Python Development Guide

Python Development Guide

5.2

Publication date May 23, 2017

Table of Contents

Chapter 1. Introduction

This document explains how to use the Python client for AMPS. Use this document to learn how to install, configure and develop applications using the AMPS Python Client API.

1.1. Prerequisites

Before reading this book, it is important to have a good understanding of the following topics:

- Developing in Python. To be successful using this guide, you will need to possess a working knowledge of the Python language. Visit `http://www.python.org` for resources on learning Python.
- AMPS concepts. Before reading this book, you will need to understand the basic concepts of AMPS, such as *topics*, *subscriptions*, *messages*, and *SOW*. Consult the *AMPS Users' Guide* to learn more about these topics before proceeding.
- An installed python interpreter. The AMPS Python Client currently supports python version 2.6 or greater, but less than version 3.0.
- Python setuptools. This is included with many Python distributions. If it is not already installed on your system, you must add setuptools to your installation. See `http://pythonhosted.org/setuptools/` for information on setuptools.

60East also provides a precompiled `.egg` built for x64 Linux with Python 2.7. If you are running on a different system, or using a different version of Python, you will also need:

- Python distutils. Most Python installations contain this package by default.
- C++ compiler. To build the client, you will need a C++ compiler as described in Section 1.2. This is only necessary for inital installation. You do not need a C++ compiler to run the client.

You will need an installed and running AMPS server to use the product as well. You can write and compile programs that use AMPS without a running server, but you will get the most out of this guide by running the programs against a working server.

1.2. Python Support

This section describes the features supported by the Python client at runtime, as well as the requirements for building and installing the Python client.

Runtime

The AMPS Python client supports Python 2.6 or later 2.X versions. The AMPS Python client does not support Python 3.X.

The features supported on your processor and operating system depend on the features supported by the underlying C++ client, as shown in the following table:

Table 1.1. C++ client supported features

	Linux x64	Windows x64	Solaris SPARC
Incredible performance	✓	✓	✓
Publish and subscribe	✓	✓	✓
State of the World (SOW) queries	✓	✓	✓
Topic and content filtering	✓	✓	✓
Atomic SOW query and subscribe	✓	✓	✓
Transaction log replay	✓	✓	✓
Historical SOW query	✓	✓	✓
Beautiful documentation	✓	✓	✓
HA: automatic failover	✓	✓	
HA: durable publish and subscribe	✓	✓	

Installation

To install or build the Python client, follow the instructions in Chapter 2.

To build the AMPS Python client, you need to have a compatible version of Python, as well as Python `distutils` installed. You also need a supported C++ compiler for your operating system:

- Linux: gcc 4.8 (recommended), 4.6, or 4.4
- Windows: Visual Studio 2010 or later
- Solaris: Oracle Solaris Studio 12.3

Notice that for a distribution of Python to be compatible, it must be a supported version built with a supported compiler for your operating system.

Chapter 2. Building and Installing the AMPS Client

2.1. Obtaining the Client

The AMPS Python client is available as a download from the 60East Technologies web site, `www.crankuptheamps.com`. Download the client from the site, then extract it.

The client source files are in the directory where you unpacked the files. By default, this is `amps-python-client-<version>`, where `<version>` is the current version of the python client, such as `amps-python-client-4.3.1.0`).

2.2. Installing the Prebuilt .egg on Linux

60East provides a prebuilt `.egg` file for x64 Linux distributions using Python 2.7. To install the `.egg`:

- To successfully install the file, you will need permission to update the Python distribution on the system you are installing on.
- Open a command prompt
- Run the following command, substituting the version number that you want to install for the *<version_number>* placeholder:

  ```
  $ easy_install \
    http://devnull.crankuptheamps.com/releases/amps/clients/
  amps_python_client-<version_number>-py2.7-linux-x86_64.egg
  ```

  If this command reports a permission error, you do not have permission to update the Python distribution. Run the command as a different user, or use `sudo` to run the command as `root`.

2.3. Installing the Prebuilt .egg on Windows

60East provides a prebuilt `.egg` file for 64-bit Windows operating systems using Python 2.7. Your Python distribution may include a more fully-featured package manager to assist with installing `.egg` files.

To install the `.egg` using the command line:

- To successfully install the file, you will need permission to update the Python distribution on the system you are installing on.
- Make sure that your Python distribution includes setuptools.
- Open a command prompt.
- Make sure that the directory that contains easy_install.exe from setuptools is is in your `PATH`. For example, if your distribution installs the setuptools scripts into `C:\Python27\Scripts`, you might set your `PATH` as follows:

```
set PATH=%PATH%;C:\Python27\Scripts
```

- Run the following command, substituting the version number that you want to install for the *<version_number>* placeholder:

```
$ easy_install.exe \
  http://devnull.crankuptheamps.com/releases/amps/clients/
amps_python_client-<version_number>-py2.7-win-amd64.egg
```

  If this command reports a permission error, you do not have permission to update the Python distribution. Run the command prompt from a different user, or start the command prompt with **Run as Administrator**.

2.4. Building the Client

The main AMPS Python client distribution includes the full source to the client. For most installations, you build the client with your Python distribution before using it. The process for building the client differs slightly depending on whether you are building the client for Linux or for Windows.

Building for Linux

Follow these steps to build the Linux version of the client:

1. The Python client includes all necessary C++ client sources. If you are using a different version of the C++ client than the one included with the Python client, set the `AMPS_CPP_DIR` environment variable to the location of the AMPS C++ client source.

2. Run `python setup.py build` from the AMPS Python Client directory to build the client.

   This script uses the Python distutils to build the library, which makes it easy to build the library correctly and install the library into your Python distribution. To see the options available in your environment, run `python setup.py --help`.

   

   The module must be built with the same C++ compiler used to build python on your system. The `setup.py` script and `distutils` package will generally ensure this unless you have added a different compiler to your `PATH`. If you typically use a different C++ compiler, remove the path to that compiler before running `setup.py`.

3. The script builds the module to a path in the `build` directory. The exact path depends on the version of Python you are building with.

4. Add the build directory path to the PYTHONPATH environment variable. For example:

   ```
   $ export PYTHONPATH=/home/AMPSdev/amps-python-client/build/lib.linux-
   x86_64-2.7:$PYTHONPATH
   ```

5. Test that the module loads correctly:

   ```
   $ python -c "import AMPS"
   ```

6. Optionally, install the module to your local python installation. While this is not required, doing this makes the AMPS module available for all of the programs that use this installation of python:

```
sudo python setup.py install
```

Building for Windows

Follow these steps to build the Windows version of the client:

1. Use the Visual Studio Command Prompt shortcut to start a command prompt window for the type of module you want to build. For example, to build a 32-bit module, you use the command prompt for x86 builds.

2. Add the Python directory (the location of the `python.exe` interpreter) to your `PATH`.

The platform of the python installation must match the target platform for the python module. If you want to build a 64-bit module, your PATH must include a 64-bit python installation. If you want to build a 32-bit module, your PATH must include a 32-bit python installation. The build process uses whatever python.exe is found first when searching the PATH. So, to build both 32-bit and 64-bit versions, you must build them separately, and change your PATH between builds.

3. Run `python setup.py build` from the AMPS Python Client directory to build the client.

   This script uses the Python distutils to build the library, which makes it easy to build the library correctly and install the library into your Python distribution. To see the options available in your environment, run `python setup.py --help`.

4. The script builds the module to a path in the `build` directory. The exact path depends on the version of Python you are building with.

5. Add the build directory path to the PYTHONPATH environment variable. For example:

   ```
   >   set   PYTHONPATH="C:\Users\AMPSdev\amps-python-client\build\lib.linux-x86_64-2.7;%PYTHONPATH%"
   ```

6. Test that the module loads correctly:

   ```
   > python -c "import AMPS"
   ```

7. Optionally, install the module to your local python installation.While this is not required, doing this makes the AMPS module available for all of the programs that use this installation of python:

   ```
   > python setup.py install
   ```

Chapter 3. Your First AMPS Program

In this chapter, we will learn more about the structure and features of the AMPS Python client and build our first Python program using AMPS.

3.1. About the client library

The AMPS client library is packaged as a single binary file, `AMPS.so` on Linux or `AMPS.pyd` on Windows. You can find the file under the `build` directory of your AMPS Python client install once you've completed the build process. Every Python application you build will need to reference this file, and the file must be deployed along with your application in order for your application to function properly.

If you chose to install the python client into your local Python installation, then Python has access to the client library in the installation, and you do not need to include the library with each specific script. Otherwise, you will need to package and include the library with your script and ensure that the library is in the path where python looks for shared libraries.

3.2. Connecting to AMPS

Let's begin by writing a simple program that connects to an AMPS server and sends a single message to a topic. This code will be covered in detail just following the example.

```
❶import AMPS
 import sys
 ❷uri_ = "tcp://127.0.0.1:9007/amps/json"

 ❸client = AMPS.Client("examplePublisher")
 ❹try:
   ❺client.connect(uri_)
   ❻client.logon()

   ❼client.publish("messages", '{ "hi" : "Hello, world!"}')
 ❽except AMPS.AMPSException as e:
    sys.stderr.write(str(e))

❾client.publish_flush()
❿client.close()
```

Example 3.1. Connecting to AMPS

❶ These import the `AMPS` and `sys` packages. All programs written using the AMPS Python Client will need to include the `import AMPS` statement at a minimum.

❷ The URI to use to connect to AMPS. The URI consists of the transport, the address, and the protocol to use for the AMPS connection. In this case, the transport is `tcp`, the address is `127.0.0.1:9007`, and the protocol is `amps`. This connection will be used for JSON messages. Check with the person who manages the AMPS instance to get the connection string to use for your programs.

❸ This line creates a new `Client` object. `Client` encapsulates a single connection to an AMPS server. Methods on `Client` allow for connecting, disconnecting, publishing, and subscribing to an AMPS server. The argument to the Client constructor, "exampleClient", is a name chosen by the client to identify itself to the server. Errors

relating to this connection will be logged with reference to this name, and AMPS uses this name to help detect duplicate messages. AMPS enforces uniqueness for client names when a transaction log is configured, and it is good practice to always use unique client names.

❹ Here, we open a try block that concludes with `except AMPS.AMPSException as e`. All exceptions in AMPS derive from `AMPSException`. If an operation throws another exception, AMPS will wrap that exception into the `AMPSException` you receive. For example, if the call to `connect()` fails because the provided address is not reachable, the `AMPSException` will contain an inner exception from the operating system, likely a `SocketException`.

❺ With this statement, we provide URI to the client and declare the AMPS connection.

❻ The AMPS `logon()` command connects to AMPS and creates a named connection. If we had provided logon credentials in the URI, the command would pass those credentials to AMPS. Without credentials, the client logs on to AMPS anonymously. AMPS versions 5.0 and later require a `logon()` command in the default configuration.

❼ Here, we publish a single message to AMPS on the messages topic, containing the data `Hello, world!`. This data is placed into a JSON message and sent to the server. Upon successful completion of this function, the AMPS client has enqueued the message to be sent to the server, and subscribers to the `messages` topic will receive this JSON message: `{ "hi" : "Hello, world!" }`.

❾ Here, we call `publish_flush` on the client. This command waits until all messages from the client have been sent. If messages may still be in the process of transmission when your application is ready to close the client, `publish_flush` helps ensure that the messages have been sent. In this case, we allow `publish_flush` to block indefinitely. A production application might specify a timeout, to avoid hanging in the event that the application loses connectivity before the messages have been sent.

❿ We close the connection to AMPS. While that doesn't matter here, since the application exits immediately after calling `close()`, it's good practice to close connections when you are done using them.

3.3. Connection Strings

The AMPS clients use connection strings to determine the server, port, transport, and protocol to use to connect to AMPS. When the connection point in AMPS accepts multiple message types, the connection string also specifies the precise message type to use for this connection. Connection strings have a number of elements.

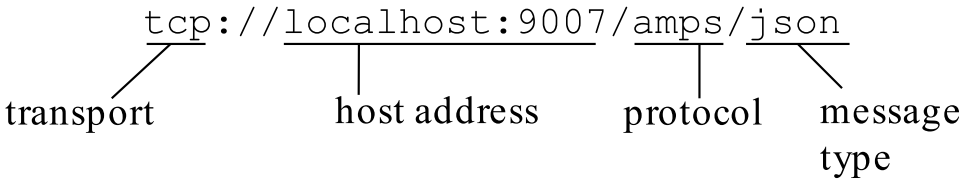

Figure 3.1. elements of a connection string

As shown in the figure above, connection strings have the following elements:

- *Transport* defines the network used to send and receive messages from AMPS. In this case, the transport is `tcp`. For connections to transports that use the Secure Sockets Layer (SSL), use `tcps`. For connection to AMPS over a Unix domain socket, use `unix`.

- *Host address* defines the destination on the network where the AMPS instance receives messages. The format of the address is dependent on the transport. For `tcp` and `tcps`, the address consists of a host name and port number. In this case, the host address is `localhost:9007`. For `unix` domain sockets, a value for hostname and port must be provided to form a valid URI, but the content of the hostname and port are ignored, and the file name provided in the ***path*** parameter is used instead (by convention, many connection strings use `localhost:0` to indicate that this is a local connection that does not use TCP/IP).

- *Protocol* sets the format in which AMPS receives commands from the client. Most code uses the default `amps` protocol, which sends header information in JSON format. AMPS supports the ability to develop custom protocols as extension modules, and AMPS also supports legacy protocols for backward compatibility.
- *MessageType* specifies the message type that this connection uses. This component of the connection string is required if the protocol accepts multiple message types and the transport is configured to accept multiple message types. If the protocol does not accept multiple message types, this component of the connection string is optional, and defaults to the message type specified in the transport.

  Legacy protocols such as `fix`, `nvfix` and `xml` only accept a single message type, and therefore do not require or accept a message type in the connection string.

As an example, a connection string such as

```
tcp://localhost:9007/amps/json
```

would work for programs connecting from the local host to a `Transport` configured as follows:

```
<AMPSConfig>
...
    <!-- This transport accepts any known
         message type for the instance: the
         client must specify the message type.
      -->
    <Transport>
      <Name>any-tcp</Name>
      <Type>tcp</Type>
      <InetAddr>9007</InetAddr>
      <ReuseAddr>true</ReuseAddr>
      <Protocol>amps</Protocol>
    </Transport>
...
</AMPSConfig>
```

See the *AMPS Configuration Guide* for more information on configuring transports.

Providing Credentials in a Connection String

When using the default authenticator, the AMPS clients support the standard format for including a username and password in a URI, as shown below:

```
tcp://user:password@host:port/protocol/message_type
```

When provided in this form, the default authenticator provides the username and password specified in the URI. If you have implemented another authenticator, that authenticator controls how passwords are provided to the AMPS server.

3.4. Connection Parameters

When specifying a URI for connection to an AMPS server, you may specify a number of transport-specific options in the parameters section of the URI. Here is an example:

```
tcp://localhost:9007/amps/json?tcp_nodelay=true&tcp_sndbuf=100000
```

In this example, we have specified the AMPS instance on `localhost`, port `9007`, connecting to a transport that uses the `amps` protocol and sending JSON messages. We have also set two parameters, `tcp_nodelay`, a Boolean (true/false) parameter, and `tcp_sndbuf`, an integer parameter. Multiple parameters may be combined to finely tune settings available on the transport. Normally, you'll want to stick with the defaults on your platform, but there may be some cases where experimentation and fine-tuning will yield higher or more efficient performance.

The AMPS client supports the value of `tcp` in the *scheme* component connection string for TCP/IP connections, and the value of `tcps` as the scheme for SSL encrypted connections.

For connections that use Unix domain sockets, the client supports the value of `unix` in the scheme, and requires the additional option described below.

TCP and SSL transport options

The following transport options are available for TCP connections:

`tcp_rcvbuf` (integer) Sets the socket receive buffer size. This defaults to the system default size. (On Linux, you can find the system default size in `/proc/sys/net/core/rmem_default`.)

`tcp_sndbuf` (integer) Sets the socket send buffer size. This defaults to the system default size. (On Linux, you can find the system default size in `/proc/sys/net/core/wmem_default`.)

`tcp_nodelay` (boolean) Enables or disables the `TCP_NODELAY` setting on the socket. By default `TCP_NODELAY` is disabled.

`tcp_linger` (integer) Enables and sets the `SO_LINGER` value for the socket. By default, `SO_LINGER` is enabled with a value of `10`, which specifies that the socket will linger for 10 seconds.

`tcp_keepalive` (boolean) . Enables or disables the `SO_KEEPALIVE` value for the socket. The default value for this option is true.

Unix transport parameters

The `unix` transport type communicates over unix domain sockets. This transport *requires* the following additional option:

`path` The path to the unix domain socket to connect to.

Unix domain sockets always connect to the local system. When the scheme specified is `unix`, the host address is *ignored* in the connection string. For example, the connection string:

```
unix://localhost:0/amps/json?path=/sockets/the-amps-socket
```

and the connection string:

```
unix://unix:unix/amps/json?path=/sockets/the-amps-socket
```

are equivalent.

The other components of the connection string, including the *protocol*, *message type*, *user name*, and *authentication token* are processed just as they would be for TCP/IP sockets.

3.5. Next steps

You are now able to develop and deploy an application in Python that publishes messages to AMPS. In the following chapters, you will learn how to subscribe to messages, use content filters, work with SOW caches, and fine-tune messages that you send.

Chapter 4. Subscriptions

Messages published to a topic on an AMPS server are available to other clients via a subscription. Before messages can be received, a client must subscribe to one or more topics on the AMPS server so that the server will begin sending messages to the client. The server will continue sending messages to the client until the client unsubscribes, or the client disconnects. With content filtering, the AMPS server will limit the messages sent to only those messages that match a client-supplied filter. In this chapter, you will learn how to subscribe, unsubscribe, and supply filters for messages using the AMPS Python client.

4.1. Subscribing to a Topic

The AMPS client makes it simple to subscribe to a topic. You call `client.subscribe()` with the topic to subscribe to and the parameters for the subscription. The client submits the subscription to AMPS and returns a MessageStream that you can iterate over to receive the messages from the subscription. Here is a short example:

```
from AMPS import Client

client = Client("test")❶
client.connect("tcp://127.0.0.1:9007/amps/json")
client.logon()

for message in client.subscribe("messages"):❷
    print message.get_data()❸
```

❶ Here we have created or received a Client that is properly connected to an AMPS server.

❷ Here we subscribe to the topic `messages`. We do not provide a filter, so AMPS does not content-filter the topic. Although we don't use the object explicitly here, the `subscribe` method returns a `MessageStream` object that we iterate over. If, at any time, we no longer need to subscribe, we can break out of the loop. When there are no more active references to the `MessageStream`, the client sends an `unsubscribe` command to AMPS and stops receiving messages.

❸ Within the loop, we process the message. In this case, we simply print the contents of the message.

AMPS creates a background thread that receives messages and copies them into the `MessageStream` that you iterate over. This means that the client application as a whole can continue to receive messages while you are doing processing work.

The simple method described above is provided for convenience. The AMPS Python client provides convenience methods for the most common form of AMPS commands. The client also provides an interface that allows you to have precise control over the command. Using that interface, the example above becomes:

```
from AMPS import Client
from AMPS import Command

client = Client("test")❶
client.connect("tcp://127.0.0.1:9007/amps/json")
client.logon()

for message in \ ❷
        client.execute(Command("subscribe").set_topic("messages")):
           print message.get_data()❸
```

Example 4.1. Subscribing to a topic using a command

❶ Here, we create a `Client` object and connect to an AMPS server.
❷ Here, we create a `Command` object for the `subscribe` command, specifying the topic `messages`. We do not provide a filter, so AMPS does not content-filter the subscription. Although we don't use the object explicitly here, the `execute` method returns a MessageStream object that we iterate over. If, at any time, we no longer need to subscribe, we can break out of the loop. When we break out of the loop, there are no more references to the MessageStream and the AMPS client sends an unsubscribe message to AMPS.
❸ Within the body of the loop, we can process the message as we need to. In this case, we simply print the contents of the message.

The `Command` interface allows you to precisely customize the commands you send to AMPS. For flexibility and ease of maintenance, 60East recommends using the `Command` interface (rather than a named method) for any command that will receive messages from AMPS. For publishing messages, there can be a slight performance advantage to using the named commands where possible.

Asynchronous Message Processing Interface

The AMPS Python client also supports an interface that allows you to process messages asynchronously. In this case, you add a message handler to the method call. The client returns the command ID of the subscribe command once the server has acknowledged that the command has been processed. As messages arrive, the client calls your message handler directly on the background thread. This can be an advantage for some applications. For example, if your application is highly multithreaded and copies message data to a work queue processed by multiple threads, there may is usually a performance benefit to enqueuing work directly from the background thread. See Section 4.3 for a discussion of threading considerations, including considerations for message handlers.

Here is a short example (error handling and connection details are omitted for brevity):

```
from AMPS import Client
from AMPS import Command

...

client = Client("exampleSubscriber")❶
 client.connect("tcp://127.0.0.1:9007/amps/json")
 client.logon()

 def on_message_printer(message):
     print message.get_data()

 subscriptionid = client.execute_async( ❷
    Command("subscribe")       \
       .set_topic("messages"), \
    on_message_printer) ❸
```

Example 4.2. Subscribing to a topic with asynchronous processing

❶ Here we have created or received a Client that is properly connected to an AMPS server.
❷ Here, we create a subscription with the following parameters:

`Command`	The AMPS Command object that contains the `subscribe` command.
on_message_printer	The message handler function. This function is called on a background thread each time a message arrives.

❸ `on_message_printer` is a function that acts as our message handler. When a message is received, this function is invoked, and in this case, the `get_data()` method from message is printed to the screen. `message` is of type `AMPS.Message`.

The AMPS client resets and reuses the message provided to this function between calls. This improves performance in the client, but means that if your handler function needs to preserve information contained within the message, you must copy the information rather than just saving the message object. Otherwise, the AMPS client cannot guarantee the state of the object or the contents of the object when your program goes to use it.

Using an Instance Method as a Message Handler

One of the more common ways of providing a message handler is as an instance method on an object that maintains message state. It's simple to provide a handler with this capability, as shown below.

```
# Define a class that saves state and
# handles messages.
class StatefulHandler:

    # Initialize self with state to save
    def __init__(self, name):
       self.name_ = name

    # Use state from this instance while handling
    # the message.
    def __call__(self, message):
        print "%s got %s" % (self.name_, message.get_data())
```

You can then provide an instance of the handler directly wherever a message handler is required, as shown below:

```
c.subscribe(StatefulHandler("An instance"), "topic")
```

4.2. Unsubscribing

With a `MessageStream`, AMPS automatically unsubscribes to the topic when there are no more references to the `MessageStream`. You can also call the `close()` method on the `MessageStream` object to remove the subscription.

With asynchronous message processing, when a subscription is successfully made messages will begin flowing to the message handler function and the `Client.subscribe()` call returns a unique string that serves as the identifier for this subscription. A `Client` can have any number of active subscriptions, and this string is used to refer to the particular subscription we have made here. For example, to unsubscribe, we simply pass in this identifier:

```
client = Client("exampleClient")

# Register an asynchronous subscription

subId = client.execute_async(                          \
```

```
          Command("subscribe").set_topic("messages"), \
          on_message_printer)

...

# when the program is done with the subscription, unsubscribe

for msg in client.execute( \
          Command("unsubscribe") \
             .set_sub_id(subId)):
    print "Response to unsubscribe: %s" % msg.get_ack_type()
```

Example 4.3. Unsubscribing from a topic

In this example, as in the previous section, we use the `Client.subscribe()` method to create a subscription to the `messages` topic. When our application is done listening to this topic, it unsubscribes (on the last line) by passing in the `subscriptionId` returned by `subscribe()`. After the subscription is removed, no more messages will flow into our `on_message_printer` function.

AMPS also accepts a comma-delimited list of subscription identifiers to the `unsubscribe` command, or the keyword `all` to unsubscribe all subscriptions for the client.

4.3. Understanding Threading and Message Handlers

When you call a subscribe command, the client creates a thread that runs in the background. The command returns, while the thread receives messages. In the simple case, using synchronous message processing, the client provides an internal handler function that populates the `MessageStream`. The `MessageStream` is used on the calling thread, so operations on the `MessageStream` do not block the background thread.

When using asynchronous message processing, AMPS calls the handler function from the background thread. Message handlers provided for asynchronous message processing must be aware of the following considerations.

The client creates one background thread per client object. A message handler that is only provided to a single client will only be called from a single thread. If your message handler will be used by multiple clients, then multiple threads will call your message handler. In this case, you should take care to protect any state that will be shared between threads.

For maximum performance, do as little work in the message handler as possible. For example, if you use the contents of the message to update an external database, a message handler that adds the relevant data to an update queue that is processed by a different thread will typically perform better than a message handler that does this update during the message handler.

While your message handler is running, the thread that calls your message handler is no longer receiving messages. This makes it easier to write a message handler, because you know that no other messages are arriving from the same subcription. However, this also means that you cannot use the same client that called the message handler to send commands to AMPS. Acknowledgements from AMPS cannot be processed, and your application will deadlock waiting for the acknowledgement. Instead, enqueue the command in a work queue to be processed by a separate thread, or use a different client object to submit the commands.

The AMPS client resets and reuses the `Message` provided to this function between calls. This improves performance in the client, but means that if your handler function needs to preserve information contained within the message, you

must copy the information rather than just saving the message object. Otherwise, the AMPS client cannot guarantee the state of the object or the contents of the object when your program goes to use it.

4.4. Understanding messages

So far, we have seen that subscribing to a topic involves working with objects of `AMPS.Message` type. A `Message` represents a single message to or from an AMPS server. Messages are received or sent for every client/server operation in AMPS.

Header properties

There are two parts of each message in AMPS: a set of headers that provide metadata for the message, and the data that the message contains. Every AMPS message has one or more header fields defined. The precise headers present depend on the type and context of the message. There are many possible fields in any given message, but only a few are used for any given message. For each header field, the `Message` class contains a distinct property that allows for retrieval and setting of that field. For example, the `Message.get_command_id()` function corresponds to the `commandId` header field, the `Message.get_batch_size()` function corresponds to the `BatchSize` header field, and so on. For more information on these header fields, consult the *AMPS User Guide* and *AMPS Command Reference*.

To work with header fields, a`Message` contains `get_xxx()`/`set_xxx()` methods corresponding to the header fields, as well as a number of `getXXX()`/`setXXX()` methods for compatibility with previous implementations of the AMPS Python client. 60East does not recommend attempting to parse header fields from the raw data of the message.

get_data() method

Access to the data section of a message is provided via the `get_data()` method. The `data` property will contain the unparsed data of the message. Your application code parses and works with the data.

The AMPS Python client contains a collection of helper classes for working with message types that are specific to AMPS (for example, FIX, NVFIX, and AMPS composite message types). For message types that are widely used, such as JSON or XML, you can use the standard Python facilities or the library you typically use in your environment.

4.5. Advanced Messaging Support

AMPS allows your application to subscribe to topics even when you do not know their exact names, and for providing a filter that works on the server to limit the messages received by your application.

Regex Subscriptions

Regular Expression (Regex) subscriptions allow a regular expression to be supplied in the place of a topic name. When you supply a regular expression, it is as if a subscription is made to every topic that matches your expression, including topics that do not yet exist at the time of creating the subscription.

To use a regular expression, simply supply the regular expression in place of the topic name in the `subscribe()` call. For example:

```
subscription_id = client.subscribe(
    message_handler,
    "orders.*",
    None)
```

Example 4.4. Regex topic subscription

In this example, messages on topics `orders-north-america` and `orders-europe` would match the regular expression, and those messages would all be sent to our `message_handler` function. As in the example, you can use the `get_topic()` method to determine the actual topic of the message sent to the function.

Content filtering

One of the most powerful features of AMPS is content filtering. With content filtering, filters based on message content are applied at the server, so that your application and the network are not utilized by messages that are uninteresting for your application. For example, if your application is only displaying messages from a particular user, you can send a content filter to the server so that only messages from that particular user are sent to the client.

To apply a content filter to a subscription, simply pass it into the `Client.subscribe()` call or use the `set_filter` method to add a filter to the `Command`:

```
for message in client.subscribe( "letters", \
                                 "/sender = 'mom'",\
                                 5000):
 print "Mom says: %s" % message.get_data()
```

Example 4.5. Using content filters

In this example, we have passed in a content filter `"/sender = 'mom'"`. This will cause the server to only send us messages from the messages topic that additionally have a sender field equal to `mom`.

For example, the AMPS server will send the following message, where `/sender` is `mom`:

```
{ "sender" : "mom",
  "text" : "Happy Birthday!",
  "reminder" : "Call me Thursday!" }
```

The AMPS server will not send a message with a different `/sender` value:

```
{ "sender" : "henry dave",
  "text" : "Things do not change; we change." }
```

Updating the Filter on a Subscription

AMPS allows you to update the filter on a subscription. When you replace a filter on the the subscription, AMPS immediately begins sending only messages that match the updated filter. Notice that if the subscription was entered with a command that includes a SOW query, using the `replace` option can re-issue the SOW query (as described in the *AMPS User Guide*).

To update a the filter on a subscription, you create a `subscribe` command. You set the `SubscriptionId` provided on the `Command` to the identifier of the existing subscription, and include the `replace` option on the `Command`. When you send the `Command`, AMPS atomically replaces the filter and sends messages that match the updated filter from that point forward.

4.6. Next steps

At this point, you are able to build AMPS programs in Python that publish and subscribe to AMPS topics. For an AMPS application to be truly robust, it needs to be able to handle the errors and disconnections that occur in any distributed system. In the next chapter, we will take a closer look at error handling and recovery, and how you can use it to make your application ready for the real world.

Chapter 5. Error Handling

In every distributed system, the robustness of your application depends on its ability to recover gracefully from unexpected events. The AMPS client provides the building blocks necessary to ensure your application can recover from the kinds of errors and special events that may occur when using AMPS.

5.1. Exceptions

Generally speaking, when an error occurs that prohibits an operation from succeeding, AMPS will throw an exception. AMPS exceptions universally derive from `AMPS.AMPSException`, so by catching `AMPSException`, you will be sure to catch anything AMPS throws, for example:

```
def read_and_evaluate(client):
    # read a new payload from the user
    payload = input("Please enter a message")

    # write a new message to AMPS
    if payload:
        try:
            client.publish("UserMessage",
            "{ \"message\" : \"%s\" }" \
                    % payload)
        except AMPS.AMPSException as e:
            sys.stderr.write("An AMPS exception" +
                "occurred: %s" % str(e))
```

Example 5.1. Catching an AMPS Exception

In this example, if an error occurs the program writes the error to `stderr`, and the `publish()` command fails. However, `client` is still usable for continued publishing and subscribing. When the error occurs, the exception is written to the console. As with most Python exceptions, `str()` will convert the exception into a string that includes a descriptive message.

AMPS exception types vary based on the nature of the error that occurs. In your program, if you would like to handle certain kinds of errors differently than others, you can handle the appropriate subclass of `AMPSException` to detect those specific errors and do something different.

```
def create_new_subscription(client):
    messageStream = None
    topicName = None

    while messageStream is None:
       ❶topicName = input("Please enter a topic name")
        try:
            messageStream = client.subscribe(
               ❷topicName,
                None)
       ❸except BadRegexTopicError,e:
           print ("Error: bad topic name or regular expression " +
```

```
                topicName + ".  The exception was " + str(e) + ".")
            # we'll ask the user for another topic
        ❹except AMPSException,e:
            print ("Error: error setting up subscription to topic" +
                topicName + ".  The exception was " + str(e) + ".")
            ❺return None
    return messageStream
```

Example 5.2. Handling AMPSException Subclasses

❶ In Example 5.2 our program is an interactive program that attempts to retrieve a topic name (or regular expression) from the user.

❷ If an error occurs when setting up the subscription, the program decides whether or not to try again based on the subclass of `AMPSException` that is thrown. In this case, if the exception is a `BadRegexTopicError`, the exception indicates that the user provided a bad regular expression. We would like to give the user a chance to correct, so we ask the user for a new topic name.

❸ This line indicates that the program catches the `BadRegexTopicError` exception and displays a specific error to the user indicating the topic name or expression was invalid. By not returning from the function in this except block, the while loop runs again and the user is asked for another topic name.

❹ If an AMPS exception of a type other than `BadRegexTopicError` is thrown by AMPS, it is caught here. In that case, the program emits a different error message to the user.

❺ At this point the code stops attempting to subscribe to the client by the `return None` statement.

Exception Types

Each method in AMPS documents the kinds of exceptions that are thrown from it. For reference, table Table A.1 contains a list of all of the exception types you may encounter while using AMPS, when they occur, and what they mean.

Exception Handling and Asynchronous Message Processing

When using asynchronous message processing, exceptions thrown form the message handler are silently absorbed by the AMPS Python client by default. The AMPS Python client allows you to register an exception listener to detect and respond to these exceptions. When an exception listener is registered, AMPS will call the exception listener with the exception. See Example 5.6 for details.

5.2. Disconnect Handling

Every distributed system will experience occasional disconnections between one or more nodes. The reliability of the overall system depends on an application's ability to efficiently detect and recover from these disconnections. Using the AMPS Python client's disconnect handling, you can build powerful applications that are resilient in the face of connection failures and spurious disconnects. For additional reliability, you can also use the high availability client (discussed in the following sections), which provides both disconnect handling and features to help ensure that messages are reliably delivered.

The `HAClient` class, included with the AMPS Python client, contains a disconnect handler and other features for building highly-available applications. The `HAClient` includes features for managing a list of failover servers,

resuming subscriptions, republishing in-flight messages, and other functionality that is commonly needed for high availability. This section covers the use of a custom disconnect handler in the event that the behavior of the `HA-Client` does not suit the needs of your application.

AMPS disconnect handling gives you the ultimate in control and flexibility regarding how to respond to disconnects. Your application gets to specify exactly what happens when an exception occurs by supplying a function to `Client.set_disconnect_handler()` which is invoked whenever a disconnect occurs.

Example 5.3 shows the basics:

```
class MyApp:
    def __init__(self, _uri):
        self.uri = _uri
        self.client = None
        self.client = AMPS.Client(...)
       ❶self.client.set_disconnect_handler(self.attempt_reconnection)
        self.client.connect(self.uri)
        self.client.logon()

    def showMessage(self,m):
        # display order data to the user

   ❷def attempt_reconnection(self, client):
        # simple: just sleep and reconnect
        time.sleep(5)
        self.client.connect(self.uri)
        self.client.logon()
```

Example 5.3. Supplying a Disconnect Handler

❶ In Example 5.3 the `set_disconnect_handler()` method is called to supply a function for use when AMPS detects a disconnect. At any time, this function may be called by AMPS to indicate that the client has disconnected from the server, and to allow your application to choose what to do about it. The application continues on to connect and subscribe to the `orders` topic.

❷ Our disconnect handler's implementation begins here. In this example, we simply try to reconnect to the original server after a 5 second pause. Errors are likely to occur here, therefore we must have disconnected for a reason, but `Client` takes care of catching errors from our disconnect handler. If an error occurs in our attempt to reconnect and an exception is thrown by `connect()`, `Client` will catch it and absorb it, passing it to the `ExceptionListener` if registered. If the client is not connected by the time the disconnect handler returns, AMPS throws `DisconnectedException`.

By creating a more advanced disconnect handler, you can implement logic to make your application even more robust. For example, imagine you have a group of AMPS servers configured for high availability -- you could implement fail-over by simply trying the next server in the list until one is found. Example 5.4 shows a brief example.

```
class MyApp:
    def __init__(self, uris):
        self._currentUri = 0
       ❶self.uris = uris
        self.client = Client(...)
        self.client.set_disconnect_handler(self.connect_to_next_uri)
```

```
    ❷self.connect_to_next_uri(self.client)

def connect_to_next_uri(self,client):
   ❸while true:
        try:
            client.connect(self.uris[self._currentUri])
           ❹client.subscribe(on_message_handler,
                "orders",
                None,
                timeout=5000)
            return
        except AMPSException, e:
            self._currentUri = (self._currentUri +
                1) % len(self.uris)

            print "failed: %s.  Failing over to %s" % (str(e),
                self.uris[self._currentUri])
```

Example 5.4. Simple Client Failover Implementation

❶ Here our application is configured with an array of AMPS server URIs to choose from, instead of a single URI. Our client is constructed and configured with. These will be used in the `connect_to_next_uri()` method as explained below.

❷ `connect_to_next_uri()` is invoked by our disconnect handler. Since our client is currently disconnected, we manually invoke our disconnect handler to initiate the first connection.

❸ In our disconnect handler, we invoke `connect_to_next_uri()`, which loops around our array of URIs attempting to connect to each one. In the `invoke()` method it attempts to connect to the current URI, and if it is successful, returns immediately. If the connection attempt fails, the exception handler for `AMPSException` is invoked. In the exception handler, we advance to the next URI, display a warning message, and continue around the loop. This simplistic handler never gives up, but in a typical implementation, you would likely stop attempting to reconnect at some point.

❹ At this point the client registers a subscription to the server we have connected to. It is important to note that, once a new server is connected, it the responsibility of the application to re-establish any subscriptions placed previously. This behavior provides an important benefit to your application: one reason for disconnect is due to a client's inability to keep up with the rate of message flow. In a more advanced disconnect handler, you could choose to not re-establish subscriptions that are the cause of your application's demise.

Using a Heartbeat to Detect Disconnection

The AMPS client includes a heartbeat feature to help applications detect disconnection from the server within a predictiable amount of time. Without using a heartbeat, an application must rely on the operating system to notify the application when a disconnect occurs. For applications that are simply receiving messages, it can be impossible to tell whether a socket is disconnected or whether there are simply no incoming messages for the client.

When you set a heartbeat, the AMPS client sends a heartbeat message to the AMPS server at a regular interval, and waits a specified amount of time for the response. If the operating system reports an error on send, or if the server does not respond within the specified amount of time, the AMPS client considers the server to be disconnected.

The AMPS client processes heartbeat messages on the client receive thread, which is the thread used for asynchronous message processing. If your application uses asynchronous message processing and occupies the thread for longer than the heartbeat interval, the client may fail to respond to heartbeat messages in a timely manner and may be disconnected by the server.

5.3. Unexpected Messages

The AMPS Python client handles most incoming messages and takes appropriate action. Some messages are unexpected or occur only in very rare circumstances. The AMPS Python client provides a way for clients to process these messages. Rather than providing handlers for all of these unusual events, AMPS provides a single handler function for messages that can't be handled during normal processing.

Your application registers this handler by setting the `last_chance_message_handler` for the client. This handler is called when the client receives a message that can't be processed by any other handler. This is a rare event, and typically indicates an unexpected condition.

For example, if a client publishes a message that AMPS cannot parse, AMPS returns a failure acknowledgement. This is an unexpected event, so AMPS does not include an explicit handler for this event, and failure acknowledgements are received in the method registered as the `last_chance_message_handler`.

Your application is responsible for taking any corrective action needed. For example, if a message publication fails, your application can decide to republish the message, publish a compensating message, log the error, stop publication altogether, or any other action that is appropriate.

5.4. Unhandled Exceptions

When using the asynchronous interface, exceptions can occur that are not thrown to the user. For example, when an exception occurs in the process of reading subscription data from the AMPS server, the exception occurs on a thread inside of the AMPS Python Client. Consider the following example using the asynchronous interface:

```
class MyApp:
    def on_message_handler(self,message):
        print message.get_data()

    def wait_to_be_poked(self,client):
        client.subscribe(
            self.on_message_handler,
            "pokes",
            "/Pokee LIKE %s" % getpass.getuser(),
            timeout=5000)
        f = input("Press a key to exit")
```

Example 5.5. Where do exceptions go?

In this example, we set up a subscription to wait for messages on the pokes topic, whose Pokee tag begins with our user name. When messages arrive, we print a message out to the console, but otherwise our application waits for a key to be pressed.

Inside of the AMPS client, the client creates a new thread of execution that reads data from the server, and invokes message handlers and disconnect handlers when those events occur. When exceptions occur inside this thread, however, there is no caller for them to be thrown to, and by default they are ignored.

In applications that use the asynchronous interface, and where it is important to deal with every issue that occurs in using AMPS, you can set an `ExceptionHandler` via `Client.set_exception_listener()` that receives these otherwise-unhandled exceptions. Making the modifications shown in Example 5.6 to our previous example will allow those exceptions to be caught and handled. In this case we are simply printing those caught exceptions out to the console.

In some cases, the AMPS Python client may wrap exceptions of unknown type into a `AMPSException`. Your exception listener should always include a except block for `AMPSException`.

```
class MyApp:
    def on_exception(self, e):
        print "Exception occurred: %s" % str(e)

    def on_message_handler(self,message):
        print message.get_data()

    def wait_to_be_poked(self, client):
        client.set_exception_listener(self.on_exception)

        # Use the advanced interface to be able to
        # accept input while processing messages.

        client.subscribe(
            self.on_message_handler,
            "pokes",
            "/Pokee LIKE %s" % getpass.getuser(),
            timeout=5000)
        f = input("Press a key to exit")
```

Example 5.6. Exception Listener

In this example we have added a call to `client.set_exception_listener()`, registering a simple function that writes the text of the exception out to the console. If exceptions are thrown in the message handler, those exceptions are written to the console.

AMPS records the stack trace and provides the stack trace to the exception handler, if the provided method includes a parameter for the stack trace. The sample below demonstrates one way to do this. (For sample purposes, the message handler always throws an exception.)

```
import AMPS
import time
import traceback

def handler(message):
  print message
  raise RuntimeError, "in my handler"

def exception_listener(exception, tb):
  print "EXCEPTION RECEIVED", exception
  if tb is not None:
    traceback.print_tb(tb)

client = AMPS.Client("client")

client.set_exception_listener(exception_listener)

client.connect("tcp://localhost:9007/amps")

client.logon()
client.subscribe(handler,"topic")
```

```
client.publish("topic","data")
time.sleep(1)
client.close()
```

5.5. Detecting Write Failures

The `publish` methods in the Python client deliver the message to be published to AMPS then return immediately, without waiting for AMPS to return an acknowledgement. Likewise, the `sow_delete` methods request deletion of SOW messages, and return before AMPS processes the message and performs the deletion. This approach provides high performance for operations that are unlikely to fail in production. However, this means that the methods return before AMPS has processed the command, without the ability to return an error in the event the command fails.

The AMPS Python client provides a `failed_write_handler` that is called when the client receives an acknowledgement that indicates a failure to persist data within AMPS. As with the `last_chance_message_handler` described in Section 5.3, your application registers a handler for this function. When an acknowledgement returns that indicates a failed write, AMPS calls the registered handler method with information from the acknowledgement message, supplemented with information from the client publish store if one is available. Your client can log this information, present an error to the user, or take whatever action is appropriate for the failure.

When no `failed_write_handler` is registered, acknowledgements that indicate errors in persisting data are treated as unexpected messages and routed to the `last_chance_message_handler`. In this case, AMPS provides only the acknowledgement message and does not provide the additional information from the client publish store.

Chapter 6. State of the World

AMPS State of the World (SOW) allows you to automatically keep and query the latest information about a topic on the AMPS server, without building a separate database. Using SOW lets you build impressively high-performance applications that provide rich experiences to users. The AMPS Python client lets you query SOW topics and subscribe to changes with ease.

6.1. Performing SOW Queries

To begin, we will look at a simple example of issuing a SOW query.

```
for message in client.sow("messages-sow"):
        if message.get_command() == Message.Command.GroupBegin:
           print "--- Begin SOW Results ---"
        if message.get_command() == Message.Command.SOW:
           print message.get_data()
        if message.get_command() == Message.Command.GroupEnd:
           print "--- End SOW Results ---"
```

Example 6.1. Basic SOW Query

In listing Example 6.1 the `execute_sow_query()` function invokes `Client.sow()` to initiate a SOW query on the `orders` topic, for all entries that have a symbol of `'ROL'`.

As the query executes, messages containing the data of matching entries have a `Command` of value `sow` or `SOW`, so as those arrive, we write them to the console.

As with `subscribe`, the `sow` command also provides an asynchronous mode, where you provide a message handler.

```
def on_message_handler(message):
    if message.get_command() == Message.Command.GroupBegin:
       print "--- Begin SOW Results ---"
    if message.get_command() == Message.Command.SOW:
       print message.get_data()
    if message.get_command() == Message.Command.GroupEnd:
       print "--- End SOW Results ---"

def execute_sow_query(client):
    return client.execute_async(               \
                Command("sow")                 \
                  .set_topic("orders")         \
                  .set_filter("/symbol='ROL'"),\
               on_message_handler)
```

Example 6.2. Asynchronous SOW Query

In listing Example 6.2 the `execute_sow_query()` function invokes `Client.execute_async()` to initiate a SOW query on the `orders` topic, for all entries that have a symbol of 'ROL'.

As the query executes, the `on_message_handler()` function is invoked for each matching entry in the topic. Messages containing the data of matching entries have a `Command` of value `sow`, so as those arrive, we write them to the console.

6.2. SOW and Subscribe

Imagine an application that displays real time information about the position and status of a fleet of delivery vans. When the application starts, it should display the current location of each of the vans along with their current status. As vans move around the city and post other status updates, the application should keep its display up to date. Vans upload information to the system by posting message to an van `location` topic, configured with a key of `van_id` on the AMPS server.

In this application, it is important to not only stay up-to-date on the latest information about each van, but to ensure all of the active vans are displayed as soon as the application starts. Combining a SOW with a subscription to the topic is exactly what is needed, and that is accomplished by the `Client.sow_and_subscribe()` method. As with the other methods for receiving messages, the AMPS Python client provides a basic, synchronous form of `sow_and subscribe` that provides you with a `MessageStream` to iterate over, and an asynchronous form that requires a message handler.

First, let's look at an example that uses the basic form of `sow_and_subscribe`:

```
def report_van_position(client):
  for message in                                \
     ❶client.execute(                           \
         Command("sow_and_subscribe")             \
           .set_topic("van_location")             \
           .set_filter("/status = 'ACTIVE'")      \
           .set_batch_size(100)                   \
          ❷.set_options("oof")):

     if (message.get_command() == Message.Command.SOW or
         message.get_command() == Message.Command.Publish):
         ❸add_or_update_van(message)
     elif message.get_command() == Message.Command.OOF:
         remove_van(message)
```

Example 6.3. Using `sow_and_subscribe`

❸ For each of these messages we call `add_or_update_van()`, that presumably adds the van to our application's display. As vans send updates to the AMPS server, those are also received by the client because of the subscription placed by `sow_and_subscribe()`. Our application does not need to distinguish between updates and the original set of vans we found via the SOW query, so we use `add_or_update_van()` to display the new position of vans as well.

❶ In listing Example 6.3 we issue a `sow_and_subscribe` command to begin receiving information about all of the active delivery vans in the system. All of the vans in the system now are returned as `Message` objects whose `get_command` returns `sow`. New messages coming in are returned as `Message` objects whose `get_command` returns `publish`.

❷ Notice here that we specified the `oof` option to the command. Setting this option causes AMPS to send *Out-of-Focus* ("OOF") messages for topic. OOF messages are sent when an entry that was sent to us in the past no longer matches our query. This happens when an entry is removed from the SOW cache via a `sow_delete` operation, when the entry expires (as specified by the expiration time on the message or by the configuration of that topic on the AMPS server), or when the entry no longer matches the content filter specified. In our case, if a van's `status` changes to something other than `ACTIVE`, it no longer matches the content filter, and becomes out of focus. When this occurs, a `Message` is sent with `Command` set to `oof`. We use OOF messages to remove vans from the display as they become inactive, expire, or are deleted.

Now we will look at an example that uses the asynchronous form of `sow_and_subscribe`:

```
def update_van_position(message):
    if (message.get_command() == Message.Command.SOW or
            message == Message.Command.Publish):
       add_or_update_van(message)
    elif message.get_command() == Message.Command.OOF:
        remove_van(message)


def subscribe_to_van_location(client):
    client.execute_async(
           Command("sow_and_subscribe")          \
             .set_topic("van_location")          \
             .set_filter("/status = 'ACTIVE'")   \
             .set_batch_size(100)                \
             .set_options("oof"),                \
       update_van_position)
```

Example 6.4. Using `sow_and_subscribe`

Notice that the two forms have the same result. However, one form performs processing on a background thread, and blocks the client from receiving messages while that processing happens, while the other form processes messages on the main thread and allows the background thread to continue to receive messages while processing occurs. In both cases, the calls to `add_or_update_van` and `remove_van` receive the same data

6.3. Setting Batch Size

The AMPS clients include a batch size parameter that specifies how many messages the AMPS server will return to the client in a single batch when returning the results of a SOW query. The 60East clients set a batch size of 10 by default. This batch size works well for common message sizes and network configurations.

Adjusting the batch size may produce better network utilitization and produce better performance overall for the application. The larger the batch size, the more messages AMPS will send to the network layer at a time. This can result in fewer packets being sent, and therefore less overhead in the network layer. The effect on performance is generally most noticeable for small messages, where setting a larger batch size will allow several messages to fit into a single packet. For larger messages, a batch size may still improve performance, but the improvement is less noticeable.

In general, 60East recommends setting a batch size that is large enough to produce few partially-filled packets. Bear in mind that AMPS holds the messages in memory while batching them, and the client must also hold the messages in memory while receiving the messages. Using batch sizes that require large amounts of memory for these operations can reduce overall application peformance, even if network utilization is good.

For smaller message sizes, 60East recommends using the default batch size, and experimenting with tuning the batch size if performance improvements are necessary. For relatively large messages (especially messages with sizes over 1MB), 60East recommends explicitly setting a batch size of 1 as an initial value, and increasing the batch size only if performance testing with a larger batch size shows improved network utilization or faster overall performance.

6.4. Client-Side Conflation

In many cases, applications that use SOW topics only need the current value of a message at the time the message is processed, rather than processing each change that lead to the current value. On the server side, AMPS provides

conflated topics to meet this need. Conflated topics are described in more detail in the AMPS *User Guide*, and require no special handling on the client side.

In some cases, though, it's important to conflate messages on the client side. This can be particularly useful for applications that do expensive processing on each message, applications that are more efficient when processing batches of messages, or for situations where you cannot provide an appropriate conflation interval for the server to use.

A `MessageStream` has the ability to conflate messages recieved for a subscription to a SOW topic, view, or conflated topic. When conflation is enabled, for each message recieved, the client checks to see whether it has already received an unprocessed message with the same `SowKey`. If so, the client replaces the unprocessed message with the new message. The application never receives the message that has been replaced.

To enable client-side conflation, you call `conflate()` on the `MessageStream`, and then use the `MessageStream` as usual:

```
# Query and subscribe
results = ampsClient.sow_and_subscribe("orders", "/symbol == 'ROL'")

# Turn on conflation
results.conflate()

# Process the results
for message in results:
  # process message here
```

When a `MessageStream` is used for a subscription that does not include `SowKeys` (such as a subscription to a topic that does not have a SOW), the `MessageStream` will allow you to turn on conflation, but no conflation will occur.

When using client-side conflation with delta subscriptions, bear in mind that client-side conflation replaces the whole message, and does not attempt to merge deltas. This means that updates can be lost when messages are replaced. For some applications (for example, a ticker application that simply sends delta updates that replace the current price), this causes no problems. For other applications (for example, when several processors may be updating different fields of a message simultaneously), using conflation with deltas could result in lost data, and server-side conflation is a safer alternative.

6.5. Managing SOW Contents

AMPS allows applications to manage the contents of the SOW by explicitly deleting messages that are no longer relevant. For example, if a particular delivery van is retired from service, the application can remove the record for the van by deleting the record for the van.

The client provides the following methods for deleting records from the SOW:

- `sow_delete` accepts a topic and filter, and deletes all messages that match the filter from the topic specified
- `sow_delete_by_keys` accepts a set of SOW keys as a comma-delimited string and deletes messages for those keys, regardless of the contents of the messages. SOW keys are provided in the header of a SOW message, and are the internal identifier AMPS uses for that SOW message.
- `sow_delete_by_data` accepts a topic and message, and deletes the SOW record that would be updated by that message

Most applications use `sow_delete`, since this is the most useful and flexible method for removing items from the SOW. In some cases, particularly when working with extremely large SOW databases, `sow_delete_by_keys` can provide better performance.

In either case, AMPS sends an OOF message to all subscribers who have received updates for the messages removed, as described in the previous section.

The simple form of the `sow_delete` command returns a `Message`. This `Message` is an acknowledgement that contains information on the delete command. For example, the following snippet simply prints informational text with the number of messages deleted:

```
msg = client.sow_delete("sow_topic",
                        "/id IN (42, 64, 37)")

print "Got an %s message containing %s: deleted %s SOW entries" % \
      (msg.get_command(), msg.get_ack_type(), msg.get_matches())
```

The `sow_delete` command also provides an asynchronous version that requires a message handler. This message handler is designed to receive `sow_delete` response messages from AMPS:

```
def delete_handler(m):
   print "Got an %s message containing %s: deleted %s SOW entries" % \
         (m.get_command(), m.get_ack_type(), m.get_matches())

client.execute_async(Command("sow_delete")     \
                        .set_topic("sow_topic") \
                        .set_filter("/id IN (42, 64, 37)"), \
                     delete_handler)
```

Acknowledging messages from a queue uses a form of the `sow_delete` command that is only supported for queues. Acknowledgement is discussed in the chapter on queues.

Chapter 7. Using Queues

AMPS message queues provide a high-performance way of distributing messages across a set of workers. The *AMPS User Guide* describes AMPS queues in detail, including the features of AMPS referred to in this chapter. This chapter does not describe AMPS queues in detail, but instead explains how to use the AMPS Python client with AMPS queues.

To publish messages to an AMPS queue, publishers simply publish to any topic that is collected by the queue. There is no difference between publishing to a queue and publishing to any other topic, and a publisher does not need to be aware that the topic will be collected into a queue.

Subscribers must be aware that they are subscribing to a queue, and acknowledge messages from the queue when the message is processed.

7.1. Backlog and Smart Pipelining

AMPS queues are designed for high-volume applications that need minimal latency and overhead. One of the features that helps performance is the *subscription backlog* feature, which allows applications to receive multiple messages at a time. The subscription backlog sets the maximum number of unacknowledged messages that AMPS will provide to the subscription.

When the subscription backlog is larger than `1`, AMPS delivers additional messages to a subscriber before the subscriber has acknowledged the first message received. This technique allows subscribers to process messages as fast as possible, without ever having to wait for messages to be delivered. The technique of providing a consistent flow of messages to the application is called *smart pipelining*.

Subscription Backlog

The AMPS server determines the backlog for each subscription. An application can set the maximum backlog that it is willing to accept with the `max_backlog` option. Depending on the configuration of the queue (or queues) specified in the subscription, AMPS may assign a smaller backlog to the subscription. If no `max_backlog` option is specified, AMPS uses a `max_backlog` of `1` for that subscription.

In general, applications that have a constant flow of messages perform better with a `max_backlog` setting higher than `1`. The reason for this is that, with a backlog greater than `1`, the application can always have a message waiting when the previous message is processed. Setting the optimum `max_backlog` is a matter of understanding the messaging pattern of your application and how quickly your application can process messages.

To request a `max_backlog` for a subscription, you explicitly set the option on the subscribe command, as shown below:

```
command = Command("subscribe") \
            .set_topic("my_queue" ) \
            .set_options("max_backlog=10")
```

Acknowledging Messages

For each message delivered on a subscription, AMPS counts the message gainst the subscription backlog until the message is explicitly acknowledged. In addition, when a queue specifies `at-least-once` delivery, AMPS re-

tains the message in the queue until the message expires or until the message has been explicitly acknowledged and removed from the queue. From the point of view of the AMPS server, acknowledgement is implemented as a `sow_delete` from the queue with the bookmarks of the messages to remove. The AMPS Python client provides several ways to make it easier for applications to create and send the appropriate `sow_delete`.

Automatic Acknowledgement

The AMPS client allows you to specify that messages should be automatically acknowledged. When this mode is on, AMPS acknowledges the message automatically in the following cases:

- *Asynchronous message processing interface*. The message handler returns without throwing an exception.
- *Synchronous message processing interface*. The application requests the next message from the `MessageStream`.

AMPS batches acknowledgements created with this method, as described in the following section.

To enable automatic acknowledgement batching, use the `set_auto_ack()` method.

```
client.set_auto_ack(True)  # enable AutoAck
```

Message Convenience Method

The AMPS Python client provides a convenience method, `ack()`, on delivered messages. When the application is finished with the message, the application simply calls `ack()` on the message.

For messages that originated from a queue with `at-least-once` semantics, this adds the bookmark from the message to the batch of messages to acknowledge. For other messages, this method has no effect.

```
message.ack() # Add this message to the next
              # acknowledgement batch.
```

Manual Acknowledgement

To manually acknowledge processed messages and remove the messages from the queue, applications use the `sow_delete` command with the bookmarks of the messages to remove. Notice that AMPS only supports using a bookmark with `sow_delete` when removing messages from a queue, not when removing records from a SOW.

For example, given a `Message` object to acknowledge and a client, the code below acknowledges the message.

```
def acknowledgeSingle(client,message):
    acknowledge = new Message()
    acknowledge.set_command("sow_delete")
               .set_topic(message.get_topic())
               .set_bookmark(message.get_bookmark())
    client.send(acknowledge)
```

Example 7.1. Simple Queue Acknowledgement

In listing Example 7.1 the program creates a `sow_delete` command, specifies the topic and the bookmark, and then sends the command to the server. Because the program does not need or expect a response from AMPS, this function uses the `Message` object rather than the `Command` object.

While this method works, creating and sending an acknowledgement for each individual message can be inefficient if your application is processing a large volume of messages. Rather than acknowledging each message individually, your application can build a comma-delimited list of bookmarks from the processed messages and acknowledge all of the messages at the same time. In this case, it's important to be sure that the number of messages you wait for is less than the maximum backlog -- the number of messages your client can have unacknowledged at a given time. Notice that both automatic acknowledgement and the helper method on the `Message` object take the maxiumum backlog into account.

Acknowledgement Batching

The AMPS Python client automatically batches acknowledgements when either of the convenience methods is used. Batching acknowledgements reduces the number of round-trips to AMPS, which reduces network traffic and improves overall performance. AMPS sends the batch of acknowledgements when the number of acknowledgements exceeds a specified size, or when the amount of time since the last batch was sent exceeds a specified timeout.

You can set the number of messages to batch and the maximum amount of time between batches:

```
client.set_ack_batch_size(10)  # Send batch after 10 messages
client.set_ack_timeout(1000)  # ... or 1 second
```

The AMPS Python client is aware of the subscription backlog for a subscription. When AMPS returns the acknowledgement for a subscription that contains queues, AMPS includes information on the subscription backlog for the subscription. If the requested batch size is larger than the subscription backlog, the AMPS Python client adjusts the requested batch size to match the subscription backlog.

Returning a Message to the Queue

A subscriber can also explicitly release a message back to the queue. AMPS returns the message to the queue, and redelivers the message just as though the lease had expired. To do this, the subscriber sends a `sow_delete` command with the bookmark of the message to release and the cancel option.

When using automatic acknowledgements and the asynchronous API, AMPS will cancel a message if an exception is thrown from the message handler.

Chapter 8. Delta Publish and Subscribe

8.1. Introduction

Delta messaging in AMPS has two independent aspects:

- *delta subscribe* allows subscribers to receive just the fields that are updated within a message.
- *delta publish* allows publishers to update and add fields within a message by publishing only the updates into the SOW.

This chapter describes how to create delta publish and delta subscribe commands using the AMPS Python client. For a discussion of this capability, how it works, and how message types support this capability see the *AMPS User Guide*.

8.2. Delta Subscribe

To delta subscribe, you simply use the `delta_subscribe` command as follows:

```
# assumes that client is connected and logged on

cmd = AMPS.Command("delta_subscribe")
cmd.set_topic("delta_topic")
cmd.set_filter("/thingIWant = 'true'")

for m in client.execute(cmd):
    # Delta messages arrive here
```

As described in the *AMPS User Guide*, messages provided to a delta subscription will contain the fields used to generate the SOW key and any changed fields in the message. Your application is responsible for choosing how to handle the changed fields.

8.3. Delta Publish

To delta publish, you use the `delta_publish` command as follows:

```
# assumes that client is connected and logged on

msg = ... # obtain changed fields here

client.delta_publish("myTopic", msg)
```

The message that you provide to AMPS must include the fields that the topic uses to generate the SOW key. Otherwise, AMPS will not be able to identify the message to update. For SOW topics that use a User-Generated SOW Key, use the `Command` form of `delta_publish` to set the `SowKey`.

```
# assumes that client is connected and logged on

```

```
msg = ... # obtain changed fields here
key = ... # obtain user-generated SOW key

cmd = AMPS.Command("delta_publish")
cmd.set_topic("delta_topic")
cmd.set_sow_key(key)
cmd.set_data(msg)

# Execute the delta publish. Use None for
# a message handler since any failure acks will
# be routed to the FailedWriteHandler
client.execute_async(cmd,None)
```

Chapter 9. High Availability

The AMPS Python Client provides an easy way to create highly-available applications using AMPS, via the `HAClient` class. `HAClient` derives from `Client` and offers the same methods, but also adds protection against network, server, and client outages.

Using `HAClient` allows applications to automatically:

- Recover from temporary disconnects between client and server.
- Failover from one server to another when a server becomes unavailable.

Because the `HAClient` automatically manages failover and reconnection, 60East recommends using the `HAClient` for applications that need to:

- Ensure no messages are lost or duplicated after a reconnect or failover.
- Persist messages and bookmarks on disk for protection against client failure.

You can choose how your application uses `HAClient` features. For example, you might need automatic reconnection, but have no need to resume subscriptions or republish messages. The high availability behavior in `HAClient` is provided by implementations of defined interfaces. You can combine different implementations provided by 60East to meet your needs, and implement those interfaces to provide your own policies.

Some of these features require specific configuration settings on your AMPS instance(s). This chapter mentions these features and describes how to use them from the AMPS Java client. You can find full documentation for these settings and server features in the *User Guide*.

9.1. Reconnection with HAClient

The most important difference between `Client` and `HAClient` is that `HAClient` automatically provides a reconnect handler.

This description provides a high-level framework for understanding the components involved in failover with the `HAClient`. The components are described in more detail in the following sections.

The `HAClient` reconnect handler performs the following steps when reconnecting:

- Calls the `ServerChooser` to determine the next URI to connect to and the authenticator to use for that connection.

  If the connection fails, calls `get_error` on the `ServerChooser` to get a description of the failure, sends an exception to the exception listener, and stops the reconnection process.

- Calls the `DelayStrategy` to determine how long to wait before attempting to reconnect, and waits for that period of time.
- Connects to the AMPS server. If the connection fails, calls `report_failure` on the `ServerChooser` and begins the process again.
- Logs on to the AMPS server. If the connection fails, calls `report_failure` on the `ServerChooser` and begins the process again.
- Calls `report_success` on the `ServerChooser`.

- Receives the bookmark for the last message that the server has persisted. Discards any older messages from the `PublishStore`.
- Republishes any messages in the `PublishStore` that have not been persisted by the server.
- Re-establishes subscriptions using the `SubscriptionManager` for the client. For bookmark subscriptions, the reconnect handler uses the `BookmarkStore` for the client to determine the most recent bookmark, and resubscribes with that bookmark. For subscriptions that do not use a bookmark, the `SubscriptionManager` simply re-enters the subscription, meaning that it is entered at the point at which the `HAClient` reconnects.

The `ServerChooser`, `DelayStrategy`, `PublishStore`, `SubscriptionManager`, and `BookmarkStore` are all extension points for the `HAClient`. You can adapt the failover and recovery behavior by setting a different object for the behavior you want to customize on the `HAClient` or by providing your own implementation.

For example, the convenience methods in the previous section customize the behavior of the `PublishStore` and `BookmarkStore` by providing either memory-backed or file-backed stores.

9.2. Choosing Store Durability

Use the `HAClient` class to create a highly-available connection to one or more AMPS instances. `HAClient` derives from `Client` and offers the same methods, but also adds protection against network, server, and client outages. Most code written with `Client` will also work with `HAClient`, and major differences involve constructing and connecting the `HAClient`.

The `HAClient` provides recovery after disconnection using *Stores*. As the name implies, *stores* hold information about the state of the client. There are two types of store:

- A bookmark store tracks received messages, and is used to resume subscriptions.
- A publish store tracks published messages, and is used to ensure that messages are persisted in AMPS.

The AMPS client provides a memory-backed version of each store and a file-backed version of each store. The store interface is public, and an application can create and provide a custom store as necessary. An `HAClient` can use either a memory backed store or a file backed store for protection. Each method provides resilience to different failures:

- Memory-backed stores provide recovery disconnection from AMPS by storing messages and bookmarks in your process' address space. This is the highest performance option for working with AMPS in a highly available manner. The trade-off with this method is there is no protection from a crash or failure of your client application. If your application is terminated prematurely or, if the application terminates at the same time as an AMPS instance failure or network outage, then messages may be lost or duplicated.
- File-backed stores provide recovery after client failure and disconnection from AMPS by storing messages and bookmarks on disk. To use this protection method, the `create_file_backed` method requests additional arguments for the two files that will be used for both bookmark storage and message storage. If these files exist and are non-empty (as they would be after a client application is restarted), the `HAClient` loads their contents and ensures synchronization with the AMPS server once connected. The performance of this option depends heavily on the speed of the device on which these files are placed. When the files do not exist (as they would the first time a client starts on a given system), the `HAClient` creates and initializes the files, and in this case the client does not have a point at which to resume the subscription or messages to republish.

The store interface is public, and an application can create and provide a custom store as necessary. While clients provide convenience methods for creating file-backed and memory-backed `HAClient` objects with the appropriate

stores, you can also create and set the stores in your application code. For the AMPS Python Client, stores are implemented in C++. You can implement stores using C++, and use the technique described in Section 12.2, Using the C++ Client, to set the store on the client.

The `HAClient` provides convenience methods for creating clients and setting stores. You can also construct an `HAClient` and set the store implementations you choose.

In this example, we create several clients. The first client uses memory stores for both bookmarks and publishes. The second client uses files for both bookmarks and publishes. The third client uses a file for bookmarks. The third client does not set a store for publishes, which means that AMPS provides the default store (and no outgoing messages are stored). The final client does not specify any stores, and so has no persistence for published messages or bookmark subscriptions, but can take advantage of the automatic failover and reconnection in the `HAClient`.

```
# Memory publish store, memory bookmark store
memoryClient = AMPS.HAClient("lessImportantMessages")

# File-backed publish store, file-backed bookmark store
diskClient = AMPS.HAClient("moreImportantMessages",
          "/mnt/fastDisk/moreImportantMessages.outgoing",
          "/mnt/fastDisk/moreImportantMessages.incoming")

# No-op publish store, file-backed bookmark store
subscriberClient = AMPS.HAClient("subscriber", no_store=True)
subscriberClient.set_bookmark_store(     \
            AMPS.MMapBookmarkStore("/mnt/fastdisk/bookmark.store"))

# No-op publish store, no-op bookmark store
# Failover behavior only.
streamReader = AMPS.HAClient("streamReader",no_store=True)
```

Example 9.1. HAClient creation examples

9.3. Connections and the ServerChooser

Unlike `Client`, the `HAClient` attempts to keep itself connected to an AMPS instance at all times, by automatically reconnecting or failing over when it detects that the client is disconnected. When you are using the `Client` directly, your disconnect handler usually takes care of reconnection. `HAClient`, on the other hand, provides a disconnect handler that automatically reconnects to the current server or to the next available server.

To inform the `HAClient` of the addresses of the AMPS instances in your system, you pass a `ServerChooser` instance to the `HAClient`. `ServerChooser` acts as a smart enumerator over the servers available: `HAClient` calls `ServerChooser` methods to inquire about what server should be connected, and calls methods to indicate whether a given server succeeded or failed.

The AMPS Python Client provides a simple implementation of `ServerChooser`,called `DefaultServerChooser`, that provides very simple logic for reconnecting. This server chooser is most suitable for basic testing, or in cases where an application should simply rotate through a list of servers. For most applications, you implement the `ServerChooser` interface yourself for more advanced logic, such as choosing a backup server based on your network topology, or limiting the number of times your application should try to reconnect to a given address.

To connect to AMPS, you provide a `ServerChooser` to `HAClient` and then call `connect_and_logon()` to create the first connection:

```
memoryClient = AMPS.HAClient("myClient")

# primary.amps.xyz.com is the primary AMPS instance, and
# secondary.amps.xyz.com is the secondary
chooser = AMPS.DefaultServerChooser()
chooser.add("tcp://primary.amps.xyz.com:12345/fix")
chooser.add("tcp://secondary.amps.xyz.com:12345/fix")
memoryClient.set_server_chooser(chooser)
memoryClient.connect_and_logon()
...
myClient.disconnect()
```

Example 9.2. HAClient logon

Similar to `Client`, `HAClient` remains connected to the server until `disconnect()` is called. Unlike `Client`, `HAClient` automatically attempts to reconnect to your server if it detects a disconnect and, if that server cannot be connected, fails over to the next server provided by the `ServerChooser`. In this example, the call to `connectAndLogon()` attempts to connect and login to `primary.amps.xyz.com`, and returns if that is successful. If it cannot connect, it tries `secondary.amps.xyz.com`, and continues trying servers from the `ServerChooser` until a connection is established. Likewise, if it detects a disconnection while the client is in use, then `HAClient` attempts to reconnect to the server with which it was most recently connected; if that is not possible, then it moves on to the next server provided by the `ServerChooser`.

The default `ServerChooser` simply provides the next URL in the sequence. This strategy works for many applications. If you need a different strategy, you can implement your own logic for failover by creating a class derived from `ServerChooser`.

Setting a Reconnect Delay and Timeout

You can control the amount of time between reconnection attempts and set a total amount of time for the `HAClient` to attempt to reconnect.

The AMPS Python client includes a method for setting a delay strategy on a client, `set_reconnect_delay_strategy`. This method accepts an instance of any type that provides the methods `get_connect_wait_duration` and `reset`, as described in the API documentation.

While you can easily implement your own delay strategy, the client also provides two delay strategies:

- `FixedDelayStrategy` provides the same delay each time the `HAClient` tries to reconnect.
- `ExponentialDelayStrategy` provides an exponential backoff until a connection attempt succeeds.

To use either of these classes, you simply create an instance, set the appropriate parameters, and install that instance as the delay strategy for the `HAClient`. For example, the following code sets up a reconnect delay that starts at 200ms and increases the delay by 1.5 times after each failure. The strategy allows a maximum delay between connection attempts of 5 seconds, and will not retry longer than 60 seconds.

```
theClient = AMPS.HAClient("myClient")

theClient.set_reconnect_delay_strategy(      \
```

```
  AMPS.ExponentialDelayStrategy(            \
                initial_delay=200,          \
                maximum_delay=5000,         \
                backoff_exponent=1.5,       \
                maximum_retry_time=60000) \
              )
```

Implementing a Server Chooser

As described above, you provide the HAClient with connection strings to one or more AMPS servers using a ServerChooser. The purpose of a ServerChooser is to provide information to the HAClient. A ServerChooser does not manage the reconnection process, and should not call methods on the HAClient.

A ServerChooser has two required responsibilities to the HAClient:

- Tells the HAClient the connection string for the server to connect to. If there are no servers, or the ServerChooser wants the connection to fail, the ServerChooser returns an empty string.

  To provide this information, the ServerChooser implements the get_current_uri() method.

- Provides an Authenticator for the current connection string. This is especially important for installations where different servers require different credentials or authentication tokens must be reset after each connection attempt.

  To provide the authenticator, the ServerChooser implements the get_current_authenticator() method.

The HAClient calls the get_current_uri() and get_current_authenticator() methods each time it needs to make a connection.

Each time a connection succeeds, the HAClient calls the report_success() method of the ServerChooser. Each time a connection fails, the HAClient calls the report_failure() method of the ServerChooser. The HAClient does not require the ServerChooser to take any particular action when it calls these methods. These methods are provided for the HAClient to do internal maintenance, logging, or record keeping. For example, an HAClient might keep a list of available URIs with a current failure count, and skip over URIs that have failed more than 5 consecutive times until all URIs in the list have failed more than 5 consecutive times.

When the ServerChooser returns an empty string from get_current_uri(), indicating that no servers are available for connection, the HAClient calls the get_error() method on the ServerChooser, if one is provided, and includes the string returned by get_error() in the generated exception.

9.4. Heartbeats and Failure Detection

Use of the HAClient allows your application to quickly recover from detected connection failures. By default, connection failure detection occurs when AMPS receives an operating system error on the connection. This system may result in unpredictable delays in detecting a connection failure on the client, particularly when failures in network routing hardware occur, and the client primarily acts as a subscriber.

The heartbeat feature of the AMPS client allows connection failure to be detected quickly. Heartbeats ensure that regular messages are sent between the AMPS client and server on a predictable schedule. The AMPS client and

server both assume disconnection has occurred if these regular heartbeats cease, ensuring disconnection is detected in a timely manner. To use heartbeating, call the `set_heartbeat` method on `Client` or `HAClient`:

```
memoryClient = AMPS.HAClient("importantStuff")
...
memoryClient.set_heartbeat(3)
memoryClient.connect_and_logon()
...
```

Method `set_heartbeat` takes one parameter: the heartbeat interval. The heartbeat interval specifies the periodicity of heartbeat messages sent by the server: the value `3` indicates messages are sent on a three-second interval. If the client receives no messages in a six-second window (two heartbeat intervals), the connection is assumed to be dead, and the `HAClient` attempts reconnection. An additional variant of `set_heartbeat` allows the idle period to be set to a value other than two heartbeat intervals.

Notice that, for `HAClient`, `setHeartbeat` must be called before the client is connected. For `Client`, `setHeartbeat` must be called after the client is connected.

Heartbeats are serviced on the receive thread created by the AMPS client. Your application must not block the receive thread for longer than the heartbeat interval, or the application is subject to being disconnected.

9.5. Considerations for Publishers

Publishing with an `HAClient` is nearly identical to regular publishing; you simply call the `publish()` method with your message's topic and data. The AMPS client sends the message to AMPS, and then returns from the `publish()` call. For maximum performance, the client does not wait for the AMPS server to acknowledge that the message has been received.

When an `HAClient` sets a publish store, the publish store retains a copy of each outgoing message and requests that AMPS acknowledge that the message has been persisted. The AMPS server acknowledges messages back to the publisher. Acknowledgements can be delivered for multiple messages at periodic intervals (for topics recorded in the trnasaction log) or after each message (for topics that are not recorded in the transaction log). When an acknowledgement for a message is received, the HAClient removes that message from the bookmark store. When a connection to a server is made, the `HAClient`automatically determines which messages from the publish store (if any) the server has not processed, and replays those messages to the server once the connection is established.

For reliable publishers, the application must choose how best to handle application shutdown. For example, it is possible for the network to fail immediately after the publisher sends the message, while the message is still in transit. In this case, the publisher has sent the message, but the server has not processed it and acknowledged it. During normal operation, the `HAClient` will automatically connect and retry the message. On shutdown, however, the application must decide whether to wait for messages to be acknowledged, or whether to exit.

Publish store implementations provide an `unpersisted_count()` method that reports the number of messages that have not yet been acknowledged by the AMPS server. When the `unpersisted_count()` reaches `0`, there are no unpersisted messages in the local publish store.

For the highest level of safety, an application can wait until the `unpersisted_count()` reaches `0`, which indicates that all of the messages have been persisted to the instance that the application is connected to, and the synchronous replication destinations configured for that instance. When a synchronous replication destination goes

offline, this approach will cause the publisher to wait to exit until the destination comes back online or until the destination is downgraded to asynchronous replication.

For applications that are shut down periodically for short periods of time (for example, applications that are only offline during a weekly maintenance window), another approach is to use the `publish_flush()` method to ensure that messages are delivered to AMPS, and then rely on the connection logic to replay messages as necessary when the application restarts.

For example, the following code flushes messages to AMPS, then warns if not all messages have been acknowledged:

```
client = AMPS.HAClient("ha-publisher",
         "/mnt/fastDisk/moreImportantMessages.outgoing",
         "/mnt/fastDisk/moreImportantMessages.incoming")
...
client.connect_and_logon()

# Publish messages
...


# We think we are done, but the server may not
# have received or acknowledged all messages yet.

# Wait for the server to have received all messages.
# The program could also specify a timeout in this
# command to avoid blocking forever if the network
# is down or all servers are offline.

client.publish_flush()

# Print warning to the console if messages have
# been published but not yet acknowledged as
# persisted

if (client.get_unpersisted_count() > 0):
    print "all messages have been published, " \
        + " but not all have been persisted"

client.disconnect()
```

Example 9.3. HA Publisher

In this example, the client sends each message immediately when `publish()` is called. If AMPS becomes unavailable between the final `publish()` and the `disconnect()`, or one of the servers that the AMPS instance replicates to is offline, the client may not have received a persisted acknowledgement for all of the published messages. For example, if a message has not yet been persisted by all of the servers in the replication fabric that are connected with synchronous replication, AMPS will not have acknowledged the message.

Before shutting down the client, the code does two This code first flushes messages to the server to ensure that all messages have been delivered to AMPS.

The code next checks to see if all of the messages in the publish store have been acknowledged as persisted by AMPS. If the messages have not been acknowledged, they will remain in the publish store file and will be published to AMPS, if necessary, the next time the application connects. An application may choose to loop until

`get_unpersisted_count()` returns `0`, or (as we do in this case) simply warn that AMPS has not confirmed that the messages are fully persisted. The behavior you choose in your application should be consistent with the high-availability guarantees your application needs to provide.

AMPS uses the name of the `HAClient` to determine the origin of messages. For the AMPS server to correctly identify duplicate messages, each instance of an application that publishes messages must use a distinct name. That name must be consistent across different runs of the application.

If your application crashes or is terminated, some published messages may not have been persisted in the AMPS server. If you use the file-based store—in other words, if you provide file names for persistent storage when you create the `HAClient`—the `HAClient` will recover the messages, and once logged on, will correlate the message store to what the AMPS server has received, re-publishing any missing messages. This occurs automatically when `HAClient` connects, without any explicit consideration in your code, other than ensuring that the same file name is used to create the `HAClient` if recovery is desired.

AMPS provides persisted acknowledgement messages for topics that do not have a transaction log enabled. However, the level of durability provided for topics with no transaction log is minimal. Learn more about transaction logs in the *User Guide*.

9.6. Considerations for Subscribers

`HAClient` provides two important features for applications that subscribe to one or more topics: re-subscription, and a bookmark store to track the correct point at which to resume a bookmark subscription.

Resubscription With Asynchronous Message Processing

Any asynchronous subscription placed using an `HAClient` is automatically reinstated after a disconnect or a failover. These subscriptions are placed in an in-memory `SubscriptionManager`, which is created automatically when the `HAClient` is instantiated. Most applications will use this built-in subscription manager, but for applications that create a varying number of subscriptions, you may wish to implement `SubscriptionManager` to store subscriptions in a more durable place. Note that these subscriptions contain no message data, but rather simply contain the parameters of the subscription itself (for instance, the command, topic, message handler, options, and filter).

When a re-subscription occurs, the AMPS Python Client re-executes the command as originally submitted, including the original topic, options, and so on. AMPS sends the subscriber any messages for the specified topic (or topic expression) that are published after the subscription is placed. For a `sow_and_subscribe` command, this means that the client reissues the full command, including the SOW query as well as the subscription.

Resubscription With Synchronous Message Processing

The `HAClient` (starting with the AMPS Python Client version 4.3.1.1) does not track synchronous message processing subscriptions in the `SubscriptionManager`. The reason for this is to preserve the iterator semantics. That is, once the `MessageStream` indicates that there are no more elements in the stream, it does not suddenly produce more elements.

To resubscribe when the HAClient fails over, you can simply reissue the subscription. For example, the snippet below re-issues the subscribe command when the message stream ends:

```
still_need_to_process = True

while still_need_to_process:
   for message in client.subscribe("messages"):
       # process messages

       # check condition on still_need_to_process
       if still_need_to_process == False: break

    # Exiting the for loop is the end of stream.
    # For a subscribe, this likely means that the
    # client has disconnected.

# Exiting the while loop means processing is done
```

Bookmark Stores

In cases where it is critical not to miss a single message, it is important to be able to resume a subscription at the exact point that a failure occurred. In this case, simply recreating a subscription isn't sufficient. Even though the subscription is recreated, the subscriber may have been disconnected at precisely the wrong time, and will not see the message.

To ensure delivery of every message from a topic or set of topics, the AMPS HAClient includes a BookmarkStore that, combined with the bookmark subscription and transaction log functionality in the AMPS server, ensures that clients receive any messages that might have been missed. The client stores the bookmark associated with each message received, and tracks whether the application has processed that message; if a disconnect occurs, the client uses the BookmarkStore to determine the correct resubscription point, and sends that bookmark to AMPS when it re-subscribes. AMPS then replays messages from its transaction log from the point after the specified bookmark, thus ensuring the client is completely up-to-date.

HAClient helps you to take advantage of this bookmark mechanism through the BookmarkStore interface and bookmarkSubscribe() method on Client. When you create subscriptions with bookmarkSubscribe(), whenever a disconnection or failover occurs, your application automatically resubscribes to the message after the last message it processed. HAClients created by createFileBacked() additionally store these bookmarks on disk, so that the application can restart with the appropriate message if the client application fails and restarts.

To take advantage of bookmark subscriptions, do the following:

- Ensure the topic(s) to be subscribed are included in a transaction log. See the *User Guide* for information on how to specify the contents of a transaction log.
- Use bookmark_subscribe() instead of subscribe() when creating a subscription(), and decide how the application will manage subscription identifiers (SubIds).
- Use the discard() method in message handlers to indicate when a message has been fully processed by the application.

The following example creates a bookmark subscription against a transaction-logged topic, and fully processes each message as soon as it is delivered:

```
def on_message_printer(message):
   print message.getData()

...

client = AMPS.HAClient(
    "aClient",
    "/logs/aClient.publishLog",
    "/logs/aClient.subscribeLog")

...

client.execute_async(                          \
    Command("subscribe")                       \
        .set_topic("myTopic")                  \
        .set_bookmark(Bookmarks.MOST_RECENT) \
        .set_sub_id("MySubID"))
```

Example 9.4. HAClient Subscription

In this example, the client is a file-backed client, meaning that arriving bookmarks will be stored in a file (`aClient.subscribeLog`). Storing these bookmarks in a file allows the application to restart the subscription from the last message processed, in the event of either server or client failure.

For optimum performance, it is critical to discard every message once its processing is complete. If a message is never discarded, it remains in the bookmark store. During re-subscription, `HAClient` always restarts the bookmark subscription with the oldest undiscarded message, and then filters out any more recent messages that have been discarded. If an old message remains in the store, but is no longer important for the application's functioning, then the client and the AMPS server will incur unnecessary network, disk, and CPU activity.

The fourth parameter, `sub_id`, specifies an identifier to be used for this subscription. Passing `None` causes `HAClient` to generate one and return it, like most other `Client` functions. However, if you wish to resume a subscription from a previous point after the application has terminated and restarted, the application must pass the same subscription ID as during its previous run. Passing a different subscription ID bypasses any recovery mechanisms, creating an entirely new subscription. When you use an existing subscription ID, the `HAClient` locates the last-used bookmark for that subscription in the local store, and attempts to re-subscribe from that point.

- `Client.Bookmarks.NOW` specifies that the subscription should begin from the moment the server receives the subscription request. This results in the same messages being delivered as if you had invoked `subscribe()` instead, except that the messages will be accompanied by bookmarks. This is also the behavior that results if you supply an invalid bookmark.
- `Client.Bookmarks.EPOCH` specifies that the subscription should begin from the beginning of the AMPS transaction log (that is, the first entry in the oldest journal file for the transaction log).
- `Client.Bookmarks.MOST_RECENT` specifies that the subscription should begin from the last-used message in the associated `BookmarkStore`. Alternatively, if this subscription has not been seen before, it instructs the subscription to begin with `EPOCH`. This is the most common value for this parameter, and is the value used in the preceding example. By using `MOST_RECENT`, the application automatically resumes from wherever the subscription left off, taking into account any messages that have already been processed and discarded.

When the `HAClient` re-subscribes after a disconnection and reconnection, it always uses `MOST_RECENT`, ensuring that the continued subscription always begins from the last message used before the disconnect, so that no messages are missed.

9.7. Conclusion

With only a few changes, most AMPS applications can take advantage of the `HAClient` and associated classes to become more highly-available and resilient. Using the `PublishStore`, publishers can ensure that every message published has actually been persisted by AMPS. Using `BookmarkStore`, subscribers can make sure that there are no gaps or duplicates in the messages received. `HAClient` makes both kinds of applications more resilient to network and server outages and temporary issues, and, by using the filebased `HAClient`, clients can recover their state after an unexpected termination or crash. Though `HAClient` provides useful defaults for the `Store`, `BookmarkStore`, `SubscriptionManager`, and `ServerChooser`, you can customize any or all of these to the specific needs of your application and architecture.

Chapter 10. AMPS Programming: Working with Commands

The AMPS clients provide named convenience methods for core AMPS functionality. These named methods work by creating messages and sending those messages to AMPS. All communication with AMPS occurs through messages.

You can use the `Command` object to customize the messages that AMPS sends. This is useful for more advanced scenarios where you need precise control over the message, or in cases where you need to use an earlier version of the client to communicate with a more recent version of AMPS, or in cases where a named method is not available.

10.1. Understanding AMPS Messages

AMPS messages are represented in the client as AMPS.Message objects. The Message object is generic, and can represent any type of AMPS message, including both outgoing and incoming messages. This section includes a brief overview of elements common to AMPS command message. Full details of commands to AMPS are provided in the *AMPS Command Reference Guide*.

All AMPS command messages contain the following elements:

- **Command**. The *command* tells AMPS how to interpret the message. Without a command, AMPS will reject the message. Examples of commands include `publish`, `subscribe`, and `sow`.
- **CommandId**. The *command id*, together with the name of the client, uniquely identifies a command to AMPS. The command ID can be used later on to refer to the command or the results of the command. For example, the command id for a `subscribe` message becomes the identifier for the subscription. The AMPS client provides a command id when the command requires one and no command id is set.

Most AMPS messages contain the following fields:

- **Topic**. The *topic* that the command applies to, or a regular expression that identifies a set of topics that the command applies to. For most commands, the topic is required. Commands such as `logon`, `start_timer`, and `stop_timer` do not apply to a specific topic, and do not need this field.
- **Ack Type**. The *ack type* tells AMPS how to acknowledge the message to the client. Each command has a default acknowledgement type that AMPS uses if no other type is provided.
- **Options**. The `options` are a comma-separated list of options that affect how AMPS processes and responds to the message.

Beyond these fields, different commands include fields that are relevant to that particular command. For example, SOW queries, subscriptions, and some forms of SOW deletes accept the **Filter** field, which specifies the filter to apply to the subscription or query. As another example, publish commands accept the **Expiration** field, which sets the SOW expiration for the message.

For full details on the options available for each command and the acknowledgement messages returned by AMPS, see the *AMPS Command Reference Guide*.

10.2. Creating and Populating the Command

To create a command, you simply construct a command object of the appropriate type:

```
command = AMPS.Command("sow")
```

Once created, you set the appropriate fields on the message. For example, the following code creates a sow command, setting the command, topic, and filter for the query:

```
command = AMPS.Command("sow")                                    \
            .set_topic("messages-sow")                           \
            .set_filter("/id > 20")
```

When sent to AMPS using the `execute()` method, AMPS performs a SOW query from the topic `messages-sow` using a filter of `/id > 20`. The results of sending this message to AMPS are no different than using the form of the `sow` method that sets these fields.

10.3. Using execute

Once you've created a message, use the `execute` method to send the message to AMPS. One form of the `execute` method returns a `MessageStream` that you can use from the calling thread to process responses from AMPS. The other form, `execute_async`, method sends the message to AMPS,waits for a `processed` acknowledgement, then returns. Messages are processed on the client background thread.

For example, the following snippet sends the command created above:

```
client.execute(command)
```

This returns a `MessageStream` identical to the `MessageStream` returned by the equivalent `client.sow()` method.

You can also provide a message handler to receive acknowledgements, statistics, or the results of subscriptions and SOW queries. The AMPS client maintains a background thread that receives and processes incoming messages. The call to `execute_async` returns on the main thread as soon as AMPS acknowledges the command as having been processed, and messages are received and processed on the background thread:

```
def handleMessages(m):
   print "%s : %s" % (m.get_ack_type(), m.get_reason())
   # other message handling here

client.execute_async(command, handleMessages)
```

While this message handler simply prints the ack type and reason for sample purposes, message handlers in production applications are typically designed with a specific purpose. For example, your message handler may fill a work queue, or check for success and throw an exception if the command failed.

Notice that the `publish` command does not provide typically return results other than acknowledgement messages. To send a `publish` command, use the `execute_sync()` method, providing `None` for the message handler:

```
client.execute_async(publishCmd, None);
```

10.4. Command Cookbook

This section is a quick guide to commonly used AMPS commands. For the full range of options on AMPS commands, see the *AMPS Command Reference*.

Publishing

This section presents common recipes for publishing to a topic in AMPS using the `Command` or `Message` interfaces. This section provides information on how to configure the request to AMPS. You can adapt this information to your application and the specific interface you are using.

The AMPS server does not return a stream of messages in response to a `publish` command.

AMPS `publish` commands do not return a stream of messages. A publish command must be used with asynchronous message processing, while passing an empty message handler.

Basic Publish

In its simplest form, a subscription needs only the topic to publish to and the data to publish. The AMPS client automatically constructs the necessary AMPS headers and formats the full `publish` command.

In many cases, a publisher only needs to use the basic publish command.

Table 10.1. Basic Publish

Header	Comment
`Topic`	Sets the topic to publish to. The topic specified must be a literal topic name. Regular expression characters in the topic name are not interpreted. Some topics in AMPS, such as views and conflated topics, cannot be published to directly. Instead, a publisher must publish to the underlying topics.
`Data`	The data to publish to the topic. The AMPS client does not interpret, escape, or validate this data: the data is provided to the server verbatim.

Publish With CorrelationId

AMPS provides publishers with a header field that can be used to contain arbitrary data, the `CorrelationId`.

Table 10.2. Publish With CorrelationId

Header	Comment
`Topic`	Sets the topic to publish to. The topic specified must be a literal topic name. Regular expression characters in the topic name are not interpreted. Some topics in AMPS, such as views and conflated topics, cannot be published to directly. Instead, a publisher must publish to the underlying topics.
`Data`	The data to publish to the topic. The AMPS client does not interpret, escape, or validate this data: the data is provided to the server verbatim.

Header	Comment
`CorrelationId`	The `CorrelationId` to provide on the message. AMPS provides the `CorrelationId` to subscribers. The `CorrelationId` has no significance for AMPS. The `CorrelationId` may only contain characters that are valid in base-64 encoding.

Publish With Explicit SOW Key

When publishing to a SOW topic that is configured to require an explicit SOW key, the publisher needs to set the SowKey header on the message.

Table 10.3. Publish with Explicit SOW Key

Header	Comment
`Topic`	Sets the topic to publish to. The topic specified must be a literal topic name. Regular expression characters in the topic name are not interpreted. Some topics in AMPS, such as views and conflated topics, cannot be published to directly. Instead, a publisher must publish to the underlying topics.
`Data`	The data to publish to the topic. The AMPS client does not interpret, escape, or validate this data: the data is provided to the server verbatim.
`SowKey`	The SOW Key to use for this message. This header is only supported for publishes to a topic that requires an explicit SOW Key.

Command Cookbook: Subscribing

This section presents common recipes for subscribing to a topic in AMPS using the `Command` or `Message` interfaces. This section provides information on how to configure the request to AMPS. You can adapt this information to your application and the specific interface you are using.

Basic Subscription

In its simplest form, a subscription needs only the topic to subscribe to.

Table 10.4. Basic Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. All messages from the topic will be delivered on this subscription. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.

Basic Subscription With Options

In its simplest form, a subscription needs only the topic to subscribe to. To add options to the subscription, set the `Options` header on the `Command`.

Table 10.5. Basic Subscription with Options

Header	Comment
`Topic`	Sets the topic to subscribe to. All messages from the topic will be delivered on this subscription. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Options`	A comma-delimited set of options for this command. See the *AMPS Command Reference* for a description of supported options.

Content Filtered Subscription

To provide a content filter on a subscription, set the `Filter` property on the command. The *AMPS User Guide* provides details on the filter syntax.

Table 10.6. Content Filtered Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.
`Filter`	Sets the content filter to be applied to the subscription. Only messages that match the content filter will be provided to the subscription.

Conflated Subscription to a SOW Topic

To request conflation on a subscription, set the `Options` property on the command to specify the conflation interval. When the topic has a SOW (including a view or a conflated topic), there is no need to provide a `conflation_key`.

Table 10.7. Conflated Subscription to a SOW topic

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.
`Options`	A comma-separated list of options for the command. Set the conflation interval in the options. For example, to set a conflation interval of 250 milliseconds, provide the following option: `conflation=250ms` To set the conflation interval to 1 minute, provide an option of:

Header	Comment
	`conflation=1m`

Conflated Subscription to a Topic With No SOW

To request conflation on a subscription, set the `Options` property on the command to specify the conflation interval. When the topic does not have a SOW, you must provide a `conflation_key`.

Table 10.8. Conflated Subscription to a SOW topic

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.

`Options`	A comma-separated list of options for the command. Set the conflation interval and the fields that determine a unique message in the options.
	
	For example, to set a conflation interval of 250 milliseconds for messages that have the same value for `/id`, provide the following option:
	
	`conflation=250ms,conflation_key=[/id]`
	
	To set the conflation interval to 1 minute for messages are the same as determined by a combination of `/customerId` and `/issueNumber` provide an option of:
	
	`conflation=1m, conflation_key=[/customerId,/ issueNumber]`

Bookmark Subscription

To create a bookmark subscription, set the `Bookmark` property on the command. The value of this property can be either a specific bookmark, a timestamp, or one of the client-provided constants. The *AMPS User Guide* provides details on creating timestamps. Notice that the `MOST_RECENT` constant tells the AMPS client to find the appropriate message in the client bookmark store and begin the subscription at that point. In this case, the client sends that bookmark value to AMPS. The `Bookmark` option is only supported for topics that are recorded in an AMPS transaction log.

Table 10.9. Bookmark Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.
`Bookmark`	Sets the point in the transaction log at which the subscription will begin. The bookmark provided can be a specific

Header	Comment
	AMPS bookmark, a timestamp, or one of the client-provided constants. AMPS also accepts a comma-delimited list of bookmarks. In this case, AMPS begins the subscription from whichever of the bookmarks is earliest in the transaction log.

Rate Controlled Bookmark Subscription

To create a bookmark subscription, set the `Bookmark` property on the command. The value of this property can be either a specific bookmark, a timestamp, or one of the client-provided constants. The *AMPS User Guide* provides details on creating timestamps. Notice that the `MOST_RECENT` constant tells the AMPS client to find the appropriate message in the client bookmark store and begin the subscription at that point. In this case, the client sends that bookmark value to AMPS. The `Bookmark` option is only supported for topics that are recorded in an AMPS transaction log.

Table 10.10. Bookmark Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.
`Bookmark`	Sets the point in the transaction log at which the subscription will begin. The bookmark provided can be a specific AMPS bookmark, a timestamp, or one of the client-provided constants. AMPS also accepts a comma-delimited list of bookmarks. In this case, AMPS begins the subscription from whichever of the bookmarks is earliest in the transaction log.
`Options`	A comma-separated list of options for the command. To control the rate at which AMPS delivers messages, the options for the command must include a rate specifier. For example, to specify a limit of 750 messages per second, include `rate=750` in the options string.

Bookmark Subscription With Content Filter

To create a bookmark subscription, set the `Bookmark` property on the command. The property can be either a specific bookmark, a timestamp, or one of the client-provided constants. The *AMPS User Guide* provides details on creating timestamps. Notice that the `MOST_RECENT` constant tells the AMPS client to find the appropriate message in the client bookmark store and begin the subscription at that point. In this case, the client sends that bookmark value to AMPS.

To add a filter to a bookmark subscription, set the `Filter` property on the command. The *AMPS User Guide* provides details on the filter syntax.

Table 10.11. Bookmark Subscription With Content Filter

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.
`Bookmark`	Sets the point in the transaction log at which the subscription will begin. The bookmark provided can be a specific AMPS bookmark, a timestamp, or one of the client-provided constants. AMPS also accepts a comma-delimited list of bookmarks. In this case, AMPS begins the subscription from whichever of the bookmarks is earliest in the transaction log.
`Filter`	Sets the content filter to be applied to the subscription. Only messages that match the content filter will be provided to the subscription.

Pausing a Bookmark Subscription

To pause a bookmark subscription, you must provide the subscription ID and the `pause` option on a subscribe command.

Table 10.12. Pause a Bookmark Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.
`SubId`	A comma-delimited list of subscription IDs to pause.
`Options`	A comma-delimited list of options for the command. To pause a subscription, the options must include `pause`.

Resuming a Bookmark Subscription

To resume a bookmark subscription, you must provide the subscription ID and the `resume` option on a subscribe command.

Table 10.13. Resume a Bookmark Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.
`SubId`	A comma-delimited list of subscription IDs to resume.
`Options`	A comma-delimited list of options for the command. To resume a subscription, the options must include `resume`.

Replacing the Filter on a Subscription

To replace the content filter on a subscription, provide the `SubId` of the subscription to be replaced, add the `replace` option, and set the `Filter` property on the command with the new filter. The *AMPS User Guide* provides details on the filter syntax.

Table 10.14. Replacing the Filter on a Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.
`SubId`	The identifier for the subscription to update. The `SubId` is the `CommandId` for the original `subscribe` command.
`Options`	A comma-separated list of options. To replace the filter on a subscription, include `replace` in the list of options.
`Filter`	Sets the content filter to be applied to the subscription. Only messages that match the content filter will be provided to the subscription.

Subscribing to a Queue and Requesting a max_backlog

To subscribe to a queue and request a `max_backlog` greater than `1`, use the `Options` field of the `subscribe` command to set the requested `max_backlog`.

Table 10.15. Requesting a max_backlog

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.

`Options`	A comma-separated list of options. To request a value for the max_backlog, pass the value in the options as follows:
	
	`max_backlog=NN`
	
	For example, to request a max backlog of 7, your application would pass the following option:
	
	`max_backlog=7`

SOW Query

This section presents common recipes for querying a SOW topic in AMPS using the `Command` or `Message` interfaces. This section provides information on how to configure the request to AMPS. You can adapt this information to your application and the specific interface you are using.

Basic SOW Query

In its simplest form, a SOW query needs only the topic to query.

Table 10.16. Basic SOW Query

Header	Comment
`Topic`	Sets the topic to query. The SOW query returns all messages in the SOW. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.

Basic SOW With Options

In its simplest form, a SOW needs only the topic to subscribe to. To add options to the subscription, set the `Options` header on the `Command`.

Table 10.17. Basic SOW Query with Options

Header	Comment
`Topic`	Sets the topic to query. The SOW query returns all messages in the SOW. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Options`	A comma-delimited set of options for this command. See the `AMPS Command Reference` for a description of supported options.

SOW Query With Ordered Results

In its simplest form, a SOW needs only the topic to subscribe to. To return the results in a specific order, provide an ordering expression in the `OrderBy` header.

Table 10.18. Basic SOW Query with Ordered Results

Header	Comment
`Topic`	Sets the topic to query. The SOW query returns all messages in the SOW. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.

`OrderBy`	Orders the results returned as specified. Requires a comma-separated list of identifiers of the form:
	
	`/field [ASC | DESC]`
	
	For example, to sort in descending order by `orderDate` so that the most recent orders are first, and ascending order by `customerName` for orders with the same date, you might use a specifier such as:
	
	`/orderDate DESC, /customerName ASC`

Header	Comment
	If no sort order is specified for an identifer, AMPS defaults to ascending order.

SOW Query With TopN Results

In its simplest form, a SOW needs only the topic to subscribe to. To return only a specific number of records, provide the number of records to return in the `TopN` header.

Table 10.19. SOW Query with TopN Results

Header	Comment
`Topic`	Sets the topic to query. The SOW query returns all messages in the SOW. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`TopN`	The maximum number of records to return. AMPS uses the `OrderBy` header to determine the order of the records. If no `OrderBy` header is provided, records are returned in an indeterminate order. In most cases, using an OrderBy header when you use the TopN header will guarantee that you get the records of interest.

`OrderBy`	Orders the results returned as specified. Requires a comma-separated list of identifiers of the form:
	
	`/field [ASC | DESC]`
	
	For example, to sort in descending order by `orderDate` so that the most recent orders are first, and ascending order by `customerName` for orders with the same date, you might use a specifier such as:
	
	`/orderDate DESC, /customerName ASC`
	
	If no sort order is specified for an identifer, AMPS defaults to ascending order.

Content Filtered SOW Query

To provide a content filter on a SOW query, set the `Filter` property on the command. The *AMPS User Guide* provides details on the filter syntax.

Table 10.20. Content Filtered SOW Query Subscription

Header	Comment
`Topic`	Sets the topic to query. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.

Header	Comment
`Filter`	Sets the content filter to be applied to the query. Only messages that match the content filter will be returned in response to the query.

Historical SOW Query

To create a historical SOW query, set the `Bookmark` property on the command. The property can be either a specific bookmark or a timestamp. The *AMPS User Guide* provides details on creating timestamps.

This command is only supported on SOW topics that have `History` enabled.

Table 10.21. Historical SOW Query

Header	Comment
`Topic`	Sets the topic to query. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Bookmark`	Sets the historical point in the SOW at which to query. The query returns the saved state of the records in the SOW as of the point in time specified in this header.

Historical SOW Query With Content Filter

To create a historical SOW query, set the `Bookmark` property on the command. The property can be either a specific bookmark or a timestamp. The *AMPS User Guide* provides details on creating timestamps. To add a filter to the query, set the `Filter` property on the command. The *AMPS User Guide* provides details on the filter syntax.

This command is only supported on SOW topics that have `History` enabled.

Table 10.22. Historical SOW Query With Content Filter

Header	Comment
`Topic`	Sets the topic to query. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Bookmark`	Sets the historical point in the SOW at which to query. The query returns the saved state of the records in the SOW as of the point in time specified in this header.
`Filter`	Sets the content filter to be applied to the query. Only messages that match the content filter will be provided to the query.

SOW Query for Specific Records

AMPS allows a consumer to query for specific records as identified by a set of `SowKeys`. For topics where AMPS assigns the `SowKey`, the SowKey for the record is the AMPS-assigned identifier. For topics configured to require

a user-provided `SowKey`, the `SowKey` for the record is the original key provided when the record was published. The *AMPS User Guide* provides more details on SOW keys.

Table 10.23. SOW Query by SOW Key

Header	Comment
`Topic`	Sets the topic to query. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`SowKeys`	A comma-delimited list of `SowKey` values. AMPS returns only the records specified in this list. For example, a valid format for a list of keys would be: `18530979318172572 02,10402779940201650075,223638799`

SOW Query with Pagination

AMPS allows a consumer to page through records in the SOW using the `top_n` and `skip_n` options. With this approach, the application uses the `top_n` option to limit the number of records returned to a single page worth of records. The application uses the `skip_n` option to set the number of records to skip ahead to get to the page to display, and sets the `OrderBy` header to specify the ordering for the records. For example, if 10 records fit on a page, and the pages are ordered by the `ClientName` field of the records, to display the fourth page, the application would set `top_n` to `10`, `skip_n` to `30` (to skip the first three pages of records), and `OrderBy` to `/ClientName`.

Table 10.24. SOW Query by SOW Key

Header	Comment
`Topic`	Sets the topic to query. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.

`OrderBy`	Orders the results returned as specified. Requires a comma-separated list of identifiers of the form:
	
	`/field [ASC | DESC]`
	
	For example, to sort in descending order by `orderDate` so that the most recent orders are first, and ascending order by `customerName` for orders with the same date, you might use a specifier such as:
	
	`/orderDate DESC, /customerName ASC`
	
	If no sort order is specified for an identifer, AMPS defaults to ascending order.

`Options`	An options string that sets the `top_n` and `skip_n` values for this query. For example, to skip 100 records and return the next 10 records, use an options string such as: `top_n=10,skip_n=100`

SOW and Subscribe

This section presents common recipes for atomic sow and subscribe in AMPS using the `Command` or `Message` interfaces. This section provides information on how to configure the request to AMPS. You can adapt this information to your application and the specific interface you are using.

Basic SOW and Subscribe

In its simplest form, a SOW and Subscribe needs only the topic to subscribe to.

Table 10.25. Basic SOW and Subscribe

Header	Comment
`Topic`	Sets the topic to query and subscribe to. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.

SOW and Subscribe With Options

In its simplest form, a SOW and subscribe command needs only the topic to subscribe to. To add options to the subscription, set the `Options` header on the `Command`.

Table 10.26. Basic SOW and Subscribe with Options

Header	Comment
`Topic`	Sets the topic to query and subscribe to. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Options`	A comma-delimited set of options for this command. See the `AMPS Command Reference` for a full description of supported options. The most common options for this command are: `oof` — Request out of order notifications `timestamp` — Include timestamps on messages

Content Filtered SOW and Subscribe

To provide a content filter on a SOW and Subscribe, set the `Filter` property on the command. The *AMPS User Guide* provides details on the filter syntax.

Table 10.27. Content Filtered SOW and Subscribe

Header	Comment
`Topic`	Sets the topic to query and subscribe to. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.

Header	Comment
`Filter`	Sets the content filter to be applied to the query. Only messages that match the content filter will be returned in response to the query.

Conflated SOW and Subscribe

To request conflation on the subscription for a SOW and subscribe, set the `Options` property on the command to specify the conflation interval. When the topic has a SOW (including a view or a conflated topic), there is no need to provide a `conflation_key`.

Table 10.28. Conflated SOW and Subscribe

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.

`Options`	A comma-separated list of options for the command. Set the conflation interval in the options.
	
	For example, to set a conflation interval of 250 milliseconds, provide the following option:
	
	`conflation=250ms`
	
	To set the conflation interval to 1 minute, provide an option of:
	
	`conflation=1m`

Historical SOW and Subscribe

To create a historical SOW query with a subscription, set the `Bookmark` property on the command. The property can be either a specific bookmark or a timestamp. The *AMPS User Guide* provides details on creating timestamps. This command is only supported on SOW topics that are recorded in an AMPS transaction log. If the `Bookmark` provided is a value other than NOW (`0|1|`), the SOW topic must have `History` enabled.

Table 10.29. Historical SOW and Subscribe

Header	Comment
`Topic`	Sets the topic to query and subscribe to. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Bookmark`	Sets the historical point in the SOW at which to query. The query returns the saved state of the records in the SOW as of the point in time specified in this header.

Historical SOW and Subscribe With Content Filter

To create a historical SOW query with a subscription, set the `Bookmark` property on the command. The property can be either a specific bookmark or a timestamp. The *AMPS User Guide* provides details on creating timestamps. This command is only supported on SOW topics that are recorded in an AMPS transaction log. If the `Bookmark` provided is a value other than NOW (`0|1|`), the SOW topic must have `History` enabled.

Table 10.30. Historical SOW and Subscribe With Content Filter

Header	Comment
`Topic`	Sets the topic to query. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Bookmark`	Sets the historical point in the SOW at which to query. The query returns the saved state of the records in the SOW as of the point in time specified in this header.
`Filter`	Sets the content filter to be applied to the query. Only messages that match the content filter will be provided to the query.

Delta Publishing

This section presents common recipes for publishing to a topic in AMPS using the `Command` or `Message` interfaces. This section provides information on how to configure the request to AMPS. You can adapt this information to your application and the specific interface you are using.

Basic Delta Publish

In its simplest form, a subscription needs only the topic to publish to and the data to publish. The AMPS client automatically constructs the necessary AMPS headers and formats the full `delta_publish` command.

In many cases, a publisher only needs to use the basic delta publish command.

Table 10.31. Basic Delta Publish

Header	Comment
`Topic`	Sets the topic to publish to. The topic specified must be a literal topic name. Regular expression characters in the topic name are not interpreted. Some topics in AMPS, such as views and conflated topics, cannot be published to directly. Instead, a publisher must publish to the underlying topics.
`Data`	The data to publish to the topic. The AMPS client does not interpret, escape, or validate this data: the data is provided to the server verbatim.

Delta Publish With CorrelationId

AMPS provides publishers with a header field that can be used to contain arbitrary data, the `CorrelationId`. A delta publish message can be used to update the `CorrelationId` as well as the data within the message.

Table 10.32. Delta Publish With CorrelationId

Header	Comment
`Topic`	Sets the topic to publish to. The topic specified must be a literal topic name. Regular expression characters in the topic name are not interpreted. Some topics in AMPS, such as views and conflated topics, cannot be published to directly. Instead, a publisher must publish to the underlying topics.
`Data`	The data to publish to the topic. The AMPS client does not interpret, escape, or validate this data: the data is provided to the server verbatim.
`CorrelationId`	The `CorrelationId` to provide on the message. AMPS provides the `CorrelationId` to subscribers. The `CorrelationId` has no significance for AMPS. The `CorrelationId` may only contain characters that are valid in base-64 encoding.

Delta Publish With Explicit SOW Key

When publishing to a SOW topic that is configured to require an explicit SOW key, the publisher needs to set the SowKey header on the message.

Table 10.33. Delta Publish with Explicit SOW Key

Header	Comment
`Topic`	Sets the topic to publish to. The topic specified must be a literal topic name. Regular expression characters in the topic name are not interpreted. Some topics in AMPS, such as views and conflated topics, cannot be published to directly. Instead, a publisher must publish to the underlying topics.
`Data`	The data to publish to the topic. The AMPS client does not interpret, escape, or validate this data: the data is provided to the server verbatim.
`SowKey`	The SOW Key to use for this message. This header is only supported for publishes to a topic that requires an explicit SOW Key.

Delta Subscribing

This section presents common recipes for subscribing to a topic in AMPS using the `Command` or `Message` interfaces. This section provides information on how to configure the request to AMPS. You can adapt this information to your application and the specific interface you are using.

Basic Delta Subscription

In its simplest form, a delta subscription needs only the topic to subscribe to.

Table 10.34. Basic Delta Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. All messages from the topic will be delivered on this subscription. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.

Basic Delta Subscription With Options

In its simplest form, a subscription needs only the topic to subscribe to. To add options to the subscription, set the `Options` header on the `Command`.

Table 10.35. Basic Delta Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. All messages from the topic will be delivered on this subscription. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Options`	A comma-delimited set of options for this command. See the `AMPS Command Reference` for a description of supported options.

Content Filtered Delta Subscription

To provide a content filter on a subscription, set the `Filter` property on the command. The *AMPS User Guide* provides details on the filter syntax.

Table 10.36. Content Filtered Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.

Header	Comment
`Filter`	Sets the content filter to be applied to the subscription. Only messages that match the content filter will be provided to the subscription.

Bookmark Delta Subscription

To create a bookmark subscription, set the `Bookmark` property on the command. The property can be either a specific bookmark, a timestamp, or one of the client-provided constants. The *AMPS User Guide* provides details on creating timestamps. Notice that the `MOST_RECENT` constant tells the AMPS client to find the appropriate message in the client bookmark store and begin the subscription at that point. In this case, the client sends that bookmark value to AMPS.

Table 10.37. Bookmark Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.
`Bookmark`	Sets the point in the transaction log at which the subscription will begin. The bookmark provided can be a specific AMPS bookmark, a timestamp, or one of the client-provided constants. AMPS also accepts a comma-delimited list of bookmarks. In this case, AMPS begins the subscription from whichever of the bookmarks is earliest in the transaction log.

Bookmark Delta Subscription With Content Filter

To create a bookmark subscription, set the `Bookmark` property on the command. The property can be either a specific bookmark, a timestamp, or one of the client-provided constants. The *AMPS User Guide* provides details on creating timestamps. Notice that the `MOST_RECENT` constant tells the AMPS client to find the appropriate message in the client bookmark store and begin the subscription at that point. In this case, the client sends that bookmark value to AMPS.

To add a filter to a bookmark subscription, set the `Filter` property on the command. The *AMPS User Guide* provides details on the filter syntax.

Table 10.38. Bookmark Delta Subscription With Content Filter

Header	Comment
`Topic`	Sets the topic to subscribe to. The topic provided can be either the exact name of the topic, or a regular expression that matches the names of the topics for the subscription.
`Bookmark`	Sets the point in the transaction log at which the subscription will begin. The bookmark provided can be a specific AMPS bookmark, a timestamp, or one of the client-provided constants.

Header	Comment
	AMPS also accepts a comma-delimited list of bookmarks. In this case, AMPS begins the subscription from whichever of the bookmarks is earliest in the transaction log.
`Filter`	Sets the content filter to be applied to the subscription. Only messages that match the content filter will be provided to the subscription.

SOW and Delta Subscribe

This section presents common recipes for atomic sow and delta subscribe in AMPS using the `Command` or `Message` interfaces. This section provides information on how to configure the request to AMPS. You can adapt this information to your application and the specific interface you are using.

Basic SOW and Delta Subscribe

In its simplest form, a SOW and Delta Subscribe needs only the topic to subscribe to.

Table 10.39. Basic SOW Query

Header	Comment
`Topic`	Sets the topic to query and subscribe to. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.

SOW and Delta Subscribe With Options

In its simplest form, a SOW and subscribe command needs only the topic to subscribe to. To add options to the subscription, set the `Options` header on the `Command`.

Table 10.40. Basic SOW and Delta Subscribe with Options

Header	Comment
`Topic`	Sets the topic to query and subscribe to. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Options`	A comma-delimited set of options for this command. See the `AMPS Command Reference` for a full description of supported options. The most common options for this command are: `oof` — Request out of order notifications `timestamp` — Include timestamps on messages

Content Filtered SOW and Delta Subscribe

To provide a content filter on a SOW and Delta Subscribe, set the `Filter` property on the command. The *AMPS User Guide* provides details on the filter syntax.

Table 10.41. Content Filtered SOW and Delta Subscribe

Header	Comment
`Topic`	Sets the topic to query and subscribe to. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Filter`	Sets the content filter to be applied to the query. Only messages that match the content filter will be returned in response to the query.

Historical SOW and Subscribe

To create a historical SOW query with a subscription, set the `Bookmark` property on the command. The property can be either a specific bookmark or a timestamp. The *AMPS User Guide* provides details on creating timestamps. This command is only supported on SOW topics that are recorded in an AMPS transaction log. If the `Bookmark` provided is a value other than NOW (`0|1|`), the SOW topic must have `History` enabled.

Table 10.42. Historical SOW and Subscribe

Header	Comment
`Topic`	Sets the topic to query and subscribe to. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Bookmark`	Sets the historical point in the SOW at which to query. The query returns the saved state of the records in the SOW as of the point in time specified in this header.

Historical SOW and Delta Subscribe With Content Filter

To create a historical SOW query with a subscription, set the `Bookmark` property on the command. The property can be either a specific bookmark or a timestamp. The *AMPS User Guide* provides details on creating timestamps. This command is only supported on SOW topics that are recorded in an AMPS transaction log. If the `Bookmark` provided is a value other than NOW (`0|1|`), the SOW topic must have `History` enabled.

Table 10.43. Historical SOW and Delta Subscribe With Content Filter

Header	Comment
`Topic`	Sets the topic to query. The topic specified can be the literal topic name, or a regular expression that matches multiple topics.
`Bookmark`	Sets the historical point in the SOW at which to query. The query returns the saved state of the records in the SOW as of the point in time specified in this header.

Header	Comment
`Filter`	Sets the content filter to be applied to the query. Only messages that match the content filter will be provided to the query.

SOW Delete

This section presents common recipes for sending a sow_delete command using the `Command` or `Message` interfaces. This section provides information on how to configure the request to AMPS. You can adapt this information to your application and the specific interface you are using.

Delete All Records in a SOW

To delete all records in a SOW, provide a filter that evaluates to TRUE for every record in the SOW. By convention, 60East recommends `1=1` for the filter.

Table 10.44. Delete All Records in a SOW

Header	Comment
`Topic`	Sets the topic from which to remove records.
`Filter`	A filter specifying the messages to remove. By convention, use `1=1` to remove all records in the SOW.

Delete SOW Records Matching a Filter

To delete the records that match a particular filter, provide the filter in the `sow_delete` command.

Table 10.45. Delete All Records in a SOW

Header	Comment
`Topic`	Sets the topic from which to remove records.
`Filter`	A filter specifying the messages to remove.

Delete A Specific Message By Data

To delete a specific message, provide the data for the message to delete. With this form of SOW delete, AMPS deletes the message that would have been updated if the data were provided as a publish message. Notice that this form of `sow_delete` relies on the `Key` definition in the SOW configuration, and is not generally useful with explicitly-keyed SOW topics.

Table 10.46. Delete All Records in a SOW

Header	Comment
`Topic`	Sets the topic from which to remove records.
`Data`	The message to remove.

Deleting Specific Messages Using Keys

To delete specific messages using SOW keys, provide the SOW keys for the message to delete.

Table 10.47. Delete All Records in a SOW

Header	Comment
`Topic`	Sets the topic from which to remove records.
`SOWKeys`	A comma-delimited list of SOWKeys that specify the messages to remove.

Acknowledging Messages from a Queue

To acknowledge messages from an AMPS queue, provide the bookmarks for the messages to acknowledge. Notice that this is the only form of the sow_delete command that can acknowledge messages from a queue, and that this form of `sow_delete` is not accepted for topics that are not queue topics.

Table 10.48. Acknowledging a queue message

Header	Comment
`Topic`	Sets the topic that contains the messages to acknowledge.
`Bookmark`	A comma-delimited list of Bookmarks that specify the messages to acknowledge.

Chapter 11. Utilities

The AMPS Python client includes a set of utilities and helper classes to make working with AMPS easier.

11.1. Composite Message Types

The client provides a pair of classes for creating and parsing composite message types.

- `CompositeMessageBuilder` allows you to assemble the parts of a composite message and then serialize them in a format suitable for AMPS.
- `CompositeMessageParser` extracts the individual parts of a composite message type

For more information regarding composite message types, refer to Chapter 4.3.

Building Composite Messages

To build a composite message, create an instance of `CompositeMessageBuilder`, and populate the parts. The `CompositeMessageBuilder` copies the parts provided, in order, to the underlying message. The builder simply writes to an internal buffer with the appropriate formatting, and does not allow you to update or change the individual parts of a message once they've been added to the builder.

The snippet below shows how to build a composite message that includes a JSON part, constructed as a string, and a binary part consisting of the bytes from an `array.array` that contains doubles.

```
json = '{"data":"sample"}';


theData = array.array('d')
# populate theData
...


# Create the payload for the composite message.
builder = AMPS.CompositeMessageBuilder();

# Construct the composite
builder.append(json)
builder.append(theData.tostring())

# Send the message
client.publish("messages", builder.get_data())
```

Parsing Composite Messages

To parse a composite message, create an instance of `CompositeMessageParser`, then use the `parse()` method to parse the message provided by the AMPS client. The `CompositeMessageParser` gives you access to each part of the message as a sequence of bytes.

For example, the following snippet parses and prints messages that contain a JSON part and a binary part that contains an array of doubles.

```
parts = parser.parse(message)

json = parser.get_field(0)
theData = array.array('d')
theData.fromstring(parser.get_field(1))

print "Received message with %d parts." % parts
   print json
   datastring = ""
   for d in theData:
      datastring += "%f " % d
   print datastring
```

Notice that the receiving application is written with explicit knowledge of the structure and content of the composite message type.

11.2. NVFIX Messages

The client provides a pair of classes for creating and parsing NVFIX messages.

- `nvfixbuilder` allows you to assemble a NVFIX message and then serialize it in a format suitable for AMPS.
- `nvfixshredder` extracts the individual fields of a NVFIX message type.

Building NVFIX Messages

To build a NVFIX message, create an instance of `nvfixbuilder`, then add the fields of the message using `append()`. `nvfixBuilder` copies the fields provided, in order, to the underlying message. The builder simply writes to an internal buffer with the appropriate formatting, and does not allow you to update or change the individual fields of a message once they've been added to the builder.

The snippet below shows how to build a FIX message and publish it to the AMPS client.

```
# create the payload for the NVFIX Message
builder = AMPS.NVFIXBuilder()

# construct the NVFIX message
builder.append("sample","data")
builder.append("even", "more data")
...

# display the data
print builder.get_string()

# publish the message
client.publish("messages-sow", builder.get_string())
```

Parsing NVFIX Messages

To parse a NVFIX message, create an instance of `nvfixShredder`, then use the `to_Map()` method to parse the message provided by the AMPS client. The `nvfixShredder` gives you access to each field of the message in a map.

The snippet below shows how to parse and print a NVFIX message.

```
# create the shredder for the message and subscribe to the topic
shredder = AMPS.NVFIXShredder()
message = client.subscribe(topic="messages-sow", timeout=5000)

# shred the message to a map
message_map = shredder.to_map(message.next().getData())

# display the values of the message
for key in message_map :
    print key + " " + message_map[key]
```

11.3. FIX Messages

The client provides a pair of classes for creating and parsing FIX messages.

- `fixbuilder` allows you to assemble a FIX message and then serialize it in a format suitable for AMPS.
- `fixshredder` extracts the individual fields of a FIX message.

Building FIX Messages

To build a FIX message, create an instance of `fixbuilder`, then add the fields of the message using `append()`. `fixBuilder` copies the fields provided, in order, to the underlying message. The builder simply writes to an internal buffer with the appropriate formatting, and does not allow you to update or change the individual fields of a message once they've been added to the builder.

The snippet below shows how to build a FIX message and publish it to the AMPS client.

```
# create the payload for the FIX Message
builder = AMPS.FIXBuilder()

# construct the FIX message
builder.append(0,"data")
builder.append(1, "more data")
...

# display the data
print builder.get_string()

# publish the message
client.publish("messages-sow", builder.get_string())
```

Parsing FIX Messages

To parse a FIX message, create an instance of `fixShredder`, then use the `to_Map()` method to parse the message provided by the AMPS client. The `fixShredder` gives you access to each field of the message in a map.

The snippet below shows how to parse and print a FIX message.

```
# create the shredder for the message and subscribe to the topic
shredder = AMPS.FIXShredder()
message = client.subscribe(topic="messages-sow", timeout=5000)

# shred the message to a map
message_map = shredder.to_map(message.next().getData())

# display the values of the message
for key in message_map :
    print (str(key) + " " + message_map[key])
```

Chapter 12. Advanced Topics

12.1. Implementing Message Handlers in C or C++

The AMPS Python client provides a wrapper that works with the python `ctypes` module to allow you to create message handlers in C or C++ and expose them to Python. This can improve performance in the message handler. When you use this technique, messages are delivered directly from the C++ client to your message handler: there is no Python code involved in handling the messages.

To use this capability, you:

1. Create a message handler with C linkage, and compile that message handler into a shared library
2. In your Python program, use the `ctypes` module to load the library
3. Construct an instance of CMessageHandler, a wrapper object that holds a pointer to the message handler function and the userdata to be provided to the handler during each call.
4. Pass the CMessageHandler to any method that expects a message handler.

The AMPS Python client registers the pointer and user data you provide as a C++ message handler. Once the handler is registered, no Python code is called when providing messages to the handler.

Implementing the Handler

To use this capability, you create a message handler that exposes a function with the following signature having C linkage:

```
extern "C"
void my_message_handler(AMPS::Message &message,
                        void *userdata);
```

Notice that this signature is the same signature used by message handlers in the AMPS C++ client.You implement the function and compile it into a shared library or DLL, using the instructions provided with your Python implementation. For details on the C++ client, you can install the client itself, or consult the documentation at http://docs.crankuptheamps.com/api/cpp/index.html.

Loading and Using the Handler

Once you've compiled the library, you use the `ctypes` module to load the library. You then create an instance of the message handler wrapper, and pass that wrapper to the AMPS client methods, as shown below:

```
import ctypes
import AMPS

...

# assumes that client is already created and connected

# load the shared object
```

```
dll = ctypes.CDLL("./libmymessagehandler.so")

# create a handler that points to the underlying C function
# and bind the user data to that handler.
handler = AMPS.CMessageHandler(dll.my_message_hander, "user data")

# handler can be used anywhere you would use a message handler
client.subscribe(handler, "myTopic")

client.set_last_chance_message_handler(handler)

# and so it goes
```

The `AMPS.CMessageHandler` type accepts a pointer to a message handler with the signature shown above and a Python object that can be marshalled into a native C type through the `ctypes` interface. Once marshalled, the object will be cast to a `void *` and provided in the `userdata` parameter of the message handler. Marshalling the userdata parameter follows the `ctypes` module conventions. If you need to explicitly control the way an object is marshalled, you can construct one of the `ctypes` objects and pass that new object into the method.

12.2. Using the C++ Client

While the AMPS Python client provides enough performance for a wide variety of applications, in some cases, using the underlying C++ implementation can provide extra performance. The AMPS Python client works with the `ctypes` module to allow you to pass the underlying C++ client to an arbitrary function, effectively allowing you to integrate C++ code directly into your Python program.

Consider using the C++ client directly when latency is at a premium or when your application works directly with C++. For example, you might you use the client directly when:

- You are assembling messages from a C++ library without a Python binding
- You need to customize client behavior that is implemented in C++ (for example, implementing a custom SubscriptionManager or BookmarkStore)
- Your application needs to execute a set of commands with AMPS with minimal latency. For example, you might need to publish an array of values as individual messages with as little latency as possible. In this case, using the underlying C++ client directly can reduce latency.

To use the underlying C++ client, you:

1. Create a function with C linkage, and compile that function into a shared library. One of the parameters of the function should be a reference to an `AMPS::Client`.
2. In your Python program, use the `ctypes` module to load the library.
3. Call the function on the library, passing the appropriate parameters for the C function.

Implementing the C++ Function

The only requirement on the C++ function is that it have C linkage and that one of the parameters is a reference to an `AMPS::Client`. By convention, 60East recommends that the first parameter is the `AMPS::Client`. However, this is not a requirement of the interface.

For example, a function that simply takes an `AMPS::Client` has the following signature:

```
extern "C"
void configure_client(AMPS::Client& client);
```

While a function that takes a client, a topic, and a pointer to data to be published might have the following signature:

```
extern "C"
void publish_data(AMPS::Client& client,
                  const char * topic,
                  const char * data);
```

The `ctypes` module provides a standard `AMPS::Client` to these functions. Although the `Client` has been created by Python code, there is nothing Python-specific about the object within the C++ function. You can use the Client just as you would any other Client object.

You can also use the `ctypes` binding with `AMPS::HAClient`, as shown below:

```
extern "C"
void install_server_chooser(AMPS::HAClient& client);
```

Since the `ctypes` module pases the data through the C ABI, the module is not able to perform extensive type checking on C++ types. Your Python code must be careful to pass only objects of the appropriate type, or you may cause a segmentation violation. For example, if a method expecting an `HAClient` receives a `Client` and calls `connectAndLogon` (which is not a method provided by Client), your program will likely crash.

The `ctypes` module does not provide strong type-safety guarantees for C++ classes. It is your responsibility to ensure that you call methods with objects of the appropriate type.

The `ctypes` module calls your function through an extern "C" interface. C++ exceptions cannot be propagated out of a function with C binding. You must catch all exceptions that may be thrown, or your application will likely crash.

Loading and Using the Function

Once you've compiled the library, you use the `ctypes` module to load the library. You can then call the function directly from Python, using the name of the C function and passing the appropriate arguments.

Let's look at a simple example. For this example, assume that you have compiled a module named `module.so` with the following function:

```
extern "C" void publish_message(AMPS::Client& client,
                                const char* topic,
                                const char* data)
{
  try {
     if(&client && topic && data)
    {
      client.publish(topic,data);
    }
  }
  catch (AMPS::Exception& e)
```

```
  {
     // Handle error reporting and recovery logic
  }
}
```

You can load the module and call the function as shown below:

```
import ctypes

module = ctypes.CDLL("module.so")
client = AMPS.Client("client")
client.connect("tcp://localhost:9007/amps/json")
client.logon()

module.publish_data(client, "my_topic", "some_data")
```

The `ctypes` module handles type conversions between Python and C types. In this case, the module passes the underlying Python client as the first argument of the C function. The two Python strings are passed as NULL-terminated `char *` arrays.

The ctypes module also handles more complicated signatures and correctly passes arrays. For example, you could implement a method that publishes an array of Python values as follows:

```
extern "C" void vector_publish(AMPS::Client& client, const char* topic,
 const char** data, size_t vector_length)
{
  try {
    if(topic && &client)
    {
      for(;vector_length;--vector_length,++data)
      {
        client.publish(topic,*data);
      }
    }
  }
  catch (AMPS::Exception& e)
  {
     // Handle reporting and recovery logic
  }
}
```

You could then use this function from Python as follows:

```
import ctypes

module = ctypes.CDLL("module.so")
client = AMPS.Client("client")
client.connect("tcp://localhost:9007/amps/json")
client.logon()

TOPIC = "topic"
DATA  = [{"data":x, "string_data":"string_data"} for x in range(5)]


#initialize vector of data to publish by
```

```
#dumping the dictionaries to JSON strings

vector = [json.dumps(data) for data in DATA[1:]]

# Set up the parameters to be passed to a C
# function as explained in the ctype documentation
param = (ctypes.c_char_p * len(vector))()
param[:] = vector

# Call the function
module.vector_publish(client,TOPIC, param,len(param))
```

The sample above creates an array of dictionaries and creates an array of JSON objects from those dictionaries.

In this case, it is important for us to control how the array of JSON objects is passed to the C function. We need to pass an array of C-style strings, that is, `const char**`. To control how the array is marshalled, the sample creates an object that knows how to translate between a Python array and `const char**`, then assigns the array to that object (see the `ctype` documentation for full details). Once we have that object, we simply call the `vector_publish` function. None of the Python infrastructure is visible to the `vector_publish` function: that function is able to use the provided data as native C++ data.

12.3. Transport Filtering

The AMPS Python client offers the ability to filter incoming and outgoing messages in the format they are sent and received on the network. This allows you to inspect or modify outgoing messages before they are sent to the network, and incoming messages as they arrive from the network. This can be especially useful when using SSL connections, since this gives you a way to monitor outgoing network traffic before it is encrypted, and incoming network traffic after it is decrypted.

To create a transport filter, you create a callable that expects a string that contains the raw data, and a direction parameter indicating whether the string is output or not. For example, the following function simply prints the direction and data:

```
def printing_filter(data, direction):
   if direction:
       print "OUTGOING ---> %s" % data
   else:
       print "INCOMING ---> %s" %data
```

You then register the filter by calling `set_transport_filter` with the callable, as shown below.

```
# client is an AMPS client
client.set_transport_filter(printing_filter)
```

12.4. Using SSL

The AMPS Python client includes support for Secure Sockets Layer. To use this support in the Python client using the default SSL implementation for the Python installation, you need only use `tcps` for the transport type in the connection string.

Loading a Different SSL Implementation

The Python client also allows you to load and use an SSL implementation other than the default implementation for the Python installation. The AMPS Python client provides the method ssl_init, which takes the name of the library to load or a full path to the file that contains the library. For example, to load the SSL implementation at `/opt/mycorp/trusted/vetted_ssl.so`, you could use the following line of code:

```
AMPS.ssl_init("/opt/mycorp/trusted/vetted_ssl.so")
```

You must load the SSL library before making the connection.

Chapter 13. Performance Tips and Best Practices

This chapter presents tips and techniques for writing high-performance applications with AMPS. This section presents principles and approaches that describe how to use the features of AMPS and the AMPS client libraries to achieve high performance and reliability.

Specific techniques (for example, the details on how to write a message handler) are described in other parts of the AMPS documentation and referenced here. Other techniques require information specific to the application (for example, determining the minimum set of information required in a message), and are best done as part of your application design.

All of the recommendations in this section are general guidelines. There are few, if any, universal rules for performance: at times, a design decision that is absolutely necessary to meet the requirements for an application might reduce performance somewhat. For example, your application might involve sending large binary data that cannot be incrementally updated. That application will use more bandwidth per message than an application that sends 100-byte messages with fields that can be incrementally updated. However, since the application depends on being able to deliver the binary payloads, this difference in bandwidth consumption is a part of the requirements for the application, not a design decision that can be optimized.

13.1. Measure Performance and Set Goals

The most important tools for creating high performance applications that use AMPS are clear goals and accurate measurement. Without accurate measurement, it's impossible to know whether a particular change has improved performance or not. Without clear goals, it's difficult to know whether a given result is sufficient, or whether you need to continue improving performance.

60East recommends that your measurements include baseline metrics for the part of your message processing that does not involve AMPS. As an example, imagine your task is to reduce the amount of time that elapses between when an order is sent and when the processed response is received from 100ms in total to 85ms in total. To achieve this reduction, you might first measure the processing that your application performs on the order. If that processing consumes 65ms, the most effective optimization may be to improve the order processing. On the other hand, if processing an order consumes 15ms, then optimizing message delivery or network utilization may be the most effective way to meet your goals.

When measuring performance, simulate your production environment as closely as possible. For example, AMPS is highly parallelized, so sending a pattern of subscriptions and publishes from a single test client that would normally come from 20 clients will produce a very different performance profile. Likewise, AMPS can typically perform at rates that fill the available bandwidth. Performance measured on a 1GbE connection may be very different than performance measured over a 10GbE connection. Consider the characteristics of your data, and the number of messages you expect to store and process. A 1GB data set consisting of 1 million records will perform differently than a 1GB data set consisting of 10 million records, or a 1GB data set consisting of 100 records.

When collecting information about performance, 60East recommends enabling persistence for the Statistics Database (`stats.db`), so you can easily collect historical data on both AMPS and the operating system. For example, a dip in performance correlated with high CPU and memory usage at the same time each day may be correlated with other activity on the system (such as cron jobs or close of business processing). In a situation like that, where the performance reduction is based on factors external to the AMPS application, the overall system metrics captured in `stats.db` can help you re-create the external state and understand the state of the system as a whole. AMPS collects

the statistics in memory by default, and persisting that data into a database does not typically have a measurable effect on performance itself, but makes measuring and tuning performance much easier.

For peformance testing, 60East recommends using dedicated hardware for AMPS to eliminate the effects of other processes. If dedicated hardware is not available and other processes are consuming resources, 60East recommends disabling AMPS NUMA tuning to ensure that AMPS threads do not unnecessarily compete with other processes during performance tuning.

13.2. Simplify Message Format and Contents

AMPS supports a wide range of message types, and is capable of filtering and processing large and complex messages. For many applications, the simplicity of being able to use messages that contain the full information is the most important consideration. For other applications, however, achieving the minimum possible latency and the maximum possible network utilization is important enough to warrant choosing a simplified message format.

To simplify message contents, carefully consider the information that downstream processors require. If a downstream process will not use information in the message, there is no need to send the information. For example, consider an application that provides orders from a UI. In such an application, the object that represents the order often contains information relevant to the local state of the application that is not relevant to a downstream system. Rather than simply serializing the full object, your application may perform better if you serialize only the fields that a downstream system will take action on.

To simplify message format, choose the simplest format that can convey the information that your application needs. The general principle is that the simpler the message format is, the more quickly AMPS and client libraries can parse messages of that type. Likewise, the more complicated the structure of each message is, the more work is required to parse the message. For the highest levels of performance, 60East recommends keeping the message structure simple and preferring message formats such as NVFIX, BFlat, or JSON as compared with more complicated formats such as XML or BSON.

13.3. Use Content Filtering Where Possible

AMPS content filtering helps your application perform better by ensuring that your application only receives the messages that it needs. Wherever possible, we recommend using content filtering to precisely specify which messages your application needs. In particular, if at any point your application is receiving a message, parsing the message, and then determining whether to act on the message or not, 60East recommends using content filters to ensure that your application only receives messages that it needs to act on.

13.4. Use Asychronous Message Processing

The synchronous message processing interface is straightforward, and presents a convenient interface for getting started with AMPS.

However, the `MessageStream` used by the synchronous interface makes a full copy of each message and provides it from the background reader thread to the thread that consumes the message. This memory overhead and synchronization between the reader thread and consumer thread happens regardless of whether the application needs all of the header fields in the message or even processes the message. The `MessageStream` also does not take into account the speed at which your program is consuming messages, and will read messages into memory as fast as the network and processor allow. If your application cannot consume messages at wire speed, this can lead to increasing memory consumption as the application falls further behind the `MessageStream`.

Most applications see improved performance by using a `MessageHandler`. With this approach, the `MessageHandler` does minimal work. If more extensive processing is needed, the `MessageHandler` dispatches the work to another thread: but it does this only when the work is necessary, and it only saves the part of the message needed to accomplish the work.

13.5. Use Hash Indexes Where Possible

When querying a SOW, hash indexes on SOW topics are supported for exact matching on string data as described in the *AMPS User Guide*. A hash index can perform many times faster than a parallel query. If the query pattern for your application can take advantage of hash indexes, 60East recommends creating those hash indexes on your SOW topics.

13.6. Use a Failed Write Handler and Exception Listener

In many cases, particularly during the early stages of development, performance problems can point to defects in the application. Even after the application is tuned, monitoring for failure is important to keep applications running smoothly.

60East recommends always installing a failed write handler if your application is publishing messages. This will help you to quickly identify cases where AMPS is rejecting publishes due to entitlement failures, message type mismatches, or other similar problems.

60East recommends always installing an exception listener if your application is using asynchronous message processing. This will help you to identify and correct any problems with your message handler.

13.7. Reduce Bandwidth Requirements

In many applications that use AMPS, network bandwidth is the single most important factor in overall performance. Your application can use bandwidth most efficiently by reducing message size. For example, rather than serializing an entire object, you might serialize only the fields that the remote process needs to act on, as mentioned above. Likewise, rather than sending one message that contains a collected set of information that processors will need to extract, consider sending a message in the units that processors will work with. This can reduce bandwidth to processors substantially. For example, rather than sending a single message with all of the activity for a single customer over a given period of time (such as a trading day), consider breaking out the record into the individual transactions for the customer.

Tune Batch Size for SOW Queries

As described in Section 6.3, tuning the batch size for SOW queries can improve overall performance by improving network utilization. In addition, because the AMPS header is only parsed once per batch, a larger batch size can dramatically improve processing performance for smaller messages.

The AMPS clients default to a batch size of `10`. This provides generally good performance for most transactional messages (such as order records or inventory records). For large messages, particularly messages greater than a megabyte in size, a batch size of `1` may reduce memory pressure in the client and improve performance.

With smaller messages (for example, message sizes of a few hundred bytes), 60East recommends measuring performance with larger batch sizes such as `50` or `100` . For large messages, reducing the batch size may improve overall performance by requiring less memory consumption on the AMPS server.

Conflate Fast-Changing Information

If your data source publishes information faster than your clients need to consume it, consider using a conflated topic. For example, in a system that presents a user interface and displays fast-moving data, it is common for the data to change at a rate faster than the user interface can format and render the data. In this case, a conflated topic can both reduce bandwidth and simplify processing in the user interface.

Minimize Bandwidth for Updates

If your application uses a SOW and processes frequent updates, consider using delta publish and delta subscribe to reduce the size of the messages transmitted. These features are designed to minimize bandwidth while still providing full-fidelity data streams.

Conflate Queue Acknowledgements

The AMPS clients include the ability to conflate acknowledgements back to AMPS as queue messages are processed. Using these features, with an appropriate `max_backlog`, can reduce the amount of network traffic required for acknowledgements.

Use a Transaction Log When Monitoring Publish Failures

When a topic is not covered by a transaction log, AMPS returns acknowledgment messages for every publish that requests one. This ensures that each message is acknowledged, even when AMPS has no persistent record of the messages in the topic. However, acknowledging each message requires more network traffic for each publish message.

When a topic is covered by a transaction log, AMPS conflates persisted acknowledgments. Conflation is possible in this case because AMPS has a full record of the messages and does not have to store additional state to conflate the acknowledgements. With conflated acknowledgements, AMPS will send a success acknowledgement periodically that covers all messages up to that point. If a message fails, AMPS immediately sends the conflated success acknowledgement for all previous messages and the failure acknowledgement for the failed message.

Combine Conflation and Deltas

In many cases, using an approach that combines delta publishes to a SOW with delta subscriptions to a conflated topic can dramatically reduce bandwidth to the application with no loss of information.

13.8. Limit Unnecessary Copies

One of the most effective ways to increase performance is to limit the amount of data copied within your application.

For example, if your message handler submits work to a set of processors that only use the `Data` and `Bookmark` from a `Message`, create a data structure that holds only those fields and copy that information into instances of that data structure rather than copying the entire `Message`. While this approach requires a few extra lines of code, the performance benefits can be substantial.

When publishing messages to AMPS, avoid unnecessary copies of the data. For example, if you have the data in a byte array, use the `publish` methods that use a byte array rather than converting the data to a string unnecessarily. Likewise, if you have the data in the form of a string, avoid converting it to a byte array where possible.

13.9. Manage Publish Stores

When using a publish store, the Client holds messages until they are acknowledged as persisted by AMPS, as determined by the replication configuration for the AMPS instance.

In the event that an instance with `sync` replication goes offline, the publish store for the Client will grow, since the messages are not being fully persisted. To avoid this problem, 60East recommends that an instance that uses `sync` replication always configure Actions to automatically downgrade the replication link if the remote instance goes offline for a period of time, and upgrade the link when the remote instance comes back online.

See the "High Availability and Replication" chapter in the *User Guide* for more information on replication, sync and async acknowledgement modes, and the Actions used to manage replication.

13.10. Work with 60East as Necessary

60East offers performance advice adapted for your specific usage through your support agreement. Once you've set your performance goals, worked through the general best practices and applied the practices that make sense for your application, 60East can help with detailed performance tuning, including recommendations that are specific to your use case and performance needs.

Chapter 14. Migrating from Earlier Python Implementations

The AMPS Python client builds on the AMPS C++ client to provide all of the features of the C++ client at high performance. 60East previously produced a different implementation of a Python client, which is now deprecated.

The AMPS Python client is fully-compatible with the previous implementation. This client also adds support for:

- High availability clients.
- Bookmark subscriptions.
- Features added after AMPS 3.5

The earlier implementation did not fully follow the style guidelines in Python Enhancement Proposal 8 (PEP 8). Functions and objects in the previous implementation were named with camel case style, while functions and objects in the Python client follow the standard Python convention of lowercase words separated by underscores.

The Python client contains compatibility functions for the previous implementation. To update your code to follow PEP 8 conventions, replace camel case functions with the equivalent. For example, the `sowAndSubscribe` function in your code becomes `sow_and_subscribe` with the Python client.

Because camel case style functions are provided for compatibility, there are no camel case style functions for features that weren't included in the previous implementation.

.

Appendix A. Exceptions

The following table details each of the exception types thrown by AMPS.

Table A.1. Exceptions supported in Client and HAClient

Exception	When	Notes
AlreadyConnectedError	Connecting	Thrown when `connect()` is called on a Client that is already connected.
AMPSError	Anytime	Base class for all AMPS exceptions.
AuthenticationError	Anytime	Indicates an authentication failure occurred on the server.
BadFilterError	Subscribing	This typically indicates a syntax error in a filter expression.
BadRegexTopicError	Subscribing	Indicates a malformed regular expression was found in the topic name.
CommandError	Anytime	Base class for all exceptions relating to commands sent to AMPS.
ConnectionError	Anytime	Base class for all exceptions relating to the state of the AMPS connection.
ConnectionRefusedError	Connecting	The connection was actively refused by the server. Validate that the server is running, that network connectivity is available, and the settings on the client match those on the server.
DisconnectedError	Anytime	No connection is available when AMPS needed to send data to the server *or* the user's disconnect handler threw an exception.
InvalidTopicError	SOW query	A SOW query was attempted on a topic not configured for SOW on the server.
InvalidTransportOptionsError	Connecting	An invalid option or option value was specified in the URI.
InvalidURIError	Connecting	The URI string provided to `connect()` was formatted improperly.
MessageTypeError	Connecting	The class for a given transport's message type was not found in AMPS.
MessageTypeNotFoundError	Connecting	The message type specified in the URI was not found in AMPS.
NameInUseError	Connecting	The client name (specified when instantiating `Client`) is already in use on the server.
RetryOperationError	Anytime	An error occurred that caused processing of the last command to be aborted. Try issuing the command again.
StreamError	Anytime	Indicates that data corruption has occurred on the connection between the client and server. This usually indicates an internal error inside of AMPS -- contact AMPS support.

Exception	When	Notes
SubscriptionAlreadyExistsException	Subscribing	A subscription has been requested using the same command ID string as another subscription. Create a unique command ID string for every subscription.
TimedOutError	Anytime	A timeout occurred waiting for a response to a command.
TransportTypeError	Connecting	Thrown when a transport type is selected in the URI that is unknown to AMPS.
UnknownError	Anytime	Thrown when an internal error occurs. Contact AMPS support immediately.
UsageException	Changing the properties of an object	Thrown when the object is not in a valid state for setting the properties. For example, some properties of a Client (such as the BookmarkStore used) cannot be changed while that client is connected to AMPS.