AMPS C/C++ Development Guide

AMPS C/C++ Development Guide

5.2

Publication date Jun 26, 2017

Table of Contents

Chapter 1. Introduction

This document explains how to use the C/C++ client for AMPS. Use this document to learn how to install, configure, develop C and C++ applications that use AMPS.

1.1. Prerequisites

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

- Developing in C or C++. To be successful using this guide, you will need to possess a working knowledge of C or C++.
- 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.

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. C & C++ Support Matrix

This version of the AMPS C++ client supports the following operating systems and features:

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	✓	✓	

This version of the AMPS C++ client has been tested with the following compilers and versions. Other compilers or versions may work, but have not been tested by 60East:

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

Chapter 2. Installing the AMPS Client

2.1. Obtaining the Client

Before using the client, you will need to download and install it on your development computer. The client is packaged into a single file, `amps-c++-client-<version>.tar.gz`, where <version> is replaced by the version of the client, such as `amps-c++-client-3.3.0.zip`. In the following examples, the version number is omitted from the filename.

Once expanded, the `amps-c++-client` directory will be created, containing sources, samples and makefiles for the C++ client. You're welcome to locate this directory anywhere that seems convenient; but for the remainder of this book, we'll simply refer to this directory as the `amps-c++-client` directory.

2.2. Explore the client

The client is organized into a number of directories that you'll be using through this book. Understanding this organization now will save you time in the future. The top level directories are:

lib

The directory that contains built libraries. The distribution contains a set of pre-built libraries from 60East. You can use the prebuilt library or rebuild the library using the provided makefiles. Many applications rebuild the library using the exact compiler version and flags that they will use for the overall project.

src

Sources and makefile for the AMPS C++ client library.

include

Location of `include` files for C and C++ programs. When building your own program, you'll add the `include` directory to your `include` path.

samples

Getting started with a new C/C++ library can be challenging. For your reference, we provide a number of small samples, along with a makefile.

2.3. Build the Client

After unpacking the `amps-c++-client` directory, you must build the client library for your platform. To do so, change to the `amps-c++-client` directory and, from a command prompt, type:

```
make
```

or on Windows, from a Visual Studio Command Prompt, type:

```
msbuild
```

Upon successful completion, the AMPS libraries are built in the `lib`, and `samples` directories, respectively.

2.4. Test Connectivity to AMPS

Before writing programs using AMPS, make sure connectivity to an AMPS server from this computer is working. Launch a terminal window and change the directory to the `AMPS` directory in your AMPS installation, and use `spark` to test connectivity to your server, for example:

```
./bin/spark ping -type fix -server 192.168.1.2:9004
```

If you receive an error message, verify that your AMPS server is up and running, and work with your systems administrator to determine the cause of the connectivity issues. Without connectivity to AMPS, you will be unable to make the best use of this guide.

Chapter 3. Your First AMPS Program

In this chapter, we will learn more about the structure and features of the AMPS C/C++ library, and build our first C/C++ program using AMPS.

3.1. 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:

```
#include <ampsplusplus.hpp>
#include <iostream>

int main(void)
{
    const char* uri = "tcp://127.0.0.1:9007/amps/json";

    // Construct a client with the name "examplePublisher".

    AMPS::Client ampsClient("examplePublisher");

    try
    {
        // connect to the server and log on
        ampsClient.connect(uri);
        ampsClient.logon();

        // publish a JSON message
        ampsClient.publish("messages",
                           R"({ "message" : "Hello, World!" ,)"
                           R"(client" : 1 })");

    }
    catch (const AMPS::AMPSException& e)
    {
        std::cerr << e.what() << std::endl;
     exit(1);
    }
    return 0;
}
```

Example 3.1. Connecting to AMPS

In the preceding Example 3.1, we show the entire program; but future examples will isolate one or more specific portions of the code. The next section describes how to build and run the application and explains the code in further detail.

Build and run

To build the program that you've created:

1. Create a new `.cpp` file and use your c compiler to build it, making sure the *`amps-c++-client`*`/include` directory is in your compiler's `include` path.

2. Link to the `libamps.a` or `amps.lib` static libraries.

3. Additionally, link to any operating system libraries required by AMPS; a full list may be found by examining the Makefile and project files in the `samples` directory.

If the message is published successfully, there is no output to the console. We will demonstrate how to create a subscriber to receive messages in Chapter 4.

Examining the code

Let us now revisit the code we listed earlier.

```
#❶include <ampsplusplus.hpp>
#include <iostream>

int main()
{
   ❷const char* uri = "tcp://127.0.0.1:9007/amps/json";


   ❸AMPS::Client ampsClient("exampleClient");

   ❹try {
        ❺ampsClient.connect(uri);
        ❻ampsClient.logon();

        // publish a JSON message
        ❼ampsClient.publish("messages",
                            R"({ "message" : "Hello, World!" ,)"
                            R"( "client" : 1 })");
        ❽} catch (const AMPS::AMPSException& e) {
        std::cerr << e.what() << std::endl;
        exit(1);
    }
    ❾return 0;
    }
```

Example 3.2. Connecting to AMPS

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`. In this case, AMPS is configured to allow any message type on that transport, so we specify `json` in the URI to let AMPS know which message type this connection will use. Even though a transport that uses the `amps` protocol can accept multiple message types, each connection must specify the exact message type that connection will use. Check with the person who manages the AMPS instance to get the connection string to use for your programs.

❶ These are the `include` files required for an AMPS C++ Client. The first is `ampsplusplus.hpp`. This header includes everything needed to compile C++ programs for AMPS. The next `include` is the Standard C++ Library `<iostream>`, necessary due to use of `std::cerr` and `std::endl`.

❸ This is where we first interact with AMPS by instantiating an `AMPS::Client` object. `Client` is the class used to connect to and interact with an AMPS server. We pass the string `"exampleClient"` as the `clientName`. This name will be used to uniquely identify this client 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. AMPS C++ classes throw exceptions to indicate errors. For the remainder of our interactions with AMPS, if an error occurs, the exception thrown by AMPS will be caught and handled in the exception handler below.

❺ At this point, we establish a valid AMPS network connection and can begin to use it to publish and subscribe to messages. In this example, we use the URI specified earlier in the file. If any errors occur while attempting to connect to AMPS, the `connect()` method will throw an exception.

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, a single message is published to AMPS on the `messages` topic, containing the data `Hello world`. This data is placed into an XML message and sent to the server. Upon successful completion of this function, the AMPS client has sent the message to the server, and subscribers to the `messages` topic will receive this `Hello world` message.

❽ Error handling begins with the `catch` block. All exceptions thrown by AMPS derive from `AMPSException`. More specific exceptions may be caught to handle certain conditions, but catching `AMPSException&` allows us to handle all AMPS errors in one place. In this example, we print out the error to the console and exit the program.

❾ At this point we return from `main()` and our `ampsClient` object falls out of scope. When this happens AMPS automatically disconnects from the server and frees all of the client resources associated with the connection. In the AMPS C++ client, objects are reference-counted, meaning that you can safely copy a `client`, for example, and destroy copies of `client` without worrying about premature closure of the server connection or memory leaks.

3.2. Using the C client

The AMPS C/C++ client is built in two layers: the C layer that exposes lower- level primitives for sending and receiving messages to AMPS, and the C++ layer providing a set of abstractions over the C layer that makes it easier to work with AMPS and create robust applications. The C++ layer is recommended for many applications, since it offers a good balance of performance, control, and ease of use. If you are integrating AMPS into existing C code, or need fine-grained control over how your application interacts with AMPS, then you may choose to use the C layer directly.

The C Client offers low-level functionality for working with AMPS. With the C client, your application is responsible for correctly assembling the parameters to each command to AMPS and interpreting the response from AMPS. The C client does not provide higher-level abstractions such as publish stores, automatic failover and reconnection, sequence number management for published messages, and so on. Instead, you build the capabilities that your application needs over the low level primitives.

As an example, Example 3.3 shows the previous sample rewritten to use the C layer directly:

```
#include <amps.h>

int main()
{
```

```
❶char errorBuffer[256];
❷amps_handle message;
 amps_handle logon_command;
 amps_handle client;
❸amps_result result;

 client = amps_client_create("cClient"); ❹

❺result = amps_client_connect(client,
     "tcp://localhost:9007/amps/json");

 if(result != AMPS_E_OK) {
     amps_client_get_error(
         client, errorBuffer, sizeof(errorBuffer));
     printf("error %s\n", errorBuffer);
     amps_client_destroy(client);
     return 1;
 }

 ❻logon_command = amps_message_create(client);
 ❼amps_message_set_field_value_nts(m, AMPS_Command, "logon");
 amps_message_set_field_value_nts(m, AMPS_ClientName, "cClient");
 amps_message_set_field_value_nts(m, AMPS_MessageType, "json");
 ❽result = amps_client_send(client, logon_command);

 if (result != AMPS_E_OK)
 {
     amps_client_get_error(
         client, errorBuffer, sizeof(errorBuffer));
     printf("error %s\n", errorBuffer);
     amps_message_destroy(logon_command);
     amps_client_destroy(client);
     return 1;
 }

 ❾amps_message_destroy(logon_command);

 ❿message = amps_message_create(client);
 ⓫amps_message_set_field_value_nts(
         message, AMPS_CommandId, "12345");
 amps_message_set_field_value_nts(
         message, AMPS_Command, "publish");
 amps_message_set_field_value_nts(
         message, AMPS_Topic, "messages");
 amps_message_set_data_nts(
         message, "{\"message\":\"HelloWorld\"}");
 ⓬result = amps_client_send(client, message);
 if(result != AMPS_E_OK){
        ⓭amps_client_get_error(
         client, errorBuffer, sizeof(errorBuffer));
         printf("error sending: %s\n", errorBuffer);
  }
  ⓮ amps_message_destroy(message);
 }
```

```
    amps_client_destroy(client);
    return 0;
}
```

Example 3.3. Connecting in C

Structurally, the example in C and in C++ are similar. In the C program more details are needed to form your program, and the messages that are sent need to be constructed directly, instead of having portions of the message already created.

❶ At this point in the program, the necessary objects are declared in order to permit interaction with AMPS. When AMPS errors occur, their text is available through the `amps_client_get_error()` function, so it is here that we will create a small `char` array to hold the errors.

❷ Here an `amps_handle` is created for each object and message objects that are constructed later. An `amps_handle` is an opaque handle to an object constructed by AMPS, which cannot be dereferenced or used by means other than AMPS functions. `amps_handle` is the size of a pointer and may be passed by value wherever needed.

❸ Next we declare an `amps_result` object, which is used to store the return value from functions that may fail, such as during connection or interaction with an AMPS server. Many AMPS functions return an `amps_results`.

❹ Here we construct our AMPS client with a unique name. This function allocates resources that must be freed, and can only be freed by a corresponding call to `amps_client_destroy`.

❺ This is how a connection is established; control continues to where the AMPS message is allocated.

❻ AMPS applications communicate with the AMPS server by sending and receiving messages. A logon command, like any other command, is simply a message to AMPS. Instead of calling a function that assembles and sends the logon command, we construct the command ourselves.

When a message is constructed, the AMPS C client allocates resources that must be freed by a corresponding `amps_message_destroy` function.

❼ When a message is created in the AMPS C client, the message contains no information at all. To make the message a `logon` command, we need to set the command type to `"logon"`.

The C client provides a number of functions to assist in interacting with the data and fields of a message. In this example the `_nts` functions are used, which allow for quick population of messages fields and data with C-style null-terminated strings.

The next few lines add a minimal set of fields for the `logon` command. See the *AMPS Command Reference* for the full set of header fields supported. For simplicity in this basic example, we set the smallest number of fields possible. For example, this sample does not provide a user name or password on the command, nor does the command request an acknowledgement message. In this case, the application relies on the fact that AMPS will disconnect the client if the logon command fails. Production applications should register a message handler, request acknowledgments for each command, and take appropriate actions if the command fails.

❽ Once the command is constructed, we send the message. The return value does not indicate the result of the command sent to AMPS. Instead, the return value indicates whether the AMPS client was able to send the command to AMPS.

❾ Free the resources associated with the `logon_command` message by calling `amps_message_destroy` with the message.

❿ As with the `logon` command, we must construct a message that contains a `publish` message. While the C ++ client constructs and sends the message for us, with the C client we construct the ourselves. Note that this line also allocates resources that must be freed by a corresponding `amps_message_destroy` function.

⓫ These next few lines are responsible for setting the necessary fields and data to construct a valid `publish` message for AMPS.

⓬ Once the message is constructed to our satisfaction, it is sent. As with the `logon` command, AMPS processes the `publish` command asynchronously. If an acknowledgement is requested, AMPS returns the acknowl-

edgement message in response to the `publish` command. Your application must process that acknowledgement asynchronously.

⓭ Any errors from the operation are detected and examined here.

⓮ This where we free any message that was allocated and then destroy the client, freeing up the remaining AMPS resources.

With the C client, your application is responsible for forming commands to AMPS, receiving the responses, and interpreting the results. As mentioned above, the `AMPS Command Reference` contains detailed information on the headers that need to be set for specific commands. The Command Cookbook in Section 10.4 contains information on how to set headers for commonly used AMPS commands.

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.

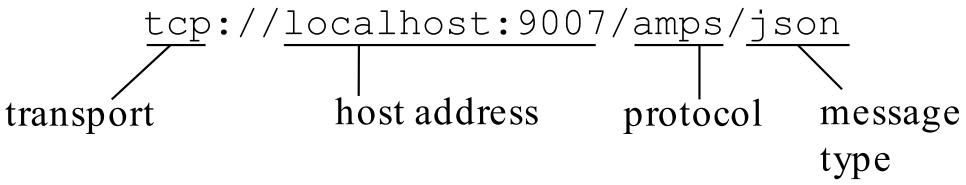

Figure 3.1. elements of a connection string

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 build an application in C or C++ 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 only to 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 C/C++ client.

4.1. Subscribing

Subscribing to an AMPS topic takes place by calling `Client.subscribe()`. Here is a short example showing the simplest way to subscribe to a topic (error handling and connection details are omitted for brevity):

```
Client client(...);
client.connect(...);❶
client.logon();

❷for ( auto message : client.subscribe("messages"))
{
      std :: cout << "Received message: "
                  ❸<< message.getData () << std :: endl ;
}
```

Example 4.1. Subscribing to a topic

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 subscription. Although we don't use the object explicitly here, the `subscribe` function 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, the MessageStream goes out of scope, the MessageStream destructor runs, and the AMPS client sends an unsubscribe command 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.

AMPS creates a background thread that receives messages and copies them into a `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 C++ client provides convenience methods for the most common form of the 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:

```
Client client(...);
client.connect(...);❶
client.logon();

❷for (auto message : ampsClient.execute(
       ❸Command("subscribe").setTopic("messages"))
```

```
{
      std :: cout << "Received message: "
                  ❹<< message.getData () << std :: endl ;
}
```

Example 4.2. Subscribing to a topic using a command

❶ Here we have created or received a Client that is properly connected to an AMPS server.
❸ Here we create a command object for the `subscribe` command, specifying the topic `messages`.
❷ Here we subscribe to 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` function 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, the MessageStream goes out of scope, the MessageStream destructor runs, and the AMPS client sends an unsubscribe command 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.

4.2. Asynchronous Message Processing Interface

The AMPS C++ client also supports an interface that allows you to process messages asynchronously. In this case, you add a message handler to the function 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 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):

```
Client client(...);
client.connect(...); ❶
client.logon();

string subscriptionId = client.executeAsync( ❷
    ❸Command("subscribe").setTopic("messages"),
    MessageHandler(myHandlerFunction, NULL));
...
void myHandlerFunction(const Message& message, void* ❹
    userData)
{
    std::cout << message.getData() << std::endl;
}
```

Example 4.3. Subscribing to a topic with asynchronous processing

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

❸ Here we create a command object for the `subscribe` command, specifying the topic `messages`.

❷ Here we create a subscription with the following parameters:

command — This is the AMPS `Command` object that contains the `subscribe` command.

MessageHandler — This is an AMPS `MessageHandler` object that refers to our message handling function `myHandlerFunction`. This function is called on a background thread each time a message arrives. The second parameter, NULL, is passed as-is from the `client.subscribe()` call to the message handler with every message, allowing you to pass context about the subscription through to the message handler.

❹ The `myHandlerFunction` is a global function that is invoked by AMPS whenever a matching message is received. The first parameter, `message`, is a reference to an AMPS Message object that contains the data and headers of the received message. The second parameter, `userData`, is set to whatever value was provided in the `MessageHandler` constructor -- NULL in this example.

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.

With newer compilers, you can use additional constructs to specify a callback function. Recent improvements in C ++ have added lambda functions -- unnamed functions declared in-line that can refer to names in the lexical scope of their creator. If available on your system, both Standard C++ Library function objects and lambda functions may be used as callbacks. Check `functional.cpp` in the samples directory for numerous examples.

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.

```
class StatefulHandler
{
   private:
       std::string _handlerName;
   public:
      // Construct the handler and save state.

      StatefulHandler(const std::string& name) : _handlerName(name) {}

      // Message handler method.
      void operator()(const AMPS::Message & message)
      {
        std::cout << _handlerName << " got "
                  << message.getData() << std::endl;
      }
};
```

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

```
client.subscribe(StatefulHandler("An instance"), "topic");
```

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. Unsubscribing

With the synchronous interface, AMPS automatically unsubscribes to the topic when the destructor for the `MessageStream` runs. You can also explicitly call the `close()` method on the `MessageStream` object to remove the subscription.

In the asynchronous interface, when a subscription is successfully made, messages will begin flowing to the message handler, and the `client.subscribe()` call will return a string for the `CommandId` that serves as the identifier for this subscription. A `Client` can have any number of active subscriptions, and this `CommandId` 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 = ...;

// Register asynchronous subscription

std::string subId = client.executeAsync(
```

```
                          Command("subscribe").setTopic("messages"),
                          MessageHandler(myHandlerFunction, NULL));

...

for (auto msg :
         client.execute(Command("unsubscribe")
                           .addAckType("processed")
                           .setSubId(subId))

{
   std::cout << "Response to unsubscribe : "
             << msg.getAckType() << std::endl;
}
```

Example 4.4. Unsubscribing from a topic

In this example, as in the previous section, we use the `client.execute_async()` method to create a subscription to the `messages` topic. When our application is done listening to this topic, it unsubscribes by passing in the `subId` returned by `subscribe()`. After the subscription is removed, no more messages will flow into our `myHandlerFunction()`.

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.

4.5. 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 `getXxx()`/`setXxx()` methods corresponding to the header fields. 60East does not recommend attempting to parse header fields from the raw data of the message.

In AMPS, fields sometimes need to be set to a unique identifier value. For example, when creating a new subscription, or sending a manually constructed message, you'll need to assign a new unique identifier to multiple fields such as `CommandId` and `SubscriptionId`. For this purpose, Message provides `new`*`Xxx`*`()` methods for each field that generates a new unique identifier and sets the field to that new value.

getData() method

Access to the data section of a message is provided via the `getData()` method. The `data` contains the unparsed data in the message, returned as a series of bytes (a `string` or `const char *`). Your application code parses and works with the data.

The AMPS C++ 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 whichever library you typically use in your environment.

4.6. Advanced Messaging Support

The `client.subscribe()` function provides options for subscribing 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 your application must process.

Regex topics

Regular Expression (Regex) Topics 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:

```
for (auto message : client.subscribe("client.*"))
{
    // receive messages for any topic that begins with 'client'
    std::cout << "Received a message on topic '" << message.getTopic() << "'
 "
              << "with the data: " << message.getData() << std::endl;
}
```

Example 4.5. Regex topic subscription

In this example, messages on topics `client` and `client1` would match the regular expression, and those messages will be returned by the `MessageStream`. As in the example, you can use the `getTopic()` method to determine the actual topic of the message sent to the lambda 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. The *AMPS User Guide* provides full details on content filtering.

To apply a content filter to a subscription, simply pass it into the `client.subscribe()` call:

```
for (auto message : ampsClient.subscribe("messages", 0, "/sender = 'mom'))
```

```
{
    // process messages from mom
}
```

Example 4.6. 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.

4.7. Next steps

At this point, you are able to build AMPS programs in C/C++ 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:

```
...
void ReadAndEvaluate(Client& client)
{
    // read a new payload from the user
    string payload;
    getline(cin, payload);
    // write a new message to AMPS
    if(!payload.empty()) {
        try {
            client.publish("UserMessage",
            string("{ \"message\" : \"data\" }");
        } catch (const AMPSException& exception)
        {
            cerr << "An AMPS exception occurred: "<<
            exception.toString() << endl;
        }
    }
}
```

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, converting the exception to a string via the `toString()` method.

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 `catch` the appropriate subclass of `AMPSException` to detect those specific errors and do something different.

```
string CreateNewSubscription(Client& client)
{
    string id;
   ❶string topicName;
    while(id.empty())
    {
        topicName = AskUserForTopicName();
        try {
       ❷id = client.subscribe(bind(HandleMessage,
```

```
            placeholders::_1),
            topicName, 5000);
        }
      ❸catch(const BadRegexTopicException& ex)
        {
            DisplayError(
               ❹ "Error: bad topic name or regular " +
                "expression '" + topicName +"'. " +
                "The error was: " + ex.toString());
            // we'll ask the user for another topic
        }
      ❺catch(const AMPSException& ex)
        {
            DisplayError(
                "Error: error setting up subscription " +
                "to topic " + topicName +". The error was: " +
                ex.toString());
           ❻return NULL; // give up
        }
    }
    return id;
    }
```

Example 5.2. Catching AMPSException Subsclasses

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 whether or not to try again based on the subclass of `AMPSException` that is thrown. If a `BadRegexTopicException`, this exception is thrown during subscription to indicate that a bad regular expression was supplied, so we would like to give the user a chance to correct.

❹ This line indicates that the program catches the `BadRegexTopicException` 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 catch block, the while loop runs again and the user is asked for another topic name.

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

❻ At this point the code stops attempting to subscribe to the client by the `return NULL` statement.

Exception Types

Each method in AMPS documents the kinds of exceptions that it can throw. For reference, 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 C++ client by default. The AMPS C++ 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 Section 5.4 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 C/C++ client's disconnect handling, you can build powerful applications that are resilient in the face of connection failures and spurious disconnects.

The `HAClient` class, included with the AMPS C++ 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. 60East recommends using the `HAClient` for automatic reconnection wherever possible, as the HA-Client disconnect handler has been carefully crafted to handle a wide variety of edge cases and potential failures. This section covers the use of a custom disconnect handler in the event that the behavior of the `HAClient` does not suit the needs of your application.

Custom 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 a disconnect occurs by supplying a function to `client.setDisconnectHandler()`, which is invoked whenever a disconnect occurs.

Example 5.3 shows the basics:

```
class MyApp
{
    string _uri;
    Client _client;
public:
    MyApp(const string& uri) : _uri(uri), _client("myapp")
    {
        _uri = uri;
       ❶_client.setDisconnectHandler(
            AttemptReconnection, (void*)this);

        _client.connect(uri);
        _client.logon();
        _client.execute_async(Command("subscribe")
                                .setTopic("orders"),
                              bind(&MyApp::ShowMessage,this
                                   placeholders::_1));
    }
    void ShowMessage(const Message& m)
    {
        // display order data to the user
        ...
    }
   ❷void AttemptReconnection(Client& client,
        void* userdata)
    {
        MyApp* app = (MyApp*) userdata;
        // simple: just try to reconnect once.
        client.connect(app->_uri);
        client.logon();
    }
```

```
    }
```

Example 5.3. Supplying a Disconnect Handler

❶ In Example 5.3 the `setDisconnectHandler()` 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. A more robust reconnect would have logic to limit either the total number of connects, frequency of connects or both. 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()`, then `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
{
   ❶ vector<string>& _uris;
    int _currentUri;
    Client _client;
public:
    MyApp(vector<string>& uris) :
        _uris(uris), _currentUri(0),
        _client("MyApp")
        {
            _client.setDisconnectHandler(
          ❷ &ConnectToNextUri, this);
           ConnectToNextUri(this);
        }

        static void ConnectToNextUri(Client client, void* me)
        {
           MyApp* app = (MyApp*)me;
          ❸while(true)
           {
               try {
               client.connect(app->_uris[app->_currentUri]);
              ❹client.subscribe(...);
               return;
           } catch(AMPSException& e) {
               app->_currentUri = (app->_currentUri + 1)
               % app->_uris.size();
           }
       }
    }
    }
```

Example 5.4. Simple Client Failover Implementation

❶ Here our application is configured with a vector of AMPS server URIs to choose from, instead of a single URI. These will be used in the `ConnectToNextUri()` method as explained below.

❷ `ConnectToNextUri()` is invoked by our disconnect handler `TestDisconnectHandler` in the AMPS Client when a disconnect occurs. Since our client is currently disconnected, we manually invoke our disconnect handler to initiate the first connection.

❸ During a disconnect the AMPS Client invokes `ConnectToNextUri()`, which loops around our array of URIs attempting to connect to each one until successful. 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 is 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 C++ client handles most incoming messages and takes appropriate action. Some messages are unexpected or occur only in very rare circumstances. The AMPS C++ 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 `UnhandledMessageHandler` 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 `UnhandledMessageHandler`.

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

In the AMPS C++ client, exceptions can occur that are not thrown to the main thread of the application. For example, when an exception is thrown from a message handler running on a background thread, AMPS does not automatically propagate that exception to the main thread.

Instead, AMPS provides the exception to an unhandled exception handler if one is specified on the client. The unhandled exception handler receives a reference to the exception object, and takes whatever action is necessary. Typically, this involves logging the exception or setting an error flag that the main thread can act on. Notice that AMPS C++ client only catches exceptions that derive from `std::exception`. If your message handler contains code that can throw exceptions that do not derive from `std::exception`, 60East recommends catching these exceptions and throwing an equivalent exception that derives from `std::exception`.

For example, the unhandled exception handler below takes a `std::ostream`, and logs information from each exception to that `std::ostream`.

```
class ExceptionLogger : public AMPS::ExceptionListener
{
  private:
      std::ostream& os_;

  public:

   ExceptionLogger() : os_(std::cout) {}
   ExceptionLogger(std::ostream& os) :
        os_(os) {}

    virtual void exceptionThrown(const std::exception& e)
    {
       os_ << e.what()
           << std::endl;
    }
}
```

5.5. Detecting Write Failures

The `publish` methods in the C++ client deliver the message to be published to AMPS and then return immediately, without waiting for AMPS to return an acknowledgement. Likewise, the `sowDelete` 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 that the command fails.

The AMPS C++ client provides a `FailedWriteHandler` that is called when the client receives an acknowledgement that indicates a failure to persist data within AMPS. To use this functionality, you implement the `FailedWriteHandler` interface, construct an instance of your new class, and register that instance with the `setFailedWriteHandler()` function on the client. 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 `FailedWriteHandler` is registered, acknowledgements that indicate errors in persisting data are treated as unexpected messages and routed to the `LastChanceMessageHandler`. 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

The 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 C++ client lets you query SOW topics and subscribe to changes with ease. AMPS SOW topics can be used as a current value cache to provide the most recently published value for each record, as a key/value object store, as the source for an aggregate or conflated topic, or all of the above uses. For more information on State of the World topics, see the *AMPS User Guide*.

6.1. Performing SOW Queries

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

```
for (auto message : ampsClient.sow("orders" ,"/symbol == 'ROL'"))
{
   if(message.getCommand() == "group_begin" )
   {
         std::cout << "Receiving messages from the SOW." << std::endl ;
   }
   else if(message.getCommand() == "group_end" )
   {
         std::cout << "Done receiving messages from SOW." << std::endl;
   }
   else {
        std::cout << "Received message: " << message.getData () <<
 std::endl;
  }
}
```

Example 6.1. Basic SOW Query

In listing Example 6.1 the program invokes `ampsClient.sow()` to initiate a SOW query on the `orders` topic, for all entries that have a symbol of 'ROL'. The SOW query is requested with a batch size of 100, meaning that AMPS will attempt to send 100 messages at a time as results are returned.

As the query executes, each matching entry in the topic at the time of the query is returned. Messages containing the data of matching entries have a `Command` of value `sow`, so as those arrive, we write them to the console. AMPS sends a "group_begin" message before the first SOW result, and a "group_end" message after the last SOW result.

When the SOW query is complete, the `MessageStream` completes iteration and the loop completes. There's no need to explicitly break out of the loop.

As with subscribe, the sow function also provides an asynchronous version. In this case, you provide a message handler that will be called on a background thread:

```
void HandleSOW(const Message& message)
{
 if (message.getCommand() == "sow")
 {
 cout << message.getData() << endl;
 }
```

```
}
void ExecuteSOWQuery(Client client)
{
 Command command("sow");
 command.setTopic("orders")
        .setFilter("/symbol='ROL'")
        .setBatchSize(100);

 client.execute_async(Command("sow")
                        .setTopic("orders")
                        .setFilter("/symbol = 'ROL'")
                        .setBatchSize(100)
                      , bind(HandleSOW, placeholders::_1));

}
```

Example 6.2. Asynchronous sow

In the listing for Example 6.2, the `ExecuteSOWQuery()` function invokes `client.sow()` to intiate a SOW query on the orders topic, for all entries that have a symbol of `ROL`. The SOW query is requested with a batch size of 100, meaning that AMPS will attempt to send 100 messages at a time as results are returned.

As the query executes, the `HandleSOW()` method is invoked for each matching entry in the topic. Messages containing the data of matching entries have a `Command` of `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 a 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.sowAndSubscribe()` method. Now we will look at an example:

```
// processSOWMessage
//
// Processes a message during SOW query. Returns
// false if the SOW query is complete, true
// if there is no more SOW processing.

bool processSOWMessage(const AMPS::Message& message)
{

    if (message.getCommand() == "group_begin")
    {
        std::cout << "Receiving messages from the SOW." << std::endl;
    }
    else if (message.getCommand() == "group_end")
```

```
    {
        std::cout << "Done receiving messages from SOW." << std::endl;
        return true;
    }
    else
    {
        std::cout << "SOW message: " << message.getData() << std::endl;
        addVan(message);
    }
    return false;
}

// processSubscriptionMessage
//
// Process messages received on a subscription, after the SOW
// query is complete.

void processSubscribeMessage(const AMPS::Message& message)
{
   if (message.getCommand() == "oof")
   {
        std::cout << "OOF : " << message.getReason()
                  << " message to remove : "
                  <<  message.getData() << std::endl;
        removeVan(message);
   }
   else
   {
       std::cout << "New or updated message: " << message.getData() <<
 std::endl;
       addOrUpdateVan(message);
   }
}

...

void doSowAndSubscribe(AMPS::Client& ampsClient)
{
        bool sowDone = false;

        std::cerr << "about to subscribe..." << std::endl;

        ❶for (auto message :
                ampsClient.execute(
                  Command("sow_and_subscribe")
                 .setTopic("van_location")
                 .setFilter("/status = 'ACTIVE'")
                 .setBatchSize(100)
                 .setOptions("oof"))
         {
             if (sowDone == false)
             {
                sowDone = processSOWMessage(message);
             }
```

```
            else
            {
               processSubscribeMessage(message);
            }
         }
}
```

Example 6.3. Using `sowAndSubscribe`

❶ In Example 6.3 we issue a `sowAndSubscribe()` to begin receiving information about all of the open orders in the system for the symbol `ROL`. These orders are now are returned as `Message`s whose `Command` returns SOW.

sub- Notice here that we specified true for the `oofEnabled` parameter. Setting this parameter to `true` causes us
ex- to receive *Out-of-Focus* ("OOF") messages for the topic. OOF messages are sent when an entry that was sent
oof: ??? to us in the past no longer matches our query. This happens when an entry is removed from the SOW cache via a `sowDelete()` 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, when an order is processed or canceled (or if the symbol changes), a `Message` is sent with `Command` set to `OOF`. The content of that message is the message sent previously. We use OOF messages to remove orders from our display as they are completed or canceled.

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

```
// handleMessage
//
// Handles messages for both SOW query and subscription.

void processSOWMessage(const AMPS::Message& message)
{

    if (message.getCommand() == "group_begin")
    {
        std::cout << "Receiving messages from the SOW." << std::endl;
        return;
    }
    else if (message.getCommand() == "group_end")
    {
        std::cout << "Done receiving messages from SOW." << std::endl;
        return true;
    }
    else  if (message.getCommand() == "oof")
    {
        std::cout << "OOF : " << message.getReason()
                  << " message to remove : "
                  <<  message.getData() << std::endl;
        removeVan(message);
    }
    else
    {
       std::cout << "New or updated message: " << message.getData() <<
 std::endl;
       addOrUpdateVan(message);
    }
}
```

```
...

std::string trackVanPositions(AMPS::Client& ampsClient)
{


        std::cerr << "about to subscribe..." << std::endl;

        return ampsClient.execute_async(
              Command("sow_and_subscribe")
               .setTopic("van_location")
               .setFilter("/status = 'ACTIVE'")
               .setBatchSize(100)
               .setOptions("oof"),
              bind(processSOWMessage(placeholders::_1));
}
```

Example 6.4. Asynchronous SOW and Subscribe

In Example 6.4, the `trackVanPositions` function invokes `sowAndSubscribe` to begin tracking vans, and returns the subscription ID. The application can later use this to unsubscribe.

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 calling thread and allows the background thread to continue to receive messages while processing occurs. In both cases, the application receives and processes the same messages.

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
MessageStream results =
    ampsClient.sowAndSubscribe("orders", "/symbol == 'ROL'");

// Turn on conflation
results.conflate();

// Process the results
for (auto message : results)
{
   // Process message here
}
```

Notice that if the `MessageStream` is used for a subscription that does not include `SowKeys` (such as a subscription to a topic that does not have a SOW), 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 functions for deleting records from the SOW.

- `sowDelete` accepts a filter, and deletes all messages that match the filter

- `sowDeleteByKeys` 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 is the internal identifier AMPS uses for that SOW message

- `sowDeleteByData` accepts a topic and message, and deletes the SOW record that would be updated by that message

Most applications use `sowDelete`, 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, `sowDeleteByKeys` 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 sowDelete command returns a MessageStream that receives the response. This response is an acknowledgement message that contains information on the delete command. For example, the following snippet simply prints informational text with the number of messages deleted:

```
for (auto msg : client.sowDelete("sow_topic",
                                 "/id in (42, 64, 37)"))
{
    std::cout << "Got a " << msg.getCommand()
              << " message containing " << msg.getAckType()
              << ": deleted " << msg.getMatches() << " entries."
              << std::endl;
}
```

The `sowDelete` command can also be sent asychronously, in a version that requires a message handler. The message handler is written to receive `sow_delete` response messages from AMPS:.

```
void HandleSOWDelete(const Message& message)
{
  std::cout << "Got a " << msg.getCommand()
              << " message containing " << msg.getAckType()
              << ": deleted " << msg.getMatches() << " entries."
              << std::endl;
}

....

client.execute_async(Command("sow_delete")
                       .setTopic("sow_topic")
                       .setFilter("/id in (42, 64, 37)")
      , bind(HandleSOWDelete, placeholders::_1));
```

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 message queues in detail, but instead explains how to use the AMPS C++ client with message queues.

To publish messages to an message 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 cmd("subscribe");
cmd.setTopic("my_queue")
   .setOptions("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 retains

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, this is implemented as a `sow_delete` from the queue with the bookmarks of the messages to remove. The AMPS C++ 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 `setAutoAck()` method.

```
client.setAutoAck(true);  // enable AutoAck
```

Message Convenience Method

The AMPS C++ 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.

```
void acknowledgeSingle(const Client & client, const Message & message)
{
    Message acknowledge;
    acknowledge.setCommand("sow_delete")
               .setTopic(message.getTopic())
               .setBookmark(message.getBookmark());
    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 C++ client automatically batches acknowledgements when either of the convenience methods is used. Batching acknowledgements reduces the number of round-trips to AMPS, reducing network traffic and improving 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.setAckBatchSize(10); // Send batch after 10 messages
client.setAckTimeout(1000); // ... or 1 second
```

The AMPS C++ 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 batch size is larger than the subscription backlog, the AMPS C++ 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 C++ 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

Command cmd("delta_subscribe");
cmd.setTopic("delta_topic");
cmd.setFilter("/thingIWant = 'true'");

for (auto m : 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

String msg = ... ; // obtain changed fields here

client.deltaPublish("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
```

```
String msg = ... ; // obtain changed fields here
String key = ... ; // obtain user-generated SOW key

Command cmd("delta_publish");
cmd.setTopic("delta_topic");
cmd.setSowKey(key);
cmd.setData(msg);

// Execute the delta publish. Use an empty
// a message handler since any failure acks will
// be routed to the FailedWriteHandler
client.executeAsync(cmd,MessageHandler());
```

Chapter 9. High Availability

The AMPS C++ 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 C++ client provides a memory-backed version of each store and a file-backed version of each store. 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 after 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. 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. The AMPS C++ client also includes default stores, which implement the appropriate interface, but do not actually persist messages.

The `HAClient` provides convenience methods for creating clients and setting stores. You can also construct an `HAClient` and set whichever 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
HAClient memoryClient = HAClient::createMemoryBacked(
    "lessImportantMessages");

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

// Default publish store, file-backed bookmark store
HAClient subscriberClient("subscriber");
subscriberClient.setBookmarkStore(
   new LoggedBookmarkStore("my_app.bookmark"));

// Default publish store, default bookmark store
// Failover behavior only
HAClient streamReader("streamReader");
```

Example 9.1. HAClient creation examples

While this chapter presents the built-in file and memory-based stores, the AMPS C/C++ Client provides open interfaces that allow development of custom persistent message stores. You can implement the `Store` and `BookmarkStore` interfaces in your code, and then pass instances of those to `setPublishStore()` or `setBookmarkStore()` methods in your `Client`. Instructions on developing a custom store are beyond the scope of this document; please refer to the *AMPS Client HA Whitepaper* for more information.

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 C/C++ 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.

In either case, you must provide a `ServerChooser` to `HAClient` and then call `connectAndLogon()` to create the first connection.

```
HAClient myClient = HAClient::createMemoryBacked(
    "myClient");

// primary.amps.xyz.com is the primary AMPS instance, and
//  secondary.amps.xyz.com is the secondary
ServerChooser chooser(new DefaultServerChooser());
chooser.add("tcp://primary.amps.xyz.com:12345/fix");
chooser.add("tcp://secondary.amps.xyz.com:12345/fix");
myClient.setServerChooser(chooser);
myClient.connectAndLogon();
...
myClient.disconnect();
```

Example 9.2. HAClient logon

Similar to `Client`, `HAClient` remains connected to the server until `disconnect()` is called. Unlike `Client`, `HAClient` provides a built-in disconnect handler that 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 log in 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, `HAClient` attempts to reconnect to the server it was most recently connected with, and, if that is not possible, it moves on to the next server provided by the `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 C++ Client includes an interface for managing this behavior called the `ReconnectDelayStrategy`.

Two implementations of this interface are provided with the client:

- `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 with 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.

```
HAClient theClient = HAClient::createMemoryBacked("demo");
```

```
theClient.setReconnectDelayStrategy(
   new ExponentialDelayStrategy(200,
                                5000,
                                1.5,
                                6000,
                                0));
```

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 getCurrentURI() 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 getCurrentAuthenticator() method.

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

Each time a connection succeeds, the HAClient calls the reportSuccess() method of the ServerChooser. Each time a connection fails, the HAClient calls the reportFailure() 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 getCurrentURI(), indicating that no servers are available for connection, the HAClient calls getError() method on the ServerChooser and includes the string returned by getError() 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 utilize heartbeat, call the `setHeartbeat` method on `Client` or `HAClient`:

```
HAClient client = HAClient::createMemoryBacked(
    "importantStuff");
...
client.setHeartbeat(3);
client.connectAndLogon();
...
```

`setHeartbeat` 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 `setHeartbeat` 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` uses a publish store (other than the `DefaultPublishStore`), 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 `unpersistedCount()` method that reports the number of messages that have not yet been acknowledged by the AMPS server. When the `unpersistedCount()` reaches `0`, there are no unpersisted messages in the local publish store.

For the highest level of safety, an application can wait until the `unpersistedCount()` 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 `publishFlush()` 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:

```
HAClient pub = HAClient.createMemoryBacked(
    "importantStuff");
...
pub.connectAndLogon();
std::string  topic = "loggedTopic";
std:string data = ...;
for(size_t i = 0; i < MESSAGE_COUNT; i++)
{
    pub.publish(topic, data);
}

// We think we are done, but the server may not
// have received or acknowledged the 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.

pub.publishFlush();

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

if (pub.getPublishStore().unpersistedCount() > 0)
{
    printf("all messages have been published,"
           " but not all have been persisted.");
}

pub.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.

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 `unpersistedCount()` 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, the store created by using `HAClient.createFileBacked()`), then the `HAClient` will recover the messages, and once logged on, 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 passed to `createFileBacked()` 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 C++ 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 C++ Client version 4.3.1.1) does not track synchronous message processing subscriptions in the `SubscriptionManager`. The reason for this is to preserve conventional iterator behavior. That is, once the `MessageStream` indicates that there are no more elements to iterate (for example, because the connection has closed), the `MessageStream` will 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 a subscribe command when the message stream ends:

```
bool still_need_to_process = true;

while (still_need_to_process)
{
    for ( auto message : client.subscribe("messages"))
    {
        // process messages

        // check condition on still_need_to_process

        if (!still_need_to_process) break;
    }
    // end of stream: for a subscribe this means
    // that the connection is likely closed, or
    // the program broke out of the loop

}
```

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 bookmarkSubscribe() instead of subscribe() when creating a subscription, and decide how the application will manage subscription identifiers (SubIds).
- Use the BookmarkStore.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:

```
HAClient client = HAClient::createFileBacked(
    "theClient",
    "/logs/theClient.publishLog",
    "/logs/theClient.subscribeLog");

namespace MyMessageHandler
{
    public void invoke(const Message& message, void* data)
    {
        ...
        client.getBookmarkStore().discard(message);
        ...
    }
}

std::string commandID =
    client.execute_async(Command("subscribe")
                         .setTopic("myTopic")
                         .setSubscriptionId("MySubId")
                         .setBookmark(AMPS::Client::BOOKMARK_RECENT()),
            AMPS::MessageHandler(MyMessageHandler::invoke,(void*)
(&client)));
```

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 (`theClient.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, the client and the AMPS server will incur unnecessary network, disk, and CPU activity.

In Example 9.4 all parameters after the bookmark are optional. However, all options before — and including the bookmark — are required when creating a `bookmarkSubscribe()`.

The last parameter, `subId`, specifies an identifier to be used for this subscription. Passing `NULL` 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.

The `subId` is also required to be unique when used within a single client, but can be the same in different clients. Internally, AMPS tracks subscriptions in each client, thus each identifier for each subscription within a client must be unique. The same `subId` can be reused across unique clients simultaneously without causing problems.

- `Client::BOOKMARK_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::BOOKMARK_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::BOOKMARK_RECENT()` specifies that the subscription should begin from the last-used message in the associated `BookmarkStore`, or, if this subscription has not been seen before, 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 discarded 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 file-based `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 AMPS, in cases where you need to use an earlier verison 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:

```
AMPS::Command command("sow");
```

Once created, you set the appropriate fields on the command. For example, the following code creates a publish message, setting the command, topic, data to publish, and an expiration for the message:

```
AMPS::Command command("sow")
       .setTopic("messages-sow")
       .setFilter("/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. The `execute` method returns a `MessageStream` that provides response messages. The `executeAsync` method sends the command 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);
```

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 `executeAsync` 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:

```
void handleMessages(const AMPS::Message& m, void* user_data)
{
   // print acknowledgement type and reason for sample purposes.
   std::cout << m.getAckType() << " : " << m.getReason() << std::endl;
}

...

client.executeAsync(command, AMPS::MessageHandler(handleMessages, NULL));

...
```

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 a command failed.

Notice that the `publish` command does not provide typically return results other than acknowledgement messages, so there is little need for a message handler with a `publish` command. To send a `publish` command, use the `executeAsync()` method with a default-constructed message handler. With a default-constructed message handler, AMPS does not enter the message handler in the internal routing table, which improves efficiency for commands that do not expect a response:

```
client.executeAsync(publishCmd, AMPS::MessageHandler());
```

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.

Header	Comment
	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.

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.

Header	Comment

`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`

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 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.

Header	Comment
`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]`

Header	Comment
	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.

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.
`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: `1853097931817257202,1040277994020165007 5,223638799`

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:

Header	Comment
	`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.
`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 C++ 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 a standard `vector`.

```
std::string json_part("{\"data\":\"sample\"}");

std::vector<double> data;
// populate data
...

// Create the payload for the composite message.

AMPS::CompositeMessageBuilder builder;

builder.append(json_part.str());
builder.append(reinterpret_cast<const char*>(data.data()),
                   data.size() * sizeof(double));

// send the message

std::string topic("messages");
ampsClient.publish(topic.c_str(), topic.length(),
                   builder.data(), builder.length());
```

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.

```
for (auto message : ampsClient.subscribe("messages"))
{

  parser.parse(message);

  // First part is JSON
  std::string json_part = std::string(parser.getPart(0));

  // Second part is the raw bytes for a vector<double>
  AMPS::Field binary = parser.getPart(1);

  std::vector<double> vec;
  double *array_start = (double*)binary.data();
  double *array_end = array_start + (binary.len() / sizeof(double));

  vec.insert(vec.end(), array_start, array_end);

  // Print the contents of the message
  std::cout << "Received message with " << parser.size() << " parts"
            << std::endl
            << "\t" << json_part
            << std::endl;

  for (auto d : vec)
       std::cout << d << " ";

  std::cout << std::endl;

 }
```

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 message types.

- `NVFIXBuilder` allows you to assemble an NVFIX message and then serialize them 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 NVFIX message and publish it to the AMPS client.

```
// Construct a client with the name "NVFIXPublisher".
AMPS::Client ampsClient("NVFIXPublisher");

// Construct a simple NVFIX message.
AMPS::NVFIXBuilder builder;

// Add data to the builder
builder.append("Test", "data");
builder.append("More", "stuff");

// Display the data
std::cout << builder.getString() << std::endl;

try
{
  // Connect to the server and log on
  ampsClient.connect(uri);
  ampsClient.logon();

  // Publish message to the topic messages
  std::string topic("messages");
  ampsClient.publish(topic, builder.getString());

}
catch (const AMPS::AMPSException& e)
{
  std::cerr << e.what() << std::endl;
  exit(1);
}
```

Parsing NVFIX Messages

To parse a NVFIX message, create an instance of `FIXShredder`, then use the `toMap()` 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 an NVFIX message.

```
// Create a client with the name "NVFIXSubscriber"
AMPS::Client ampsClient("NVFIXSubscriber");

try
{
  // Connect to the server and log on
```

```
  ampsClient.connect(uri);
  ampsClient.logon();

  // Subscribe to the messages topic

  // This overload of the subscribe method returns a MessageStream
  // that can be iterated over.  When the MessageStream destructor
  // runs, the destructor unsubscribes.

  // Set up the shredder
  AMPS::FIXShredder shredder;

  for(auto message : ampsClient.subscribe("messages"))
  {
    // Shred the data to a map
    auto subscription = shredder.toMap(message.getData());

    // Display the data
    for(auto iterator = subscription.begin(); iterator !=
 subscription.end(); ++iterator)
    {
       std::cout << iterator->first << " " << iterator->second << std::endl;
    }
  }
}
catch (const AMPS::AMPSException& e)
{
  std::cerr << e.what() << std::endl;
  exit(1);
}
```

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 them 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.

```
// Construct a client with the name "FIXPublisher".
AMPS::Client ampsClient("FIXPublisher");
```

```
// Construct a simple FIX message.
AMPS::FIXBuilder builder;

// Add data to the builder
builder.append(0, "123");

// Display the data
std::cout << builder.getString() << std::endl;

try
{
  // connect to the server and log on
  ampsClient.connect(uri);
  ampsClient.logon();

  // publish message to the messages topic
  std::string topic("messages");
  ampsClient.publish(topic, builder.getString());

}
catch (const AMPS::AMPSException& e)
{
  std::cerr << e.what() << std::endl;
  exit(1);
}
```

Parsing FIX Messages

To parse a FIX message, create an instance of `FIXShredder`, then use the `toMap()` 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 a client with the name "NVFIXSubscriber"
AMPS::Client ampsClient("NVFIXSubscriber");

try
{
  // Connect to the server and log on
  ampsClient.connect(uri);
  ampsClient.logon();

  // Subscribe to the messages topic

  // This overload of the subscribe method returns a MessageStream
  // that can be iterated over.  When the MessageStream destructor
  // runs, the destructor unsubscribes.

  // Set up the shredder
  AMPS::FIXShredder shredder;
```

```
  for(auto message : ampsClient.subscribe("messages"))
  {
    // Shred the data to a map
    auto subscription = shredder.toMap(message.getData());

    // Display the data
    for(auto iterator = subscription.begin(); iterator !=
 subscription.end(); ++iterator)
    {
       std::cout << iterator->first << " " << iterator->second << std::endl;
    }
  }
}
catch (const AMPS::AMPSException& e)
{
  std::cerr << e.what() << std::endl;
  exit(1);
}
```

Chapter 12. Advanced Topics

12.1. Transport Filtering

The AMPS C/C++ 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 function with the following signature

```
void amps_tcp_filter_function(const unsigned char* data,size_t len,short
 direction, void* userdata);
```

You then register the filter by calling `amps_tcp_set_filter_function` with a pointer to the function and a pointer to the data to be provided in the userdata parameter of the callback.

For example, the following filter function simply prints the data provided to the standard output:

```
void amps_tcp_trace_filter_function(const unsigned char* data,
                                    size_t len,
                                    short direction,
                                    void* userdata)
{
   // Output the direction marker
   if (direction == 0)
    {
          std::cout << "OUTGOING ---> ";
    }
    else
    {
         std::cout << "INCOMING ---> ";
    }

    // Output the data
    std::cout << std::string(data, len) << std::endl;

}
```

Registering the function is a matter of calling the amps_set_transport_filter_function with the transport to filter, as shown below:

```
  // client is an existing Client object


 amps_tcp_set_filter_function(amps_client_get_transport(client.getHandle()),
                              &amps_tcp_trace_filter_function,
                              (void*)NULL);
```

The snippet above gets the underlying C client handle from the C++ class, retrieves the transport handle associated with the client handle, and the installs the filter for that transport.

12.2. Using SSL

The AMPS C++ client includes support for Secure Sockets Layer (SSL) connections to AMPS. The client automatically attempts to make an SSL connection when the transport in the connection string is set to `tcps`, as described in connection string is set to `tcps`, as described in Section 3.3.

To use the `tcps` transport, your application must have an SSL library loaded before making the `tcps` connection. Notice that the AMPS C++ client uses the OpenSSL implementation that you provide. The AMPS client distribution doesn't include OpenSSL, and doesn't provide facilities for certificate generation, certificate signing, key management, and so forth. Those facilities are provided by the OpenSSL implementation you choose.

Loading the SSL Library

To make an SSL connection, the AMPS client must have an SSL library loaded before making the SSL connection.

There are two common ways to load the library:

1. At link time, specify the OpenSSL shared object file (Linux) or DLL (Windows) to the linker. With this approach, the operating system will load the SSL library for your application automatically when the application starts up.

2. Use the `amps_ssl_init` function to load the SSL library. This function accepts either the library name, or a full path including the library name. When called with the library name, the AMPS C++ client will search appropriate system paths for shared libraries (for example, the `LD_LIBRARY_PATH` on Linux) and load the first object found that matches the provided name. When called with a full path, the AMPS C++ client will load exactly the object specified.

Once the SSL library is loaded, you can connect using `tcps` as a transport type. The fact that the connection uses a secure socket is only important when making the connection, and does not affect the operation of the client once the connection has been made.

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.

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
AlreadyConnected	Connecting	Thrown when `connect()` is called on a Client that is already connected.
AMPS	Anytime	Base class for all AMPS exceptions.
Authentication	Anytime	Indicates an authentication failure occurred on the server.
BadFilter	Subscribing	This typically indicates a syntax error in a filter expression.
BadRegexTopic	Subscribing	Indicates a malformed regular expression was found in the topic name.
Command	Anytime	Base class for all exceptions relating to commands sent to AMPS.
Connection	Anytime	Base class for all exceptions relating to the state of the AMPS connection.
ConnectionRefused	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.
Disconnected	Anytime	No connection is available when AMPS needed to send data to the server *or* the user's disconnect handler threw an exception.
InvalidTopic	SOW query	A SOW query was attempted on a topic not configured for SOW on the server.
InvalidTransportOptions	Connecting	An invalid option or option value was specified in the URI.
InvalidURI	Connecting	The URI string provided to `connect()` was formatted improperly.
MessageType	Connecting	The class for a given transport's message type was not found in AMPS.
MessageTypeNotFound	Connecting	The message type specified in the URI was not found in AMPS.
NameInUse	Connecting	The client name (specified when instantiating `Client`) is already in use on the server.
RetryOperation	Anytime	An error occurred that caused processing of the last command to be aborted. Try issuing the command again.
Stream	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 `CommandId` as another subscription. Create a unique `CommandId` for every subscription.
TimedOut	Anytime	A timeout occurred waiting for a response to a command.
TransportType	Connecting	Thrown when a transport type is selected in the URI that is unknown to AMPS.
Unknown	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.