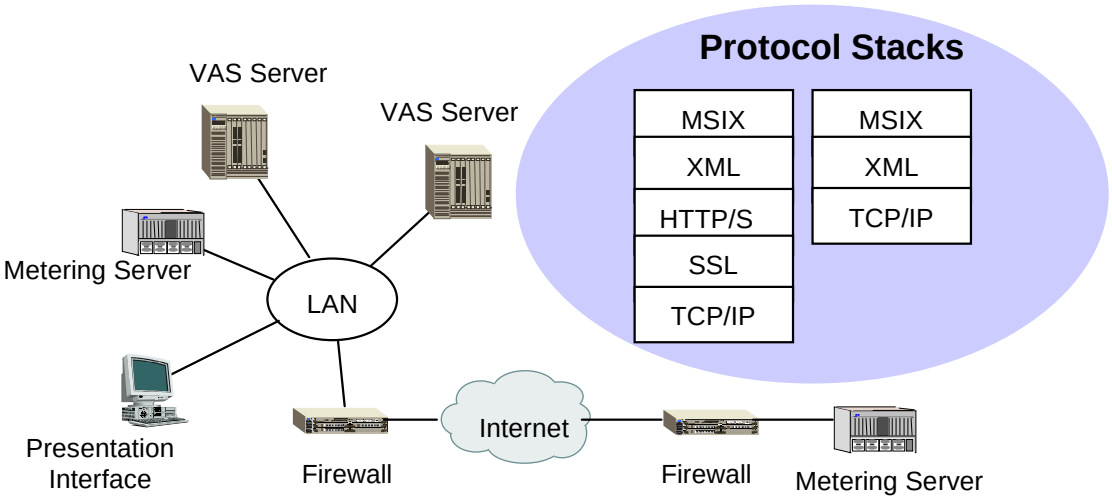


Figure 1: Diagram of MetraTech's Communications Architecture

MSIX

MSIX is an open protocol that enables NSPs to meter detailed usage information and assign charges for value-added services. The protocol provides a common interface for applications to easily exchange detailed usage information with network billing and management systems. MSIX is based on the e**X**tensible **M**arkup **L**anguage (XML). For more information on MSIX, visit www.msix.org.

The SDK implements MSIX and insulates the developer from the details of the protocol. The following six MSIX concepts are important for the SDK developer to understand:

Clients and Servers

A network entity is a device that is connected to a network. It may act as an MSIX client or an MSIX server. An entity is an MSIX server if it accepts MSIX sessions and an MSIX client if it generates them. An application server is an entity that provides application services to clients. Application servers are typically clients of MSIX servers.

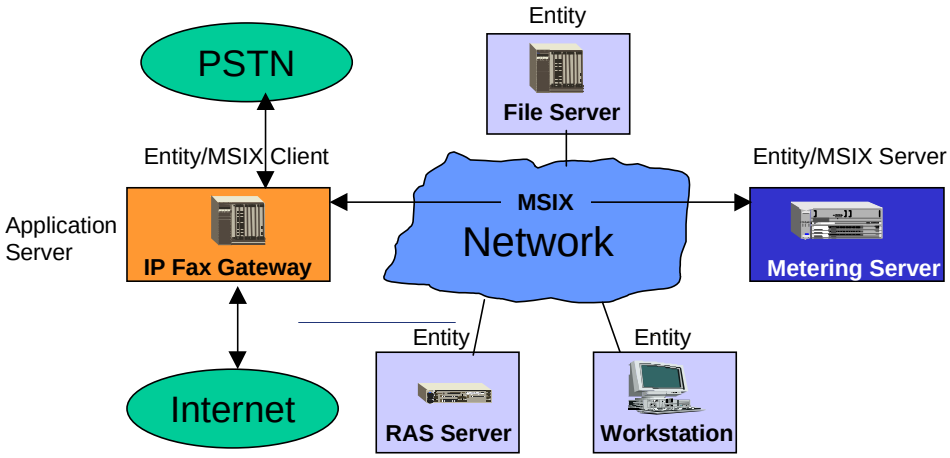


Figure 2: MSIX Clients and Servers

Services

A service is a task that is performed by an application for a client. The task performed may be anything: IP-PSTN fax/voice calls, video streams, quality of service (QoS) routing, virtual private networks (VPN), outsourced remote access, web hosting, data storage, etc.

Services may be either atomic or compound, as defined below. Additional information about Atomic and Compound services can be found in the ***Atomic or Compound*** section on page 60.

- **Atomic services** are those services whose usage can be metered completely in a single transaction. Examples of atomic services are point-to-point faxes, voice calls, and video streams.

- **Compound service** usage involves the composition of multiple services. These services can be atomic services or can themselves be other compound services. Thus, a compound service transaction is composed of a hierarchical collection of related service transactions as shown in Figure 3 below. Examples include fax broadcasts, conference calls and multicast video streams. There is no limit on the number of sub-services, the depth of the hierarchy or the type of services encompassed.

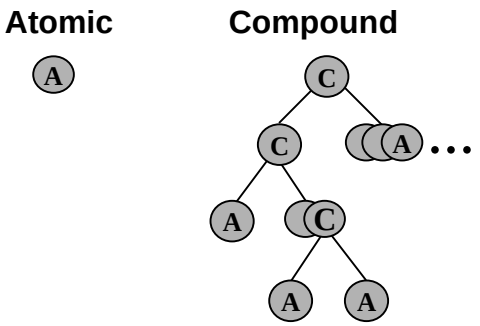


Figure 3: Atomic and Compound Services

Properties

Properties describe the quantities that can be metered for a particular service. A property is represented by a name/value pair, where the name is a unique, human-readable identifier understood within the context of a service. Examples of property names include "bytes-sent" and "duration." A property therefore, would be represented by the pair "duration=120" or "bytes-set=1024". MSIX clients communicate usage information by transmitting names and associated values to the Metering Server.

Sessions

A session represents the use of a service by a client as a collection of properties. To meter service usage, clients create sessions and add properties to those sessions. Properties can be added to or updated until the session is "closed".

Unique IDs

Unique IDs define MSIX sessions and to create the hierarchies inherent in compound services. UIDS are generated transparently by the SDK for the SDK user.

NT System Requirements

On an NT system, the Metering SDK requires the following:

- Windows NT 4.0 (service pack 5) or higher.
- 5MB of free disk space.
- 32 MB RAM (recommended minimum).
- Microsoft Visual C++ 6.0 is required to build applications that use the Metering SDK.
- Microsoft Internet Explorer (version 4.0 or higher) or Netscape Navigator (version 3.0 or higher).

UNIX System Requirements

On a UNIX system, the Metering SDK requires the following:

- SUN Solaris 2.5 or above
- SUN C compiler 4.2 or above

Installing the Metering SDK on UNIX

The UNIX metering SDK package is in one compressed file "unixmeteringSDK.tar.gz". There is no need to change any environment variable or install any other third party software products.

To install the Metering SDK on UNIX:

1. Run "gunzip unixmeteringSDK.tar.gz", it will create a file "unixmeteringSDK.tar".
2. Run "tar xvf unixmeteringSDK.tar", it will create a directory "unixmeteringSDK".
3. "cd unixmeteringSDK" and do "make clean".
4. Then "make", it will create an executable file called simple.

Installing the Metering SDK on NT

To install the Metering SDK on NT:

1. Launch the **mtsdk_nt.exe** executable (available from partner.metratech.com or from the CD).
2. You will be asked to agree to the Metering SDK license agreement.
3. After you have agreed to the license, you will be prompted to indicate the directory in which to store the libraries, DLL, sample code and documentation. You can store these in any directory that you wish to. For the purposes of this document, we will refer to this directory as INSTALLDIR.
4. Next, you will be prompted for the name of the Program Folder where shortcuts to the sample application, documentation and the uninstall application will be created.
5. After all files have been installed you will have the option of viewing the README file contained in the Metering SDK. The release notes in this file contain last minute additions to the documentation so it is important that you consult this file if you encounter any problems when using the Metering SDK.

   Note: Install shield puts uninstall information in the registry.

Uninstalling the Metering SDK

To remove the Metering SDK from your system, select the *Programs → MetraTech Metering SDK → Uninstall Metering SDK* menu item from the *Start* menu.

Testing the Metering SDK after Installation

The SDK contains a test program that meters a very simple service. The service consists of the following properties:

- AccountName
- Description
- Units

The next four sections describe the steps required to test the installation:

Step1: Find the account information necessary to run the test program.

Before you run the test program, you should know the following:

- Name of the Metering Server
- Username and Password for the Metering Server
- Account Name

If you do not have this information, refer to the ***Partner Data Sheet*** appendix on page 67.

Step 2: Run the test program

By default, the Metering SDK installation program creates a shortcut to a console application test program which you can use to test your installation.

For NT, there are the following two ways to run the program:

- **Running the test program from the Windows *Start* menu:** The test program can be run by selecting the *Programs* → *MetraTech Metering SDK* → *Metering SDK Test Program* menu item from the *Start* menu.
  - **Running the test program from DOS:** From DOS, change to the installation BIN directory and type simple.exe, as shown below:

    ```
    C:\>cd INSTALLDIR\bin
    C:\DevTools\MT-SDK\bin>simple.exe
    ```

    Note: It is important to change the directory to INSTALLDIR\bin, as simple.exe relies on a DLL on which is in this directory and has not been placed in your PATH.
  - **Running the test program from UNIX:** From DOS, change to the installation BIN directory and type simple.exe, as shown below:

    ```
    >cd INSTALLDIR\bin
    \DevTools\MT-SDK\bin>simple.exe
    ```

Step 3: Enter the test information into the test program.

After you launch the test program, you will be prompted to enter the following information:

- Server host name: The name of the destination Metering Server.
- Server username: The username to use when sending sessions to the Metering Server. If there is no username, press **Enter**.
- Server password: The password to use when sending sessions to the Metering Server.
- Failover server prompt: To enter an alternate server to be accessed in case of failover.
- AccountName: The account to which the session will be metered.
- Transaction description: The description of the transaction.
- Units: The number of units.

The following is an example execution of the test program. User responses are in bold.

```
C:\DevTools\MT-SDK\bin>simple.exe
MetraTech simple metering example.
Server host name: partner.metratech.com
Server username: partnerCorp
Server password: partnerPass
Units (floating point number): 3.5
Transaction description: This is a test
Account name: 123
Units (floating point number): 3.5
Metering with the following information:
Host: partner.metratech.com
Username: partnerCorp
Password: partnerPass
Account name: 123
Description: This is a test
Units (floating point number): 3.5
The session has been metered!
```

```
Press any key to continue...
```

Please see the Troubleshooting section on page 67 if problems arise when running this test program.

Step 4: Verify that the test program worked by viewing your transaction.

Assuming that the test program returned with success, you should verify that the transaction was metered by following the steps outlined in this section:

1. Use an Internet browser, Microsoft Internet Explorer 3.0 (or higher) or Netscape Navigator 3.0 (or higher), to access the Presentation Server's logon page. Please see the *Partner Data Sheet* for detailed web address, logon and password information.
2. Enter a logon and password and click on the *Enter* button, as shown in Figure 4 below. This enables the Presentation Server to display metered usage for your particular account. This account will be the same account as that identified by the AccountID that was used to meter the test service.

Figure 4: Enter logon and password

3. To view your transaction, click on the *Usage* button from the main menu.
4. Click Your Bill to show the available services as shown in Figure 5 below.

Figure 5: Your bill

5. Select **Test Service**. The browser displays a new line with one or more metered product line items. The product associated with the test program is: *Test Service Product*. Click on the icon to the left of this line, to display a listing of metered transactions associated with this product.
6. On this screen, you should see your test transaction, as shown in Figure 6 below.

Figure 6: Test service detail

7. Please check the Date, Time, Description, and Amount fields to verify that the transaction matches what you just metered.

   If you do not see your transaction, please check that the AccountName used to send the test matches the AccountName on your *Partner Data Sheet.*

Chapter 2 — Quick Start

Overview

The sections in this chapter are:

-
-

```
*************************************************************************/

// Include metering objects
#include <mtsdk.h>

#ifdef WIN32
#include <conio.h>                                          // for _getch
#endif
#include <string.h>
#include <iostream.h>

#ifdef UNIX
#include <curses.h>
#define _getch getch
#endif

int bUseSynch=FALSE ;

class SimpleMeter
{
public:
  // construct the default configuration object, and use that to
  // construct the meter object
  SimpleMeter() : mMeter(mConfig)
  { }

  // entry point called from main
  void TestSimple(int argc, char * argv[]);

private:

  // meter the transaction based on the info entered by the user
  // returns NULL if there is no error
  MTMeterError * MeterIt(const char * host, const char * username,
    const char * password, const char * accountname,
    const char * desc, float units);

  // print an error if there is one
  void PrintError(const char * prefix, const MTMeterError * err);

  // wait for a key and return its value
  static int GetKey();

  // configuration object - used to initialize the Metering SDK
  // with HTTP transport
  MTMeterHTTPConfig mConfig;

  // the entry point to the Metering SDK - all Metering objects
  // are created from here.
```

```
  MTMeter mMeter;


};


void SimpleMeter::TestSimple(int argc, char * argv[])
{
  cout << "MetraTech simple metering example." << endl;

  // read the server host name
  cout << "Server host name: ";
  char host[256];
  cin.getline(host, sizeof(host));

  // read the server username
  cout << "Server username: ";
  char username[256];
  *username = 0;
  cin.getline(username, sizeof(username));

  // read the server password
  cout << "Server password: ";
  char password[256];
  *password = 0;
  cin.getline(password, sizeof(password));


    // read the backup server host name
  cout << "Failover Server host name <return> for none: ";
  char backuphost[256];
  cin.getline(backuphost, sizeof(backuphost));

  char backupusername[256];
  char backuppassword[256];
  if (strlen(backuphost))
  {
       // read the back server username
       cout << "Failover Server username: ";
       *backupusername = 0;
       cin.getline(backupusername, sizeof(backupusername));

       // read the back up server password
       cout << "Failover Server password: ";
       *backuppassword = 0;
       cin.getline(backuppassword, sizeof(backuppassword));
  }


  // read the user's account name
  cout << "Account Name: ";
  char accountname[256];
  cin.getline(accountname, sizeof(accountname));

  // read the description
  cout << "Transaction description: ";
  char desc[256];
  cin.getline(desc, sizeof(desc));

  // read the units
  cout << "Units (floating point number): ";
  float units;
  cin >> units;
```

```
  // check to see if the user wants to send the session securely
  cout << "Do you want the session sent using SSL? [y/n] ";
  char sendViaSSL[10];
  int bUseSSL=FALSE ;
  *sendViaSSL = 0;
  cin.getline(sendViaSSL, sizeof(sendViaSSL));
  cin.getline(sendViaSSL, sizeof(sendViaSSL));

  // if no indicator was entered ... do not send the session
  // using SSL
  if (strlen(sendViaSSL) == 0)
  {
    bUseSSL = FALSE ;
  }
  else if (strlen(sendViaSSL) == 1)
  {
    if (sendViaSSL[0] == 'y' || sendViaSSL[0] == 'Y')
    {
      bUseSSL = TRUE ;
    }
    else if (sendViaSSL[0] == 'n' || sendViaSSL[0] == 'N')
    {
      bUseSSL = FALSE ;
    }
    else
    {
      cout << "Invalid response. Not sending the session using SSL." << endl ;
      bUseSSL = FALSE ;
    }
  }
  else
  {
    cout << "Invalid response. Not sending the session using SSL." << endl ;
    bUseSSL = FALSE ;
  }

    // prompt for synchronous mode
  cout << "Do you want the session sent using Synchronous mode (the pipeline must
be running) ? [y/n] ";
  char sendSynch[10];

  *sendSynch = 0;
  cin.getline(sendSynch, sizeof(sendSynch));

  // if no indicator was entered ... do not send the session
  // using SSL
  if (strlen(sendSynch) == 0)
  {
    bUseSynch = FALSE ;
  }
  else if (strlen(sendSynch) == 1)
  {
    if (sendSynch[0] == 'y' || sendSynch[0] == 'Y')
    {
      bUseSynch = TRUE ;
    }
    else if (sendSynch[0] == 'n' || sendSynch[0] == 'N')
    {
      bUseSynch = FALSE ;
    }
    else
```

```
    {
      cout << "Invalid response. Not sending the session using Sychronous mode." <<
endl ;
      bUseSynch = FALSE ;
    }
  }
  else
  {
    cout << "Invalid response. Not sending the session using Synchronous mode." <<
endl ;
    bUseSynch = FALSE ;
  }

  // summarize the metering information
  cout << endl << "Metering with the following information:" << endl;
  cout << "Host: " << host << endl;
  cout << "Username: " << username << endl;
  cout << "Password: " << password << endl;
  cout << "Account Name: " << accountname << endl;
  cout << "Description: " << desc << endl;
  cout << "Units: " << units << endl;

  if (bUseSSL == TRUE)
  {
    cout << "Using SSL to send the session." << endl ;
  }
  else
  {
    cout << "Not using SSL to send the session." << endl ;
  }

  if (bUseSynch == TRUE)
  {
    cout << "Using Synchronous mode to send the session." << endl ;
  }
  else
  {
    cout << "Not using Synchronous to send the session." << endl ;
  }

  // initialize the SDK
  if (!mMeter.Startup())
  {
    MTMeterError * err = mMeter.GetLastErrorObject();
    PrintError("Could not initialize the SDK: ", err);
    delete err;
    return;
  }

  // if we are using SSL to send the session ... pass the
  // appropriate parameters correctly (port and secure flag)
  if (bUseSSL == TRUE)
  {
    mConfig.AddServer(0,                                          // priority
(highest)
      host,                    // hostname
      MTMeterHTTPConfig::DEFAULT_HTTPS_PORT,               // port (from mtsdk.h)
      TRUE,                              // secure (yes)
      username,                         // username
      password);               // password
  }
  // otherwise ... we're not using SSL to send the session ... pass the
  // appropriate parameters correctly (port and secure flag)
```

```
  else
  {
    mConfig.AddServer(0,                                          // priority
(highest)
      host,                                     // hostname
      MTMeterHTTPConfig::DEFAULT_HTTP_PORT,                       // port (default
plaintext HTTP)
      FALSE,                            // secure? (no)
      username,                         // username
      password);                  // password
  }


  // Add the FAILOVER server if requested
  // if we are using SSL to send the session ... pass the
  // appropriate parameters correctly (port and secure flag)
  if (strlen(backuphost))
  {
       if (bUseSSL == TRUE)
       {
              mConfig.AddServer(0,                                        //
priority (highest)
                    backuphost,                    // hostname
                    MTMeterHTTPConfig::DEFAULT_HTTPS_PORT,               // port
(from mtsdk.h)
                    TRUE,                             // secure (yes)
                    backupusername,                        // username
                    backuppassword);               // password
       }
       // otherwise ... we're not using SSL to send the session ... pass the
       // appropriate parameters correctly (port and secure flag)
       else
       {
              mConfig.AddServer(0,                                                //
priority (highest)
                    backuphost,                                       // hostname
                    MTMeterHTTPConfig::DEFAULT_HTTP_PORT,                        //
port (default plaintext HTTP)
                    FALSE,                            // secure? (no)
                    backupusername,                          // username
                    backuppassword);                  // password
       }
  }

  // meter the session
  MTMeterError * err = MeterIt(host, username, password, accountname, desc, units);
  if (!err)
    cout << "The session has been metered!" << endl;
  else
  {
    PrintError("The session has not been metered: ", err);
    delete err;
  }


  // close the sdk
  mMeter.Shutdown();

  // pause, in case this is run from a shortcut
  cout << "Press any key to continue..." << endl;
  (void) GetKey();
  return;
}
```

```
MTMeterError * SimpleMeter::MeterIt(const char * host, const char * username,
                                    const char * password, const char *
accountname,
                                    const char * desc, float units)
{

  // service name is "metratech.com/TestService"
  MTMeterSession * session = mMeter.CreateSession("metratech.com/TestService");


  session->SetResultRequestFlag(bUseSynch);

  // set the session's time field to the current time.
  time_t t = time(NULL);

  // these property names have to match those on the server
  if (!session->InitProperty("AccountName", accountname)
    || !session->InitProperty("Description", desc)
    || !session->InitProperty("Units", units)
    || !session->InitProperty("Time", t))
  {
    MTMeterError * err = session->GetLastErrorObject();
    delete session;

    return err;
  }

  // send the session to the server
  if (!session->Close())
  {
    MTMeterError * err = session->GetLastErrorObject();
    delete session;

    return err;
  }

  if (session->GetResultRequestFlag())
  {
      MTMeterSession * ResultSession = session->GetSessionResults();
      if (ResultSession)
      {
             float Amount;
             float perunit;
             cout.setf (ios::showpoint);
             cout.setf (ios::fixed, ios::floatfield);
             cout.precision (2);
             if (ResultSession->GetProperty ("_Amount", Amount)  &&
                          ResultSession->GetProperty ("PerUnit", perunit))
                    cout << endl << "A per unit cost of " << perunit << " was
applied, resulting in a charge of " << Amount << endl << endl;
      }
  }

  // sessions created with CreateSession must be deleted.
  delete session;

  // success! no error to return
  return NULL;
}


void SimpleMeter::PrintError(const char * prefix, const MTMeterError * err)
```

```
{
  cerr << prefix << ": ";
  if (err)
  {
    int size = 0;
    err->GetErrorMessage((char *) NULL, size);
    char * buf = new char[size];
    err->GetErrorMessage(buf, size);

    cerr << hex << err->GetErrorCode() << dec << ": " << buf << endl;
  }
  else
    cerr << "*UNKNOWN ERROR*" << endl;
}


// wait for a key
int SimpleMeter::GetKey()
{
#ifdef _WIN32
  return _getch();
#else
  return getchar();
#endif
}



int main(int argc, char * argv[])
{
  SimpleMeter meter;
  meter.TestSimple(argc, argv);
  return 0;
}
```

Chapter 3 — Creating C++ Applications for the Metering SDK

Overview

This chapter describes the build environment and provides sample C++ applications for the SDK. You can use these sample applications as a starting point for your own integration with the SDK.

The sections in this chapter are:

- Build Environment
- Sample 1: C++ Application for an Atomic Session
- Sample 2: C++ Application for a Compound Session

Build Environment

The current version of the Metering SDK was built with:

- Windows NT 4.0 with the Microsoft Visual C++ compiler, version 6.0. The libraries and DLLs included were built with Microsoft Visual C++, version 6.0.
- Sun Solaris 2.5 with the Sun CC compiler, version 4.2.

Helpful Hint:
The following sample applications have the correct build environment settings. If you want to change the settings, see *Step 4: Implementation* in the *Building Services* chapter.

Sample 1: C++ Application for an Atomic Session

The Sample 1 application shows basic functionality for metering an atomic session for the metratech.com/TestService service. The Sample 1 application prompts the user for input, meters the session, and exits. Components of the service are described in the following sections.

Time

The Metering SDK requires that all times are represented in GMT. Before submitting time values to the Metering server, you must make applicable conversions from local time to GMT. The components of the service are described in the following sections.

Header Files

The directive below includes the single header file required by users of the Metering SDK. **mtsdk.h** must be within your include path. For details about how to add this file to your include path, see step 1 in the *Building Services* chapter. The project file for the Sample 1 application already has the file in its include path.

```
#include <mtsdk.h>
```

The header file **sdk_msg.h** holds the definition for each of the error codes that are stored in **sdk_msg.dll**. This header file must be included to interpret error codes returned from the Metering SDK. The Sample 1 application does not include this file.

Application Class

The application class descriptions follow:

- The **SimpleMeter** class is the main application class.
- The **TestSimple** function is the entry point to the Sample 1 application. It is called by **main()**.
- **PrintError** prints errors if they occur.
- The **MTMeter** object is used to create metering objects, and an **MTMeterHTTPConfig** object is used to configure the **MTMeter** object.
- **MeterIt** is called by **TestSimple** to send the data read from the user to the Metering server using the SDK. **MeterIt** does the real work of the Sample 1 application.

Sample code follows:

```
class SimpleMeter
{
public:
  SimpleMeter() : mMeter(mConfig)
  { }

  void TestSimple(int argc, char * argv[]);

// meter the transaction based on the info entered by the user
  // returns NULL if there is no error
  MTMeterError * MeterIt(const char * host, const char * username,
    const char * password, const char * accountname,
    const char * desc, float units);

  // print an error if there is one
  void PrintError(const char * prefix, const MTMeterError * err);

  MTMeterHTTPConfig mConfig;

  MTMeter mMeter;
};
```

Obtaining Data

The **TestSimple** function is called by **main()**. **TestSimple** prompts the user for the information necessary to submit a TestService session. C++ streams are used for input and output.

To keep the Sample 1 application basic, only code specific to the Metering SDK is shown. More extensive stream error handling and data validation is normally required.

Sample code follows:

```
void SimpleMeter::TestSimple(int argc, char * argv[])
{
  cout << "MetraTech simple metering example." << endl;

  // read the server host name
  cout << "Server host name: ";
  char host[256];
  cin.getline(host, sizeof(host));

  // read the server username
  cout << "Server username: ";
  char username[256];
  *username = 0;
  cin.getline(username, sizeof(username));

  // read the server password
  cout << "Server password: ";
  char password[256];
  *password = 0;
  cin.getline(password, sizeof(password));

// read the backup server host name
  cout << "Failover Server host name <return> for none: ";
  char backuphost[256];
  cin.getline(backuphost, sizeof(backuphost));

  char backupusername[256];
  char backuppassword[256];
  if (strlen(backuphost))
  {
       // read the back server username
       cout << "Failover Server username: ";
       *backupusername = 0;
       cin.getline(backupusername, sizeof(backupusername));

       // read the back up server password
       cout << "Failover Server password: ";
       *backuppassword = 0;
       cin.getline(backuppassword, sizeof(backuppassword));
  }

  // read the user's account name
  cout << "Account Name: ";
  char accountname[256];
  cin.getline(accountname, sizeof(accountname));

  // read the description
  cout << "Transaction description: ";
  char desc[256];
  cin.getline(desc, sizeof(desc));

  // read the units
```

```
  cout << "Units (floating point number): ";
  float units;
  cin >> units;

  // check to see if the user wants to send the session securely
  cout << "Do you want the session sent using SSL? [y/n] ";
  char sendViaSSL[10];
  int bUseSSL=FALSE ;
  *sendViaSSL = 0;
  cin.getline(sendViaSSL, sizeof(sendViaSSL));
  cin.getline(sendViaSSL, sizeof(sendViaSSL));

  // if no indicator was entered ... do not send the session
  // using SSL
  if (strlen(sendViaSSL) == 0)
  {
    bUseSSL = FALSE ;
  }
  else if (strlen(sendViaSSL) == 1)
  {
    if (sendViaSSL[0] == 'y' || sendViaSSL[0] == 'Y')
    {
      bUseSSL = TRUE ;
    }
    else if (sendViaSSL[0] == 'n' || sendViaSSL[0] == 'N')
    {
      bUseSSL = FALSE ;
    }
    else
    {
      cout << "Invalid response. Not sending the session using SSL." << endl ;
      bUseSSL = FALSE ;
    }
  }
  else
  {
    cout << "Invalid response. Not sending the session using SSL." << endl ;
    bUseSSL = FALSE ;
  }

    // prompt for synchronous mode
  cout << "Do you want the session sent using Synchronous mode (the pipeline must
be running) ? [y/n] ";
  char sendSynch[10];

  *sendSynch = 0;
  cin.getline(sendSynch, sizeof(sendSynch));

  // if no indicator was entered ... do not send the session
  // using SSL
  if (strlen(sendSynch) == 0)
  {
    bUseSynch = FALSE ;
  }
  else if (strlen(sendSynch) == 1)
  {
    if (sendSynch[0] == 'y' || sendSynch[0] == 'Y')
    {
      bUseSynch = TRUE ;
    }
    else if (sendSynch[0] == 'n' || sendSynch[0] == 'N')
    {
      bUseSynch = FALSE ;
```

```
    }
    else
    {
      cout << "Invalid response. Not sending the session using Synchronous mode."
<< endl ;
      bUseSynch = FALSE ;
    }
  }
  else
  {
    cout << "Invalid response. Not sending the session using Synchronous mode." <<
endl ;
    bUseSynch = FALSE ;
  }

  // summarize the metering information
  cout << endl << "Metering with the following information:" << endl;
  cout << "Host: " << host << endl;
  cout << "Username: " << username << endl;
  cout << "Password: " << password << endl;
  cout << "Account Name: " << accountname << endl;
  cout << "Description: " << desc << endl;
  cout << "Units: " << units << endl;

  if (bUseSSL == TRUE)
  {
    cout << "Using SSL to send the session." << endl ;
  }
  else
  {
    cout << "Not using SSL to send the session." << endl ;
  }

  if (bUseSynch == TRUE)
  {
    cout << "Using Synchronous mode to send the session." << endl ;
  }
  else
  {
    cout << "Not using Synchronous to send the session." << endl ;
  }
```

Initializing the Metering SDK

Before **MeterIt** can be called, **TestSimple** initializes the Metering SDK using the **Startup** function. If errors occur during initialization, **GetLastErrorObject** is called to return an error object. This error object is allocated memory that you should delete when processing the error completes. Otherwise, the error object continues to consume memory. If **MeterIt** returns an error, the error is displayed to the user.

Sample code follows:

```
if (!mMeter.Startup())
{
  cout << "Could not initialize the SDK." << endl;
  return mMeter.GetLastError();
}
```

Adding Metering Servers

For the Metering SDK to meter a session, it must be configured to communicate with a Metering server. Using the **AddServer** method on the **MTMeterHTTPConfig** object, you can add and specify settings for Metering servers.

If one of the Metering servers is unavailable when the Metering SDK tries to submit a session, the Metering SDK tries to send the session to the server with the next highest priority.

Sample code follows:

```
// if we are using SSL to send the session ... pass the
  // appropriate parameters correctly (port and secure flag)
  if (bUseSSL == TRUE)
  {
    mConfig.AddServer(0,                        // priority (highest)
      host,                                     // hostname
      MTMeterHTTPConfig::DEFAULT_HTTPS_PORT,    // port (from mtsdk.h)
      TRUE,                                     // secure (yes)
      username,                                 // username
      password);                                // password
  }
  // otherwise ... we're not using SSL to send the session ... pass the
  // appropriate parameters correctly (port and secure flag)
  else
  {
    mConfig.AddServer(0,                        // priority (highest)
      host,                                     // hostname
      MTMeterHTTPConfig::DEFAULT_HTTP_PORT,     // port (from mtsdk.h)
      FALSE,                                    // secure? (no)
      username,                                 // username
      password);                                // password
  }
```

The **AddServer** properties are:

- **Priority:** The priority of Metering servers for the Metering SDK to use in ascending order from 0. For example, a server with a priority of 0 is used before a server with a priority of 1. If one server fails, the Metering SDK uses the next server. If multiple servers have the same priority, the Metering SDK randomly chooses one.
- **Host name:** The computer name of the Metering server.

- **Port:** The port for the Metering server. For servers that do not use Secure Socket Layer (SSL), 80 is usually specified. For HTTPS servers using SSL, 443 is usually specified.
- **Secure:** Whether SSL is used to encrypt connections with the Metering server. To use SSL, specify True. Otherwise, specify False.
- **Username:** The username for authentication on the Metering server.
- **Password:** The password for authentication on the Metering server.

Failing Over to Another Metering Server

If one of the Metering servers is unavailable when the Metering SDK tries to send a session, the Metering SDK tries to send the session to the backup server specified by the user.

Sample code follows:

```
// Add the FAILOVER server if requested
// if we are using SSL to send the session ... pass the
// appropriate parameters correctly (port and secure flag)
if (strlen(backuphost))
{
     if (bUseSSL == TRUE)
     {
            mConfig.AddServer(0,                    // priority (highest)
                  backuphost,                       // hostname
                  MTMeterHTTPConfig::DEFAULT_HTTPS_PORT,  // port (from mtsdk.h)
                  TRUE,                            // secure (yes)
                  backupusername,                   // username
                  backuppassword);                  // password
     }
     // otherwise ... we're not using SSL to send the session ... pass the
     // appropriate parameters correctly (port and secure flag)
     else
     {
            mConfig.AddServer(0,                    // priority (highest)
                  backuphost,                       // hostname
                  MTMeterHTTPConfig::DEFAULT_HTTP_PORT,   // port (default
                                                          plaintext HTTP)
                  FALSE,                            // secure? (no)
                  backupusername,                  // username
                  backuppassword);                  // password
     }
}
```

Submitting a Session and Shutting Down the Metering SDK

To submit a session to the Metering server, the **MeterIt** function uses data entered by the user. If errors occur while performing this function, the **MTMeterError** object is returned. If no errors occur, NULL is returned.

The session is metered, and the Metering SDK is then shut down by calling the **Shutdown** function. The Sample 1 application waits for the user to press a key before displaying the metering results. When the Sample 1 application exits, the console window disappears.

Sample code for submitting a session follows:

```
// meter the session
MTMeterError * err = MeterIt(host, username, password, accountname,
desc, units);
if (!err)
  cout << "The session has been metered!" << endl;
else
{
  PrintError("The session has not been metered: ", err);
  delete err;
}

// close the sdk
mMeter.Shutdown();

// pause, in case this is run from a shortcut
cout << "Press any key to continue..." << endl;
(void) GetKey();
return;
```

Creating the Session

Using the **CreateSession** method, create an instance of an **MTMeterSession** object. The **Session** object contains property values that are used to describe a metered transaction. The service name is passed into the **CreateSession** method. The service name must match the service name on the Metering server.

Sample code follows:

```
MTMeterSession * session =
               mMeter.CreateSession("metratech.com/TestService");
```

Setting Synchronous Metering

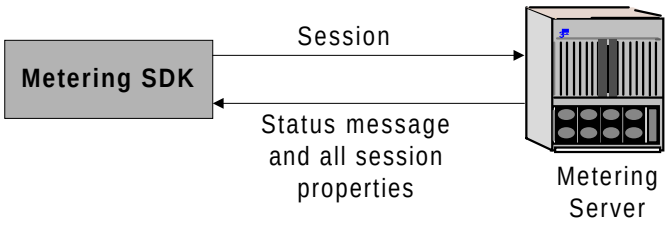
To enable or disable synchronous metering, use **SetResultRequestFlag**. This depends on information the user entered previously.

Sample code follows:

```
session->SetResultRequestFlag(bUseSynch);
```

Synchronous metering allows for bi-directional data exchange. The Metering SDK maintains the connection with the Metering server until the Metering SDK receives the results of the session. These results include all the properties in the session and the status message.

The following diagram shows how synchronous metering works:

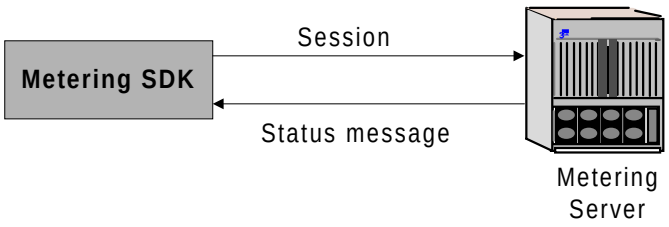


You might use synchronous metering to create an account and wait for the account ID before processing other transactions for the account. Synchronous metering is also useful for credit card pre-authorization.

Note: Use synchronous metering only where applicable. It uses more server resources than non-synchronous metering, because it keeps the connection open until the request is processed.

Asynchronous metering differs from synchronous metering in that the Metering SDK sends a session to the Metering server, which responds only with a status message indicating whether errors exist in the startup session. The Metering SDK closes the connection without waiting for the session to be processed.

The following diagram shows how non-synchronous metering works:



Adding Properties to the Session

Using the **InitProperty** method, add the property values to the session. **InitProperty** is an overloaded function that receives any supported data type.

For a set of supported data types, see the *InitProperty* sections in the *MetraTech Metering SDK Reference Manual*.

Descriptions of the properties follow:

- **AccountName:** The account name supplied by MetraTech. If you do not have an account name, specify **demo** (ASCII string).
- **Description:** A description of the data being metered (ASCII string).
- **Units:** The value being metered (float).
- **Time:** The time the transaction was metered in GMT (time_t value). This property must be specified as a timestamp in the service definition.

In the Sample 1 application, the **Time** property is set to the current time. The rest of the properties are added as entered by the user.

Sample code follows:

```
// set the session's time field to the current time.
time_t t = time(NULL);

// these property names have to match those on the server
  if (!session->InitProperty("AccountName", accountname)
  || !session->InitProperty("Description", desc)
  || !session->InitProperty("Units", units)
  || !session->InitProperty("Time", t))
{
{
  MTMeterError * err = session->GetLastErrorObject();
  delete session;

  // shutdown the metering library
  mMeter.Shutdown();
  return err;
}
```

Closing the Session

Before **Close** is called, **InitProperty** must add all required properties to the session. Use **Close** to close the session. This passes the value of each property to the Metering server. While closing the session, the Metering server validates the service name and list of properties.

If errors occur, **GetLastErrorObject** is called to retrieve an **MTMeterError** object that remains valid after the session object is deleted or modified. To free up memory allocated to it, delete the **MTMeterError** object after it is used.

Sample code follows:

```
// send the session to the server
if (!session->Close())
{
  MTMeterError * err = session->GetLastErrorObject();
  delete session;
```

Handling Errors

PrintError outputs any error messages generated by the Sample 1 application. To get the size of the buffer required to hold the error message, **GetErrorMessage** is called first with a size of zero. **GetErrorMessage** is then called again to populate the buffer. The buffer is deleted after the message is printed.

Sample code follows:

```
void SimpleMeter::PrintError(const char * prefix, const MTMeterError * err)
{
  cerr << prefix << ": ";
  if (err)
  {
    int size = 0;
    err->GetErrorMessage((char *) NULL, size);
    char * buf = new char[size];
    err->GetErrorMessage(buf, size);

    cerr << hex << err->GetErrorCode() << dec << ": " << buf << endl;
  }
  else
    cerr << "*UNKNOWN ERROR*" << endl;
}
```

Obtaining Session Results

Using the **GetSessionResults** method, obtain the session. Using the **GetProperty** method, obtain the monetary amount calculated in the session and the per unit cost. Display the calculated amount to the user.

Sample code follows:

```
if (session->GetResultRequestFlag())
  {
       MTMeterSession * ResultSession = session->GetSessionResults();
       if (ResultSession)
       {
```

```
                float Amount;
                float perunit;
                cout.setf (ios::showpoint);
                cout.setf (ios::fixed, ios::floatfield);
                cout.precision (2);
                if (ResultSession->GetProperty ("_Amount", Amount)  &&
                              ResultSession->GetProperty ("PerUnit", perunit))
                       cout << endl << "A per unit cost of " << perunit << " was
applied, resulting in a charge of " << Amount << endl << endl;
        }
  }
```

Shutting Down the Metering SDK

Use **Shutdown** to terminate the Metering SDK and free up the memory it uses.

Sample code follows:

```
  // shutdown the metering library
  mMeter.Shutdown();
  return err;
}
```

Deleting the Session

Before **MeterIt** returns, the session is deleted. This frees up all memory associated with the session and the session properties.

In the following sample code, the session was metered and no error occurred.

```
  delete session;
  // success! no error to return
  return NULL;
}
```

Main Function

The **main** function invokes the test.

Sample code follows:

```
int main(int argc, char * argv[])
{
       SimpleMeter meter;
       meter.TestSimple(argc, argv);
       return 0;
}
```

Sample 2: C++ Application for a Compound Session

The Sample 2 application meters a compound session with the TestService service as an atomic, child session. Components of the service are described in the following sections.

Header Files

The directive below includes the single header file required by users of the Metering SDK. **mtsdk.h** must be within your include path. For details about how to add this file to your include path, see step 1 in the *Building Services* chapter. The project file for the Sample 2 application already has the file in its include path.

The header file **sdk_msg.h** holds the definition for each of the error codes that are stored in **sdk_msg.dll**. This header file must be included to interpret error codes returned from the Metering SDK. The Sample 1 application does not include this file.

Sample code follows:

```
// Include metering objects
#include <mtsdk.h>
```

Application Class

The application class descriptions follow:

- The **CompoundMeter** class is the main application class.
- The **TestCompound** function is the entry point to the Sample 2 application. It is called by **main()**.
- **PrintError** prints errors if they occur.
- The **MTMeter** object is used to create metering objects, and an **MTMeterHTTPConfig** object is used to configure the **MTMeter** object.
- **MeterIt** is called by **TestCompound** to send the data read from the user to the Metering server using the SDK. **MeterIt** does the real work of the Sample 2 application.

Sample code follows:

```
class CompoundMeter
{
public:
  // construct the default configuration object, and use that to
  // construct the meter object
  CompoundMeter() : mMeter(mConfig)
  { }

  // entry point called from main
  void TestCompound(int argc, char * argv[]);

private:

  // meter the transaction based on the info entered by the user
  // returns NULL if there is no error
  MTMeterError * MeterIt(const char * host, const char * username,
    const char * password, const char * accountname, const char * parentdesc);

  // print an error if there is one
  void PrintError(const char * prefix, const MTMeterError * err);
```

```
  // wait for a key and return its value
  static int GetKey();

  // configuration object - used to initialize the Metering SDK
  // with HTTP transport
  MTMeterHTTPConfig mConfig;

  // the entry point to the Metering SDK - all Metering objects
  // are created from here.
  MTMeter mMeter;


};
```

Obtaining Data

The **TestCompound** function is called by **main()**. **TestCompound** prompts the user for the information necessary to submit a TestService session, including the parent session's account name and description. C++ streams are used for input and output.

To keep the Sample 2 application basic, only code specific to the Metering SDK is shown. More extensive stream error handling and data validation is normally required.

Sample code follows:

```
void CompoundMeter::TestCompound(int argc, char * argv[])
{
  cout << "MetraTech Compound metering example." << endl;

  // read the server host name
  cout << "Server host name: ";
  char host[256];
  cin.getline(host, sizeof(host));

  // read the server username
  cout << "Server username: ";
  char username[256];
  *username = 0;
  cin.getline(username, sizeof(username));

  // read the server password
  cout << "Server password: ";
  char password[256];
  *password = 0;
  cin.getline(password, sizeof(password));

  // read the user's account name
  cout << "Account Name: ";
  char accountname[256];
  cin.getline(accountname, sizeof(accountname));

  // read the description
  cout << "Parent description: ";
  char desc[256];
  cin.getline(desc, sizeof(desc));


  // summarize the metering information
  cout << endl << "Metering with the following information:" << endl;
  cout << "Host: " << host << endl;
  cout << "Username: " << username << endl;
  cout << "Password: " << password << endl;
```

```
        cout << "Account Name: " << accountname << endl;
        cout << "Description: " << desc << endl;
```

Initializing the Metering SDK

Before **MeterIt** can be called, **TestCompound** initializes the Metering SDK using the **Startup** function. If errors occur during initialization, **GetLastErrorObject** is called to return an error object. This error object is allocated memory that you should delete when processing the error completes. Otherwise, the error object continues to consume memory. If **MeterIt** returns an error, the error is displayed to the user.

Sample code follows:

```
// initialize the SDK
  if (!mMeter.Startup())
  {
    MTMeterError * err = mMeter.GetLastErrorObject();
    PrintError("Could not initialize the SDK: ", err);
    delete err;
    return;
  }

  int bUseSSL = FALSE;
```

Adding Metering Servers

For the Metering SDK to meter a session, it must be configured to communicate with a Metering server. Using the **AddServer** method on the **MTMeterHTTPConfig** object, you can add and specify settings for Metering servers.

If one of the Metering servers is unavailable when the Metering SDK tries to submit a session, the Metering SDK tries to send the session to the server with the next highest priority.

Sample code follows:

```
// if we are using SSL to send the session ... pass the
// appropriate parameters correctly (port and secure flag)
if (bUseSSL == TRUE)
{
mConfig.AddServer(0,                            // priority (highest)
host,                                           // hostname
MTMeterHTTPConfig::DEFAULT_HTTPS_PORT,          // port (from mtsdk.h)
TRUE,                                           // secure (yes)
username,                                       // username
password);                                      // password
}
// otherwise ... we're not using SSL to send the session ... pass the
// appropriate parameters correctly (port and secure flag)
else
{
mConfig.AddServer(0,                              // priority (highest)
host,                                             // hostname
MTMeterHTTPConfig::DEFAULT_HTTP_PORT,             // port (default plaintext HTTP)
FALSE,                                            // secure? (no)
username,                                         // username
password);                                        // password
}
```

The **AddServer** properties are:

- **Priority:** The priority of Metering servers for the Metering SDK to use in ascending order from 0. For example, a server with a priority of 0 is used before a server with a priority of 1. If one server fails, the Metering SDK uses the next server. If multiple servers have the same priority, the Metering SDK randomly chooses one.
- **Host name:** The computer name of the Metering server.
- **Port:** The port for the Metering server. For servers that do not use Secure Socket Layer (SSL), 80 is usually specified. For HTTPS servers using SSL, 443 is usually specified.
- **Secure:** Whether SSL is used to encrypt connections with the Metering server. To use SSL, specify True. Otherwise, specify False.
- **Username:** The username for authentication on the Metering server or an empty string.
- **Password:** The password for authentication on the Metering server or an empty string.

Submitting a Session and Shutting Down the Metering SDK

To submit a session to the Metering server, the **MeterIt** function uses data entered by the user. If errors occur while performing this function, the **MTMeterError** object is returned. If no errors occur, NULL is returned.

The session is metered, and the Metering SDK is shut down by calling the **Shutdown** function. The Sample 2 application waits for the user to press a key before displaying the metering results. When the Sample 2 application exits, the console window disappears.

Sample code for submitting a session follows:

```
// meter the session
MTMeterError * err = MeterIt(host, username, password, accountname, desc);
if (!err)
cout << "The session has been metered!" << endl;
else
{
PrintError("The session has not been metered: ", err);
delete err;
}


// close the sdk
mMeter.Shutdown();

// pause, in case this is run from a shortcut
cout << "Press any key to continue..." << endl;
(void) GetKey();
return;
}
```

Creating the Parent Session

Using the **CreateSession** method, create an instance of a **Session** object for the parent session. The **Session** object contains property values that are used to describe a metered transaction. The service name is passed into the **CreateSession** method. The service name must match the service name on the Metering server.

Sample code follows:

```
// service name is "metratech.com/testparent"
MTMeterSession * session = mMeter.CreateSession("metratech.com/testparent");
```

```
time_t t = time(NULL);
```

Adding Properties to the Parent Session

Using the **InitProperty** method, add the property values to the parent session. **InitProperty** is an overloaded function that receives any supported data type.

For a set of supported data types, see the *InitProperty* sections in the *MetraTech Metering SDK Reference Manual*.

Descriptions of the properties follow:

- **AccountName:** The account name supplied by MetraTech. If you do not have an account name, specify **demo** (ASCII string).
- **Description:** A description of the data being metered (ASCII string).
- **Time:** The time the transaction was metered in GMT (time_t value). This property must be specified as a timestamp in the service definition.

In the Sample 2 application, the **Time** property is set to the current time. The rest of the properties are added as entered by the user.

Sample code follows:

```
// Set the parent property names which have to match those on the server
      if (!session->InitProperty("AccountName", accountname)
            || !session->InitProperty("Description", parentdesc)
            || !session->InitProperty("Time", t))
      {
            MTMeterError * err = session->GetLastErrorObject();
            delete session;

            return err;
      }

  for (int i=0 ; i < sessions; i++)
  {
```

Obtaining the Child Session Description and Units

Prompt the user for a description of the child session and number of units to process.

Sample code follows:

```
// read the description
cout << "Child description: ";
char desc[256];
cin.getline(desc, sizeof(desc));

// read the units
cout << "Units (floating point number): ";
float units;
cin >> units;
cin.getline(temp, sizeof(temp));
```

Creating the Child Session

Using the **CreateChildSession** method, create an instance of a **Session** object for the child session. The **Session** object contains property values that are used to describe a metered

transaction. The service name is passed into the **CreateChildSession** method. The service name must match the service name on the Metering server.

When you create a child session, it is attached to the parent session.

Sample code for creating a session follows:

```
MTMeterSession * child = session->CreateChildSession("metratech.com/TestService");
```

Adding Properties to the Child Session

Using the **InitProperty** method, add the property values to the child session. **InitProperty** is an overloaded function that receives any supported data type.

For a set of supported data types, see the *InitProperty* sections in the *MetraTech Metering SDK Reference Manual*.

Descriptions of the properties follow:

- **AccountName:** The account name supplied by MetraTech. If you do not have an account name, specify **demo** (ASCII string).
- **Description:** A description of the data being metered (ASCII string).
- **Units:** The value being metered (float).
- **Time:** The time the transaction was metered in GMT (time_t value). This property must be specified as a timestamp in the service definition.

In the Sample 2 application, the **Time** property is set to the current time. The rest of the properties are added as entered by the user.

```
// set the session's time field to the current time.
t = time(NULL);

// these property names have to match those on the server
if (!child->InitProperty("AccountName", accountname)
 || !child->InitProperty("Description", desc)
 || !child->InitProperty("Units", units)
 || !child->InitProperty("Time", t))
{
 MTMeterError * err = child->GetLastErrorObject();
 delete child;

 return err;
 }
}
```

Closing the Parent Session

Before **Close** is called, **InitProperty** must add all required properties to the session. Use **Close** to close the session. This passes the value of each property to the Metering server. While closing the session, the Metering server validates the service name and list of properties. Closing the parent session closes all child sessions.

If errors occur, **GetLastErrorObject** is called to retrieve an **MTMeterError** object that remains valid after the session object is deleted or modified. To free up memory allocated to it, delete the **MTMeterError** object after it is used.

Sample code follows:

```
// send the session to the server
if (!session->Close())
{
  MTMeterError * err = session->GetLastErrorObject();
return err;
}
  delete session;
```

Obtaining Session Results

Using **GetSessionResults**, obtain the session results. You can only get the results back using synchronous metering.

Sample code follows:

```
if (session->GetResultRequestFlag())
  {
       MTMeterSession * ResultSession = session->GetSessionResults();
       if (ResultSession)
       {
              float Amount;
              float perunit;
              cout.setf (ios::showpoint);
              cout.setf (ios::fixed, ios::floatfield);
              cout.precision (2);
              if (ResultSession->GetProperty ("_Amount", Amount)  &&
                           ResultSession->GetProperty ("PerUnit", perunit))
                     cout << endl << "A per unit cost of " << perunit << " was
applied, resulting in a charge of " << Amount << endl << endl;
       }
  }
```

Deleting the Session

Delete the session.

Sample code follows:

```
// sessions created with CreateSession must be deleted.
  delete session;

  // success! no error to return
  return NULL;
}
```

Handling Errors

PrintError outputs any error messages generated by the Sample 2 application. To get the size of the buffer required to hold the error message, **GetErrorMessage** is called first with a size of zero. **GetErrorMessage** is then called again to populate the buffer. The buffer is deleted after the message is printed.

Sample code follows:

```
void CompoundMeter::PrintError(const char * prefix, const MTMeterError * err)
{
  cerr << prefix << ": ";
  if (err)
  {
    int size = 0;
    err->GetErrorMessage((char *) NULL, size);
    char * buf = new char[size];
    err->GetErrorMessage(buf, size);

    cerr << hex << err->GetErrorCode() << dec << ": " << buf << endl;
  }
  else
    cerr << "*UNKNOWN ERROR*" << endl;
}
```

Main Function

Main constructs a single CompoundMeter object and calls its main entry point, **TestCompound**.

Sample code follows:

```
int main(int argc, char * argv[])
{
  CompoundMeter meter;
  meter.TestCompound(argc, argv);
  return 0;
}
```

Chapter 4 — Creating COM Applications For the Metering SDK

Overview

This chapter describes the build environment and provides sample COM applications for the SDK. You can use these sample applications as a starting point for your own integration with the SDK.

The sections in this chapter are:

- Build Environment
- Sample 1: COM Application for an Atomic Session
- Sample 2: COM Application for a Compound Session
- Sample 3: Synchronous Metering COM Application
- Sample 4: Secure Metering COM Application
- Sample 5: Local Mode COM Application
- Sample 6: Failover COM Application

Build Environment

The current version of the Metering SDK for Windows NT was built with the Microsoft Visual C++ compiler, version 6.0. The libraries and DLLs included were built with Microsoft Visual C++, version 6.0.

The sample applications were created in Visual Basic.

Helpful Hint:
The following sample applications have the correct build environment settings. If you want to change the settings, see *Step 4: Implementation* in the *Building Services* chapter.

Sample 1: COM Application for an Atomic Session

The Sample 1 application meters an atomic session for the metratech.com/TestService service. Components of the service are described in the following sections.

Initializing the Metering SDK

Using the **Startup** function, **MeterSimple** initializes the Metering SDK.

Sample code follows:

```
Private Sub MeterSimple()

Dim objSession As Session
Dim objMeter As Meter
```

```
' Create the meter object
Set objMeter = New Meter

' start up the SDK
objMeter.Startup
```

Setting the Timeout

Using **HTTPTimeout**, you can set the timeout interval for all Metering servers using the application. This is an optional function. The timeout interval is the number of milliseconds during which the Metering SDK tries to send a session to a Metering server before retrying or failing over to the server with the next highest priority.

For a description of the priority to which servers are assigned, see *Adding Metering Servers* on page 45. For a description of how **HTTPTimeout** works with **HTTPRetries**, see *Setting the Number of Retries* on page 40.

Sample code follows:

```
objMeter.HTTPTimeout = 30
```

Setting the Number of Retries

Using **HTTPRetries**, you can set the number of times the Metering SDK tries to send a session to a Metering server before failing over to the server with the next highest priority. This is an optional function.

For example, **HTTPRetries** is set to 2 and **HTTPTimeout** is set to 30. The Metering SDK tries to send the session to the first Metering server for 30 milliseconds. If it fails, the Metering SDK tries for 30 milliseconds to send the session to the server again. If it fails again, the Metering SDK tries to send the session to the server with the next highest priority.

For a description of how priorities work, see *Adding Metering Servers* on page 45. For a description of how **HTTPRetries** works with **HTTPTimeout**, see *Setting the Timeout* on page 44.

Sample code follows:

```
objMeter.HTTPRetries = 2
```

Adding Metering Servers

For the Metering SDK to meter a session, it must be configured to communicate with a Metering server. Using the **AddServer** method on the **Meter** object, you can add and specify settings for Metering servers.

If one of the Metering servers is unavailable when the Metering SDK tries to submit a session, the Metering SDK tries to send the session to the server with the next highest priority.

Sample code for a Metering server without Secure Socket Layer (SSL) follows. For sample code for a server with SSL, see *Sample 4: Secure Metering COM Application* on page 53.

```
' priority, servername, port, secure, username, password
objMeter.AddServer 0, txtServer.Text, DEFAULT_HTTP_PORT, False, "", ""
```

The **AddServer** properties are:

- **Priority:** The priority of Metering servers for the Metering SDK to use in ascending order from 0. For example, the server with the priority of 0 is used before a server with a priority of 1. If one server fails, the Metering SDK uses the next server. If multiple servers have the same priority, the Metering SDK randomly chooses one.
- **Server name:** The computer name of the Metering server.
- **Port:** The port for the Metering server. For servers that do not use Secure Socket Layer (SSL), 80 is usually specified. For HTTPS servers using SSL, 443 is usually specified.
- **Secure:** Whether SSL is used to encrypt connections with the Metering server. To use SSL, specify True. Otherwise, specify False. For an example of an application for secure metering, see *Sample 4: Secure Metering COM Application* on page 53.
- **Username:** The username for authentication on the Metering server.
- **Password:** The password for authentication on the Metering server.

Creating the Session

Using the **CreateSession** method, create an instance of a **Session** object. The **Session** object contains property values that are used to describe a metered transaction. The service name is passed into the **CreateSession** method. The service name must match the service name on the Metering server.

Sample code for creating a session follows:

```
' create a test session
Set objSession = objMeter.CreateSession("metratech.com/TestService")
```

Adding Properties to the Session

Using the **InitProperty** method, add the property values to the session. **InitProperty** is an overloaded function that receives any supported data type.

For a set of supported data types, see the *InitProperty* sections in the *MetraTech Metering SDK Reference Manual*.

Descriptions of the properties follow:

- **AccountName:** The account name supplied by MetraTech. If you do not have an account name, specify **demo** (ASCII string).
- **Description:** A description of the data being metered (ASCII string).
- **Units:** The value being metered (float).
- **Time:** The time the transaction was metered in GMT (time_t value). This property must be specified as a timestamp in the service definition.

In the Sample 1 application, the **Time** property is set to the current time. The rest of the properties are added as entered by the user.

Sample code follows:

```
' set the session properties
objSession.InitProperty "AccountName", txtAccount.Text
objSession.InitProperty "Description", txtDescription.Text
objSession.InitProperty "Units", CDbl(txtUnits.Text)
objSession.InitProperty "Time", Now
```

Closing the Session

Before **Close** is called, **InitProperty** must add all required properties to the session. Use **Close** to close the session. This passes the value of each property to the Metering server. While closing the session, the Metering server validates the service name and list of properties.

Sample code follows:

```
' close the session
objSession.Close
```

Shutting Down the Metering SDK

Use **Shutdown** to terminate the Metering SDK and free up the memory it uses.

Sample code follows:

```
' shutdown the SDK
objMeter.Shutdown

End Sub
```

Sample 2: COM Application for a Compound Session

The Sample 2 application meters a compound session with the TestService service as an atomic, child session.

In the following sample code, functionality related to metering compound sessions is shown in bold. For a description of this code, see the following sections. For a description of the rest of the code, see *Sample 1: COM Application for an Atomic Session* on page 43.

```
Private Sub MeterCompound()

Dim objParent As Session
Dim objChild As Session
Dim objMeter As Meter

' Create the meter object
Set objMeter = New Meter

' start up the SDK
objMeter.Startup

' Set the timeout and number of server retries
objMeter.HTTPTimeout = 30
objMeter.HTTPRetries = 1

' priority, servername, port, secure, username, password
objMeter.AddServer 0, txtServer.Text, DEFAULT_HTTP_PORT, False, "", ""

' create the parent session
Set objParent = objMeter.CreateSession("metratech.com/testparent")

' init the properties in the parent

objParent.InitProperty "AccountName", txtAccount.Text
objParent.InitProperty "Description", txtDescription.Text
objParent.InitProperty "Time", Now

' create some test children
For i = 1 To 5
' create a child session
Set objChild = objParent.CreateChildSession("metratech.com/TestService")

' set the session properties
objChild.InitProperty "AccountName", txtAccount.Text
objChild.InitProperty "Description", txtDescription.Text
objChild.InitProperty "Units", CDbl(txtUnits.Text)
objChild.InitProperty "Time", Now

Next i

' close the parent which will close all the children
objParent.Close

' shutdown the SDK
objMeter.Shutdown

End Sub
```

Creating the Parent Session

Using the **CreateSession** method, create an instance of a **Session** object for the parent session. The **Session** object contains property values that are used to describe a metered transaction. The service name is passed into the **CreateSession** method. The service name must match the service name on the Metering server.

Sample code follows:

```
' create the parent session
Set objParent = objMeter.CreateSession("metratech.com/testparent")
```

Adding Properties to the Parent Session

Using the **InitProperty** method, add the property values to the parent session. **InitProperty** is an overloaded function that receives any supported data type.

For a set of supported data types, see the *InitProperty* sections in the *MetraTech Metering SDK Reference Manual*.

Descriptions of the properties follow:

- **AccountName:** The account name supplied by MetraTech. If you do not have an account name, specify **demo** (ASCII string).
- **Description:** A description of the data being metered (ASCII string).
- **Time:** The time the transaction was metered in GMT (time_t value). This property must be specified as a timestamp in the service definition.

In the Sample 2 application, the **Time** property is set to the current time. The rest of the properties are added as entered by the user.

Sample code follows:

```
' init the properties in the parent

objParent.InitProperty "AccountName", txtAccount.Text
objParent.InitProperty "Description", txtDescription.Text
objParent.InitProperty "Time", Now
```

Creating the Child Sessions

Using the **CreateChildSession** method, create an instance of a **Session** object for the child session. The **Session** object contains property values that are used to describe a metered transaction. The service name is passed into the **CreateChildSession** method. The service name must match the service name on the Metering server.

When you create a child session, it is attached to the parent session.

Sample code for creating a session follows:

```
' create some test children
For i = 1 To 5
' create a child session
Set objChild = objParent.CreateChildSession("metratech.com/TestService")
```

Adding Properties to the Child Session

Using the **InitProperty** method, add the property values to the session. **InitProperty** is an overloaded function that receives any supported data type.

For a set of supported data types, see the *InitProperty* sections in the *MetraTech Metering SDK Reference Manual*.

Descriptions of the properties follow:

- **AccountName:** The account name supplied by MetraTech. If you do not have an account name, specify **demo** (ASCII string).
- **Description:** A description of the data being metered (ASCII string).
- **Units:** The value being metered (float).
- **Time:** The time the transaction was metered in GMT (time_t value). This property must be specified as a timestamp in the service definition.

In the Sample 1 application, the **Time** property is set to the current time. The rest of the properties are added as entered by the user.

Sample code follows:

```
' set the session properties
objChild.InitProperty "AccountName", txtAccount.Text
objChild.InitProperty "Description", txtDescription.Text
objChild.InitProperty "Units", CDbl(txtUnits.Text)
objChild.InitProperty "Time", Now

Next i
```

Closing the Parent Session and Shutting Down the SDK

Before **Close** is called, **InitProperty** must add all required properties to the session. Use **Close** to close the session. This passes the value of each property to the Metering server. While closing the session, the Metering server validates the service name and list of properties. Closing the parent session closes all child session.

To shut down the Metering SDK, call the **Shutdown** function.

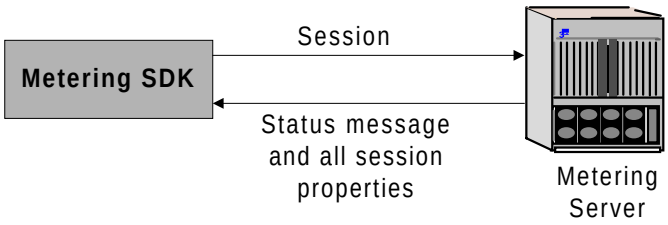
Sample code follows:

```
' close the parent which will close all the children
    objParent.Close

    ' shutdown the SDK
    objMeter.Shutdown

End Sub
```

Sample 3: Synchronous Metering Application

Synchronous metering allows for bi-directional data exchange. The Metering SDK maintains the connection with the Metering server until the Metering SDK receives the results of the session. These results include all the properties in the session and the status message.

The following diagram shows how synchronous metering works:

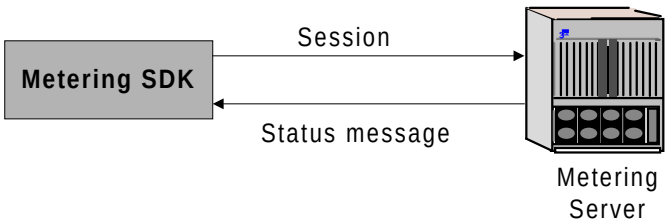


You might use synchronous metering to create an account and wait for the account ID before processing other transactions for the account. Synchronous metering is also useful for credit card pre-authorization.

Note: Use synchronous metering only where applicable. It uses more resources than non-synchronous metering, because it keeps the connection open until the request is processed.

Non-synchronous metering differs from synchronous metering in that the Metering SDK sends a session to the Metering server, which responds with a status message indicating whether errors exist. The Metering SDK closes the connection without waiting for the session to be processed.

The following diagram shows how non-synchronous metering works:



In the following sample code, functionality related to synchronous metering is shown in bold. For a description of this code, see the following sections. For a description of the rest of the code, see *Sample 1: COM Application for an Atomic Session* on page 43.

Sample code follows:

```
Private Sub MeterSynchronous()

Dim objSession As Session
Dim objResultSession As Session
Dim objMeter As Meter

' Create the meter object
Set objMeter = New Meter

' start up the SDK
objMeter.Startup

' Set the timeout and number of server retries
objMeter.HTTPTimeout = 30
objMeter.HTTPRetries = 1

' priority, servername, port, secure, username, password
objMeter.AddServer 0, txtServer.Text, DEFAULT_HTTP_PORT, False, "", ""

' create a test session
Set objSession = objMeter.CreateSession("metratech.com/TestService")

' Request a response session
objSession.RequestResponse = True

' set the session properties
objSession.InitProperty "AccountName", txtAccount.Text
objSession.InitProperty "Description", txtDescription.Text
objSession.InitProperty "Units", CDbl(txtUnits.Text)
objSession.InitProperty "Time", Now

' close the session
objSession.Close

' Get the result session
Set objResultSession = objSession.ResultSession

' Get the property added by the test stage
amount = objResultSession.GetProperty("_Amount", MTC_DT_DOUBLE)

' Display the property
txtAmount.Text = amount

' shutdown the SDK
objMeter.Shutdown

End Sub
```

Requesting Session Results

Using the **RequestResponse** method, request and wait for the results of the session processing.

Sample code follows:

```
' Request a response session
objSession.RequestResponse = True
```

Obtaining Session Results

Using the **ResultSession** method, obtain the session.

Sample code follows:

```
' Get the result session
Set objResultSession = objSession.ResultSession
```

Obtaining the Result Property

Using the **GetProperty** method, obtain the monetary amount calculated in the session.

```
' Get the property added by the test stage
amount = objResultSession.GetProperty("_Amount", MTC_DT_DOUBLE)
```

Displaying the Session Results to the User

Using the **txtAmount** method, display the monetary amount calculated in the session to the user.

```
' Display the property
txtAmount.Text = amount
```

Sample 4: Secure Metering COM Application

You can use SSL to encrypt connections. This provides for secure communications between the Metering SDK and Metering servers.

In the following sample code, functionality related to secure metering is shown in bold. For a description of this code, see *Adding Metering Servers* on page 45. For a description of the rest of the code, see *Sample 1: COM Application for an Atomic Session* on page 43.

```
Private Sub MeterSecure()

Dim objSession As Session
Dim objMeter As Meter

' Create the meter object
Set objMeter = New Meter

' start up the SDK
objMeter.Startup

' Set the timeout and number of server retries
objMeter.HTTPTimeout = 30
objMeter.HTTPRetries = 1

' priority, servername, port, secure, username, password
objMeter.AddServer 0, txtServer.Text, DEFAULT_HTTPS_PORT, True, txtUser.Text,
txtPassword.Text

' create a test session
Set objSession = objMeter.CreateSession("metratech.com/TestService")

' set the session properties
objSession.InitProperty "AccountName", txtAccount.Text
objSession.InitProperty "Description", txtDescription.Text
objSession.InitProperty "Units", CDbl(txtUnits.Text)
objSession.InitProperty "Time", Now

' close the session
objSession.Close

' shutdown the SDK
objMeter.Shutdown

End Sub
```

Sample 5: Local Mode COM Application

If there is a problem with the network, you can process sessions on the local computer running the Metering SDK instead of sending the sessions over the network to a remote Metering server. Data goes from the client application to the SDK computer and does not continue to the Metering server. Instead, it goes to a file.

Note: You cannot perform synchronous metering in local mode.

In the following sample code, functionality related to local mode is shown in bold. For a description of this code, see the following sections. For a description of the rest of the code, see *Sample 1: COM Application for an Atomic Session* on page 43.

```
Private Sub MeterLocalMode()

Dim objSession As Session
Dim objMeter As Meter

' Create the meter object
Set objMeter = New Meter

' start up the SDK
objMeter.Startup

' Set the timeout and number of server retries
objMeter.HTTPTimeout = 30
objMeter.HTTPRetries = 1

' priority, servername, port, secure, username, password
objMeter.AddServer 0, txtServer.Text, DEFAULT_HTTP_PORT, False, "", ""

' turn on local mode by setting a file
objMeter.LocalModePath = "C:\temp\comlocalmode.dat"

' Meter 10 test sessions locally
For I = 1 To 10
' create a test session
Set objSession = objMeter.CreateSession("metratech.com/TestService")

' set the session properties
objSession.InitProperty "AccountName", txtAccount.Text
objSession.InitProperty "Description", txtDescription.Text
objSession.InitProperty "Units", CDbl(txtUnits.Text)
objSession.InitProperty "Time", Now

' close the session
objSession.Close
Next I

' turn off local mode
objMeter.LocalModePath = ""

' set a journal for replay
' this is used to keep track of the status
objMeter.MeterJournal = "c:\temp\meterstore.dat"

' play it back
objMeter.MeterFile "C:\temp\comlocalmode.dat"
```

```
' shutdown the SDK
objMeter.Shutdown


End Sub
```

Turning On Local Mode

To turn on local mode, use **LocalModePath**. This sets a path for the file to which the sessions are written by the Metering SDK.

Sample code follows:

```
' turn on local mode by setting a file
objMeter.LocalModePath = "C:\temp\comlocalmode.dat"
```

Metering Test Sessions

Meter ten test sessions in local mode.

Sample code follows:

```
' Meter 10 test sessions locally
For I = 1 To 10
```

Turning Off Local Mode

To turn off the local mode path, use **LocalModePath** to set the path to null.

Sample code follows:

```
' turn off local mode
objMeter.LocalModePath = ""
```

Setting the Journal

To record the status of locally recorded sessions, use **MeterJournal**. The journal contains the Session ID and status of the session processing. The statuses indicate whether the session was not sent, sent but not confirmed as received, or sent and confirmed as received. Sessions not sent are re-submitted.

Sample code follows:

```
' set a journal for replay
' this is used to keep track of the status
objMeter.MeterJournal = "c:\temp\meterstore.dat"
```

Metering the File to the Original Server

When the network problem is resolved, send the batch file of sessions to the original Metering server. To do this, use **MeterFile**.

Sample code follows:

```
' play it back
objMeter.MeterFile "C:\temp\comlocalmode.dat"
```

Sample 6: Failover COM Application

If one of the Metering servers is unavailable when the Metering SDK tries to send a session, the Metering SDK tries to send the session to another server. Before trying the next server, the Metering SDK tries to send the session a specific number of times and for a specific amount of time.

In the following sample code, functionality related to failover is shown in bold. For a description of this code, see *Setting the Timeout Interval* on page 44, *Setting the Number of Retries* on page 44, and *Adding Metering Servers* on page 45. For a description of all the functionality in the following sample code, see *Sample 1: COM Application for an Atomic Session* on page 45.

```
Private Sub MeterFailover()

Dim objSession As Session
Dim objMeter As Meter

' Create the meter object
Set objMeter = New Meter

' start up the SDK
objMeter.Startup

' Set the timeout and number of server retries
objMeter.HTTPTimeout = 30
objMeter.HTTPRetries = 2

' add a primary server
' priority, servername, port, secure, username, password
objMeter.AddServer 0, txtServer.Text, DEFAULT_HTTP_PORT, False, "", ""

' add a backup server
' priority, servername, port, secure, username, password
' priority, servername, port, secure, username, password
objMeter.AddServer 0, txtBackupServer.Text, DEFAULT_HTTP_PORT, False, "", ""

' create a test session
Set objSession = objMeter.CreateSession("metratech.com/TestService")

' set the session properties
objSession.InitProperty "AccountName", txtAccount.Text
objSession.InitProperty "Description", txtDescription.Text
objSession.InitProperty "Units", CDbl(txtUnits.Text)
objSession.InitProperty "Time", Now

' close the session
objSession.Close

' shutdown the SDK
objMeter.Shutdown
```

Chapter 5 — Building Services

Overview

This chapter describes how to create services and builds on topics presented in the MetraTech Concepts section of this document. While this section provides a good overview of many of the variables to consider when creating services, it does not cover all the business models that NSPs are looking to deploy. MetraTech partners with both ISVs and NSPs to determine the best way to fit services into targeted business models.

The sections in this chapter are:

- Step 1: Planning
- Step 2: Filling Out Service Worksheets
- Step 3: Contacting MetraTech
- Step 4: Implementation
- Step 5: Testing the Code

The Process

The process of building your own service is straightforward and can be done in as little as one day for simple services. The 5-step process is listed below. A detailed explanation of each step is provided in this section:

1. **Planning**: Determine what services and service properties you are going to define.
2. **Filling Out Service Worksheets**: Fill out the included Excel-based service detail worksheets.
3. **Contacting MetraTech**: Email the worksheets to sdk_support@metratech.com.
4. **Implementation**
5. **Testing**: Test that the integration works, including the web presentation of the transactions.

Step 1: Planning

The first step in planning a new service is to identify the application-level tasks that your server provides for a client. Application-level tasks are best thought of in terms of data flows in which specific events generate data that is suitable for metering, rating, presentation and billing. While most servers typically write all of the data required for billing to log files, the data is usually intermixed with a significant amount of unrelated information and may be scattered across multiple log files. Thus, the issue is not that the data does not exist, but rather that it exists at the system-level instead of at the application-level.

Thinking at the application-level is important for two reasons: (1) end-users typically understand and are willing to pay for application-level charges, and (2) it is typically very difficult for any third-party system to distill application-level transactions from system-level data. When planning your services, you should always consider the following issues, described in the next four sections of this chapter:

- To Meter or Not to Meter
- Atomic or Compound
- Data Volume
- Customer self-help
- Customer Support

To Meter or Not to Meter

Many ISVs wonder if they should meter certain services because they can't imagine an NSP actually billing customers for it. In many situations this may be the case, but you should keep in mind that while an NSP may not conceptually want to bill for a given service, they may rely on billing plans to prevent excessive use by charging abusers once they cross a normal use threshold.

For example, imagine a universal messaging service that has fax, voice, video, pager and email capabilities. Most people would assume that everything but email should be metered because email is "free." While this is typically true, spamming is still a common practice and email files containing attachments are increasing in size. One can easily imagine an NSP creating a plan that allows users to send 5,000 emails and 100MB of traffic per month for "free" as part of a messaging service, but then applying usage-based charges for anything above those usage levels. If the NSP does not want to meter the service, they can always disable it.

Data volume is a common concern with metering high-volume services. While not discussed in this document, MetraTech's Pipeline Server has the ability to summarize this type of data into a single counter on a per account basis to reduce the volume of data stored in the database.

Atomic or Compound

In many cases application-level services encompass multiple system events. When this occurs you should determine which, if any, system events can be collapsed into a single MSIX session. Collapsing multiple system events into a session is a more efficient, but often less flexible way of describing service usage. The following three scenarios should provide some insight into this issue:

- **Scenario 1:** Imagine a fax system in which a gatekeeper logs the caller's ID, the assigned QoS and the IP address of the IP-PSTN (public switched telephone network) gateway. The gateway, on the other hand, logs the caller's ID, the start time and the end time.

  Clearly there are two system-level events on two separate platforms, but there is only one application-level event. If we assume that the NSP will want a rating algorithm that takes into account the start time, the elapsed time and the QoS level, the two system events will need to be joined to create an application event. This could be done in one of two ways: (1) create a compound session that contains a "gate keeper" session and a "gateway" session or (2) collapse the two system events into a single session. In this scenario, it is recommended that the sessions be collapsed because the added complexity of a compound session adds no value.

- **Scenario 2:** Imagine that this same system has the ability to retry sending faxes if there is a transmission problem or the destination fax machine is busy or out of paper.

  This scenario has all of the complexities of the first scenario, but now the single application event that we created at the end of scenario one is capable of generating *N* other application events where *N* is the number of retries. Furthermore, each of these retries might incur PSTN charges. While these attempts could be collapsed into a single session that had a *NumberRetrySeconds* property, it is recommended that each retry is given its own session for

the following reasons (1) each retry could generate a charge that the user would like to understand (2) the user might want to know why the fax took so long to go through, and (3) the NSP might want to reconcile their records with their telco provider. It should be noted that this retry session might be defined to be a subset of the data contained in the initial attempt to reduce data volume.

- **Scenario 3**: Imagine that this same system has the ability to send fax broadcasts to an unlimited number of recipients.

  This scenario obviously lends itself to a compound session that contains the type of sessions that we created in the second scenario. The question is what properties, if any, can be added to the compound session to provide summary-level data. It is suggested that properties like *NumberSuccessful*, *NumberFailed*, etc. be added to provide high-performance, top-level views of the compound session.

Data Volume

The metering server is scaleable and has the ability to filter and summarize data before persistently storing it, but it is not a replacement for a generic log file. Careful consideration should be given to the services and properties that you choose to meter. Data volume issues usually arise around services that persist over a long period of time.

A good example of this is a webserver. Many web-hosting providers are beginning to charge for the bandwidth used during the course of a month. Generating an MSIX session for every server response would degrade the performance of the web server as well as place a high burden on the metering server. A better implementation would be to maintain an in-memory bandwidth counter and generate sessions on a configurable, time-based interval. Some NSPs might want a daily interval to enable end-users to see their bill on a daily basis and enable them to run daily trending reports whereas another NSP might want an hourly interval to enable time-of-day rating. The important point is to find a reasonable balance between data volume and data resolution and provide the NSP with flexible configuration options.

Customer Self-help

One of the primary value propositions of online billing systems is to reduce the number of inbound customer service calls. In many cases services can be defined to include information that is useful to the end-user that is not used in the rating process or in legacy paper billing systems. The information was not typically included in paper-based bills because there was not enough room to fit the data into a single line item. The Presentation Server has a sophisticated data navigation interface that provides summary-level information which, when clicked on, presents all of the detail associated with a session. This type of presentation can often enable users to answer their own questions without the need to call a customer service representative. Before adding a property to a service, however, you should weigh the value of the data with respect to the increase in data volume.

For example, adding a subject, a recipient name and a baud rate property to the above fax scenarios would provide the end-user with data that might prevent a customer service call. The subject and recipient name would help the user identify the fax better than the recipient's fax number and the baud rate would explain the wide variations in cost for broadcast faxes sent to the same area code. The baud rate property is important because most fax services rate on a per minute basis and the duration of the fax is highly dependent on the baud rate of the recipient's machine (something the end-user and the NSP have no control over). Fax machines vary from 2,400 to 14,000 baud which results in a rating variation of about 600%.

Customer Support

Not all end-users have access to the Presentation Server and therefore any data that is available via the Presentation Server is helpful to a customer support representative (CSR) as well. In addition to these properties, however, you might want to add additional properties that the CSR has access to but are not visible to the end user (the Presentation Server enables NSPs to determine which fields an end-user has access to). These fields can be very important when a CSR needs to solve a problem or evaluate providing a credit to an unhappy customer. This type of data will become increasingly important as customers look to NSPs to provide proof that they are maintaining their service-level agreements.

For example, imagine a video conferencing service where the user can choose from several levels of quality. While the video conferencing server could easy meter the packet loss rate, an NSP may not choose to expose this information to the end-user via the Presentation Server. However, if a customer calls and complains about poor image quality, the CSR will have the ability to make a decision based on facts rather than the user's impression.

Step 2: Filling Out Service Worksheets

Worksheets and samples have been included with the SDK installation to facilitate the creation of your own services. Please refer to the *Planning Aids* appendix on page 67 for further details. Before filling out these sheets you need to understand the following service details:

Service Names

Service names are designed to be globally unique. They are composed of two or more elements separated by single slashes ("/"). This first element is the Internet domain name of the organization that is defining the service and subsequent elements are alphanumeric strings, without whitespace and optionally with dashes or underscores. For example, if MetraTech were to define fax, voice and video services, their service names would be metratech.com/fax, metratech.com/voice and metratech.com/video respectively. Vendors are responsible for maintaining their own namespace of services underneath their domain name element.

The general representation for service names is as follows:

vendor_domain_name/*service*/*service*…

Sub-services

Compound services by definition contain other services. When listing sub-services, use their fully qualified service names.

Properties

Services are composed of one or more properties. As described earlier, clients communicate metering information to servers via properties. Clients provide name/value pairs in the information sent to the server. In addition to name and value, properties are represented on the server using several other elements. Each property is composed of the following elements:

Name: Property names are strings of alphabetic and/or numeric characters, lacking white space and starting with an alphabetic character. While names are case insensitive, MetraTech recommends a mixed case implementation.

Type: Property types are restricted to those listed below.

Type	Description
STRING	Arbitrary-length character string
UNISTRING	Arbitrary-length Unicode character string
INT32	String representation of 4-byte signed integer
FLOAT	String representation of 4-byte IEEE floating point number
DOUBLE	String representation of 8-byte IEEE floating point number
TIMESTAMP	ISO8601 date string, as defined in appendix

Value: The value of the property.

Length: The maximum length of a string or unistring value. Valid ranges are from 1 to 255.

UOM: Units of measure (UOM) describe numeric properties (non-numeric properties do not have UOMs). A UOM determines the exact interpretation of a numeric quantity by defining measurement units such as "seconds", "bytes", "milliseconds", etc. UOMs are defined by the MSIX client and are limited to 18 characters.

Required: Indicates if the property is required. If a property is required and not passed by the SDK client, an error is generated. If the property is not required and not specified, the default value is used.

Default: The default value of property. This value is used when the property is not required and not specified.

Description: The property description is a free-form field used for on-line help. It is limited to 80 characters.

Step 3: Contacting MetraTech

When you have completed the Service Worksheets, email them to sdk_support@metratech.com. A MetraTech representative will review the services, create the appropriate database objects and generate customized Presentation Server web pages and notify you when the changes are completed. In a future release, you will be able to do this directly through the SDK.

Step 4: Testing the Code

The next step to get our service up and running is to compile and test the new service:

1. Compile the code.
2. To test it, run your application in the debugger.

Warning! Make sure you have registered your service with MetraTech. Otherwise, the call to **Close** will fail. Please consult the *Partner's Data Sheet* for details on who to contact to register your new service.

3. Having registered the service with MetraTech's reference server, you should be ready for your first end-to-end test.
4. Execute your code, noting the values of the properties being set.
5. Using your Internet browser, browse to the reference server and log onto the Presentation Server. Unless otherwise specified, your transaction will appear under the *Miscellaneous Grouping*. Browse to your transaction. Verify that the properties displayed match the properties that you set.

Enhancing the code

We have used the Simple example as a starting point to get our service instrumented. We did this to ensure that the simple case works before complicating the service too much. It is likely that you will want to modify the structure of the code to integrate more closely with your software.

After all the properties for a service have been added correctly, C**lose** can be called. **Close** stores a copy of the session and its properties on the server but still allows modification of any of its properties before sending the session back to the server. The intended use for **Close** is for events that span a long period of time. In this scenario, there is an increased risk that the network entity providing the service might fail before metering usage information. Without the ability to checkpoint session information with the **Close** method, important usage information might be lost. If **Close** is called, the **GetProperty** and **SetProperty** methods may be called to modify property values. If any property values are changed, **Close** can be called. **InitProperty** cannot be called once a session has been saved.

Normally **InitProperty** is called to add the appropriate properties before **Close** is called. **Close** sends the session and its properties to the server and restricts any further changes. Once a session is marked closed, the server is allowed to begin further processing on it. It is not possible to modify a session's properties once it has been closed.

Some services are hierarchical in nature with a "parent" service being composed of multiple component "child" services. In this case, a parent session is opened using **CreateSession** and each child session is opened using **CreateChildSession** on the parent session object. The session returned from **CreateChildSession** is linked to its parent session. Deleting a parent session will delete all of its children. You can create any number of children with **CreateChildSession**. Child sessions can in turn have other children.

If Save is called on a parent session, any children that need to be saved are saved as well. **CreateChildSession** can still be called from this parent session to create new children. If Close is called on a parent session, all of its children will be closed. No more child sessions can be created from a closed parent. Calling Close on a child session will save all of its parents, still allowing new children to be created.

Chapter 6 — Technical Support

Overview

This chapter gives you information about obtaining technical support from MetraTech:

- Committed to Your Success
- Before You Contact Us
- Online Support
  - Email
  - Release Notes
- Developer Support Hotline

Committed to Your Success

MetraTech is committed to supporting its partners. We offer high quality on-line and telephone support to provide you with the information that you need, when you need it.

Before You Contact Us

In many situations, you can get immediate answers to your technical questions without contacting our Developer Support Center. The Troubleshooting section of this manual and MetraTech's On-line Support Options can help isolate, resolve or provide work arounds for problems you may encounter in developing your application.

Online Support

On-line support is provided free of charge 24 hours a day via MetraTech's password-protected partner site (**partner.metratech.com**). There are currently the following channels for online support:

Email

Questions can be directed to MetraTech Customer Services via email at **sdk_support@metratech.com**. We'll provide responsive acknowledgement and best-effort turnaround. If you require an immediate response, please use the Developer Support Hotline.

Release Notes

Release notes provide you with information regarding points of interest in the product. While release notes are shipped with each tool kit, the **SDK Release Notes** web page is kept up to date with information that did not make the initial release of the product.

Developer Support Hotline

In addition to the free online support, you also have the option of calling MetraTech Customer Services. Your call is answered by a Customer Service Representative who gathers key information and assigns your incident a case number before routing it to the appropriate

Developer Support Engineer. This identifier allows us to prioritize and track your case, and it helps to share additional information, as the case is resolved. Please make sure that you record the reference number of your question or request for assistance. It will help make additional calls on that case quicker and more efficient.

The Developer Support Hotline, (781) 839-8300 option 4, is available from 10:00 AM to 6:00 PM Eastern Standard Time, Monday through Friday.

Appendix A — Troubleshooting

Partner Data Sheet

The Partner Data Sheet provides the information that you need to integrate with MetraTech's reference server and access MetraTech's partner extranet. If you have not received your *Partner Data Sheet* yet, please contact please MetraTech's customer support group.

Planning Aids

The following two Excel spreadsheets are included in the SDK installation to facilitate the planning of your services:

- sample.xls: Sample service definitions
- template.xls: A service definition template

These files are located in the INSTALLDIR/doc directory. A detailed explanation for each field and column is provided in Step 2: Filling Out Service Worksheets on page 62.

Troubleshooting

Consult the release notes that came with the Metering SDK installation for possible last minute changes that might affect the behavior of the Metering SDK. This section contains some general suggestions to help you get your application working correctly.

Compile Problems

If you are having problems compiling an application using the Metering SDK, this checklist contains tips to help you compile.

- Are you able to compile the Simple sample? See *Chapter 3: Samples* for instructions on how to compile the simple sample.
- Is the compiler complaining that it can't find the header file **mtsdk.h** or **sdk_msg.h**? If so, make sure you have added the directory containing these files to your include path, for both Debug and Release versions of your application.
- Is the linker complaining about an undefined entry point? If so, make sure you've added **mtsdk.lib** for the release version and **mtsdkd.lib** for the debug version of your project.
- Have you specified the Debug Multithreaded DLL version of the C runtime library for the debug version of your project?
- Have you specified the Multithreaded DLL version of the C runtime library for the release version of your project?

Runtime problems

If you've successfully compiled your application which uses the Metering SDK but you're unable to run it, this checklist has information to help your diagnose your problem.

- Can you run the Simple sample? The simple.exe application does a basic test of the Metering SDK. It tests that you have connectivity to the metering server, that you have privileges to submit sessions to the server, and that you can load the required DLLs. You should get this sample working before you attempt to debug your own application.
- Are **mtsdk.dll** and **sdk_msg.dll** both in your path? If not, you must either move these DLLs to a directory that is in your path or add the directory that holds them to your path.
- Have you called the Startup method of the **MTMeter** object before calling other metering SDK functions?
- Is your session created properly? Make sure you've initialized each property associated with the service as it's defined on the metering server. Also be sure that you have initialized the properties with the correct data type.
- Have you forgotten a property or added too many properties? You must have exactly the same properties for your session as defined by the service definition.
- Are your property names and service names spelled correctly? These names are case sensitive and must have the same capitalization as the metering server's service definition.

Further troubleshooting

If a method of the Metering SDK is not working for you, examine the return code and error messages contained in the error object returned from **GetLastErrorObject**. The **GetErrorMessageEx** method of the **MTMeterError** class sometimes returns additional information that might help diagnose the source of an error.

If your application still doesn't run correctly, the diagnostic logging capability of the Metering SDK can help diagnose problems. The **EnableDiagnosticLogging** method of the **MTMeter** class allows you to get a trace of the calls made to the SDK as well as some internal debugging messages.

If you are still unable to solve your problem you will need to contact MetraTech at sdk_support@metratech.com. Supplying MetraTech with the output of the diagnostic logging can help MetraTech determine what the problem is and determine if a bug fix is necessary.

Error Codes

Symbolic Name	Description
MT_ERR_PARENT_COMMITTED	Parent session already committed.
MT_ERR_NO_PROPERTY	Property does not exist.
MT_ERR_DUPLICATE_PROPERTY	Property already exists.
MT_ERR_MESSAGE_MODULE_NOT_FOUND	Error message module not found.
MT_ERR_BAD_HTTP_RESPONSE	Bad HTTP response.
MT_ERR_PARSE_ERROR	Unable to parse message.
MT_ERR_NOT_INITIALIZED	Not initialized.
MT_ERR_SERVER_ERROR	Unknown server error.
MT_ERR_UNKNOWN_SERVICE	Server has no knowledge of this service.

Symbolic Name	Description
MT_ERR_BAD_PROPERTY	A property is invalid.
MT_ERR_NO_HOSTS	No metering servers added.

Figure 7: Error Codes