Monetra®

IP, SSL, and DropFile Protocol Specification

Programmer's Guide v7.12
Updated April 2013

This page intentionally left blank.

Table of Contents

This page intentionally left blank.

1 Introduction

1.1 Overview

Monetra boasts a true Server/Client architecture, which allows for ALL system functionality to transpire either on the local machine or remotely across many dispersed systems using one or more (simultaneous) connection methods. The connection methods Monetra uses are completely modular by design, allowing for quick custom integration modules to be created and added. Modules are loaded at runtime, and currently MainStreet distributes and supports these production connectivity methods: IP, SSL, DropFile, XML-HTTP, XML-HTTPS, and XML-DropFile.

This guide covers the IP, SSL, and DropFile connectivity methods. For the XML connectivity methods, please reference our XML Protocol Specification (http://www.monetra.com/documentation.html).

You will also be required to review the Monetra Client Interface Protocol Specification (http://www.monetra.com/documentation.html) to cross-reference each transaction type, which will have multiple corresponding key/value pairs (ie. username, password, action, etc).

1.2 Connectivity Method Overviews

1.2.1 DropFile

NOTE: *Because of security related issues (such as account numbers being stored in plain text for short periods of time), it is unclear whether or not DropFile support meets the requirements for PABP/CISP communication. It is therefore recommended you use either the IP or SSL communication methods.*

This communication method tends to be the easiest to implement. If desired, files can be created by hand that conform to the formatting detailed in this document and placed manually into a waiting transaction directory. If you can write a file, you can communicate with Monetra!

This method is the slowest, since it relies heavily upon the filesystem with no certain way to determine when a transaction has been completely written to disk. Also, all communication occurs in plain-text, so anyone with permissions to read files in the designated transaction directory can retrieve sensitive information. Use at own risk.

This connectivity method is disabled by default.

1.2.2 IP/Sockets

NOTE: *Over public networks (like the Internet) or untrusted private networks, it is required by PABP/CISP that SSL is used instead of IP for communication to ensure card numbers and other critical data is not transmitted in plain text.*

Standard IP based communication is the fastest option available to Monetra. It is unencrypted, making it applicable only to the local machine or trusted private networks. It fully supports interleaving of transactions, with out-of-order responses, but does not have to be utilized in this fashion. It is easy to implement because it shares a lot in common with the dropfile format, but is wrapped in a simple header and footer before being sent 'over the wire'.

As of Monetra 7.12 this connectivity option is disabled by default.

1.2.3 SSL

The SSL communication protocol is identical to that of IP, but all data is encrypted via SSL v3 or TLS v1. Use of a standard encryption library such as OpenSSL (http://www.openssl.org) is highly recommended. Existing IP-based applications can be easily adapted to utilize SSL for communication by simply modifying the transport layer. SSL adds very little overhead to the datastream and shares all the capabilities of IP, such as interleaving and out-of-order responses.

The Default port numbers for this connectivity method are 8665 for user-level functions and 8666 for administrative-level (MADMIN) functions.

2 Transaction Structure

2.1 Common Structure Properties

2.1.1 Definitions

Identifier: A unique string of numbers and/or letters assigned to the transaction. This identifier may be repeated once a response has been given for the original transaction using the identifier.

Message: This is the actual data of a transaction that Monetra interprets and processes. Any data outside of this is strictly for transport/communication.

Start of Transaction: Indicator representing the start of a transaction, commonly known as STX.

End of Transaction: Indicator representing the end of a transaction, commonly known as ETX.

Field Separator: Character to separate one major portion of a message from another.

2.1.2 Basic Structure

The protocol used to communicate with Monetra is a simple text-based protocol. It formats the key/value pairs as detailed in the Monetra Client Interface Protocol Specification as simply as possible. An equals sign (**=**) is used to delimit the key and value. Each key/value pair is then separated from another pair of key/values by an optional carriage return (CR: hex **0x0D**, decimal **13**) followed by a mandatory line feed (LF: hex **0x0A**, decimal **10**). Responses returned from Monetra will follow the same formatting guidelines (though Monetra will always return both a CR and LF after each key/value pair). The last line of a message should also contain a return character to ensure proper parsing by Monetra. The order of the key/value pairs is irrelevant, do not expect Monetra to return the response key/value pairs in any particular order.

The value portion of the key/value pair may be optionally quoted using a double-quote character. Quoting is mandatory if the value itself contains double-quotes or new line characters. Quotes contained within the value must be escaped with another quote, such as is done with RFC4180.

Comments are allowed in the Monetra protocol by prefixing the line with a pound sign (#).

2.1.3 Example of a Basic Transaction

```
username=vitale
password=test123
action=sale
account=4012888888881
expdate=0512
amount=12.00
comment=”Hello world
this is a new line
””this is a quoted line”””
extravalue1=data1
extravalue2=data2
```

Note: extravalue1, extravalue2, etc are there as an example of how to add additional key/value pairs. You may replace them with proper values as designated in the Monetra Client Interface Protocol Specification.

2.2 Connectivity-specific formatting requirements

2.2.1 Drop File

Monetra periodically scans the configured drop-file directory for files using the reserved suffix, **`.trn.`** When a filename with a matching suffix is located, the file is read into memory and deleted from the directory. The prefix of the file (the content before the **`.trn`**), is stored in Monetra as the ***identifier.*** This filename MUST attempt to be unique, as it is the only identifying element to the transaction for the client. The identifier will be used to generate the response file name, except with a **`.rtn`** extension. The data written to the file is the basic message detailed earlier in this document. No headers or footers are required.

For example, if someone were to write a file named **`12A426B.trn`** into the transaction directory, Monetra would read the file, delete it, and write a response file named **`12A426B.rtn`**. Please note that the extensions/suffixes are case sensitive!

2.2.2 IP/Socket

A standard IP connection should be made to the Monetra engine. There is no additional 'handshake' procedure that needs to be performed. Transactions may be sent immediately upon successful connection. The default IP port number for communication is 8333.

The basic transaction structure is as follows:
<*STX*>`identifier`**<*FS*>**`message`**<*ETX*>**

Where:
<*STX*> = Hex: 0x02 Dec: 02
<*FS*> = Hex: 0x1c Dec: 28
<*ETX*> = Hex: 0x03 Dec: 03

The identifier should be unique to the session, but it may be reused once a response is received from a transaction that shares the same identifier. The identifier may be any unique string of numbers and letters and will be echoed in the response Monetra provides.

Both requests and responses share the same formatting requirements. The message portion of the data stream is formatted as detailed above.

If a connection to Monetra is lost while there are pending transactions for your connection, you must issue a QC (queue check), GUT (get unsettled transactions), or GFT (get failed transactions) report in order to determine the status of your transaction. There is no recovery process to resume a failed session, therefore a stable connection is strongly recommended.

2.2.3 SSL

We suggest using preexisting free libraries to perform encryption and decryption with SSL. The most widely used libraries are available at http://www.openssl.org/. These libraries will help perform all encryption "behind the scenes". The protocol used is identical to the IP/Socket protocol utilized above, except the default ports used are 8665 (user), and 8666 (admin). Please reference that section for implementation details.

3 Examples

3.1 PHP

Though it is recommended to use our PHP module to communicate with Monetra, it is by no means a requirement, and can be easily done using 100% PHP code, and it's internal socket support. Please reference the snippet below for the basic structuring and parsing. Please note that this does not go into parsing of the key/value pairs, or comma delimited data as this is left as an exercise for the reader. What it does show you is basic communication (protocol wrappers, formatting), and error handling. Also note there is an example using the XML protocol for PHP in the Monetra XML Protocol Specification.

```
<?php
$identifier = "sadg34ytl4yt1"; /* Some random, unique data */
/* Transaction data, key/value pairs formatted as described */
$message = "username=vitale\n" .
        "password=test\n" .
        "action=sale\n" .
        "account=4012888888881\n" .
        "expdate=0512\n" .
        "amount=12.00\n";
/* Protocol Wrappers for IP */
$trans = chr(0x02) . $identifier . chr(0x1c) . $message . chr(0x03);
$error = "";
$errno = 0;
/* Open the connection */
$sock = fsockopen("localhost", 8333, $errno, $error, 5);
if (!$sock) {
        echo "Connection error: $error\n";
        return;
}
echo "Connected ...\n";
/* Write the entire transaction */
if (fwrite($sock, $trans) != strlen($trans)) {
        echo "Error writing...\n";
        return;
}
echo "Wrote Transaction...\n";
/* Read until we receive an ETX, or an error condition */
$read_data = "";
while (strchr($read_data, chr(0x03)) === FALSE) {
        $mydata = fread($sock, 1024);
        if ($mydata === FALSE) {
                echo "Disconnect before response\n";
                return;
         }
```

```
        $read_data .= $mydata;
}
echo "Got Response\n";
fclose($sock);
/* Find our STX, FS, and ETX characters */
$stx_loc = strpos($read_data, chr(0x02));
$fs_loc = strpos($read_data, chr(0x1c));
$etx_loc = strpos($read_data, chr(0x03));
if ($stx_loc === FALSE || $fs_loc === FALSE || $etx_loc === FALSE) {
        echo "Invalid response format\n";
        return;
}
/* Parse out our identifier, and response message */
$read_identifier = substr($read_data, $stx_loc+1, $fs_loc - $stx_loc -
1);
$read_message = substr($read_data, $fs_loc+1, $etx_loc - $fs_loc - 1);
/*  Make sure the identifier that was returned is the same
  * one that was sent. Remember, you can restructure this code
  * to send multiple transactions at once, each with different
  * identifiers. */
if  (strcmp($read_identifier, $identifier) != 0) {
        echo "Got response for different transaction than
expecting!\n";
        return;
}
echo "Response:\n$read_message\n";
/* Parse response here, remember, this may be comma-delimited data
  * on certain requests, or key/value pairs ... */
?>
```