AMPS C/C++ Development Guide

AMPS C/C++ Development Guide

4.3

Publication date Oct 29, 2015

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

An empty directory tree where built libraries are placed. Before using the AMPS C++ client, you must build them using a C++ compiler and the provided makefile or solution file.

src

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

inc

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

spark

Spark is a command-line utility for interacting with AMPS, provided with each client library. This directory includes the source code for Spark, as an example of how to build a more complex application with AMPS.

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, samples, and spark client are built in the `lib`, `samples` and `spark` 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 Windows Command Prompt and change the directory to the `AMPS` directory in your AMPS installation, and use `spark.exe` to test connectivity to your server, for example:

```
./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";

    // 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";


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

   ❹try {
        ❺ampsClient.connect("tcp://127.0.0.1:9007/amps");
        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`. 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 creates a named connection in AMPS.

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 make your programs more robust.

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. 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 client;
   ❸amps_result result;

    client = amps_client_create("cClient"); ❹

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

    if(result != AMPS_E_OK) {
        amps_client_get_error(
            client, errorBuffer, sizeof(errorBuffer));
        printf("error %s\n", errorBuffer);
    } else {
       ❻message = amps_message_create(client);
       ❼amps_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\"}");
          ❽result = amps_client_send(client, message);
        if(result != AMPS_E_OK){
           ❾amps_client_get_error(
            client, errorBuffer, sizeof(errorBuffer));
            printf("error sending: %s\n", errorBuffer);
        }
       ❿ 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 deref-

erenced 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 Unlike the convenience methods in the C++ client, every message sent to the server must be created by your application. Instead of calling a function to send a `publish` message for us, we construct it ourselves. Note that this line also allocates resources that must be freed by a corresponding `amps_message_destroy` function.

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

❽ Once the message is constructed to our satisfaction, it is sent.

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

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

3.3. Connection Strings

The AMPS clients use connection strings to determine the server, port, transport, and protocol to use to connect to AMPS. Connection strings have three 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`.

- *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`, the address consists of a host name and port number. In this case, the host address is `127.0.0.1:9007`.

- *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.

This connection string works for programs connecting from the local host to a transport configured as follows:

```
<AMPSConfig>
...
    <Transport>
      <Name>json-tcp</Name>
      <Type>tcp</Type>
      <InetAddr>9007</InetAddr>
      <ReuseAddr>true</ReuseAddr>
      <MessageType>json</MessageType>
      <Protocol>amps</Protocol>
    </Transport>
...
</AMPSConfig>
```

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

Providing Credentials in a Connection String

The AMPS clients support the standard format for including a user name and password in a URI, as shown below:

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

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?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. 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.

AMPS supports the value of `tcp` in the connection string for TCP/IP connections, and the value of `shm` in the connection string for the AMPS shared memory protocol.

Transport options

The following transport options are available:

`tcp_rcvbuf` (integer) Sets the system receive buffer size.

`tcp_sndbuf` (integer) Sets the system send buffer size.

`tcp_nodelay` (boolean) Enables or disables the `TCP_NODELAY` setting on the socket.

`tcp_linger` (integer) Enables and sets the `SO_LINGER` value for the socket.

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.

4.2. Asynchronous Subscribe Interface

The AMPS C++ client also supports an advanced, asynchronous interface. 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 may be a performance benefit to enqueuing work directly from the background thread.

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

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

string subscriptionId = client.execute_async( 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 asynchronously

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.

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, 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.

For advanced subscription, AMPS calls the handler function from the background thread. Message handlers provided to advanced subscriptions 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. 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.execute_async(
                          Command("subscribe").setTopic("messages"),
                          MessageHandler(myHandlerFunction, NULL));

...

for (auto msg :
         client.execute(Command("unsubscribe")
                           .setSubscriptionId(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()`.

4.5. Understanding messages

So far, we have seen that subscribing to a topic involves creating a lambda function that receives a single parameter, an `AMPS::Message`. A `Message` represents a single message to or from an AMPS server and client. Messages are received or sent for every client/server operation in AMPS. `Message` contains a variety of methods:

`Message` contains member functions for every header field and for the message data, for both setting and getting. In AMPS, every message has one or more header fields defined, depending 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 corresponding `getXXX()` and `setXXX()` method, which may be used to retrieve and set the value of that message. For example, the `getCommandId()` method retrieves the string representing the `CommandId` header field, and the `setBatchSize()` method sets the value of the `BatchSize` header field. For more information on these header fields, consult the *AMPS User Guide*.

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 `newXXX()` methods for each field that generates a new unique identifier and sets the field to that new value.

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.

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

```
for (auto message : ampsClient.subscribe("messages"))
{
    // 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 simply replaces the filter on the subscription. AMPS does not re-run the SOW query.

To update a the filter on a subscription, you create a `subscribe` command. You set the subscriptionId of the Command to the identifier of the existing subscription, and include the `replace` option on the Command. Many of the named convenience methods also accept a bookmark parameter, and replace a subscription when the bookmark is provided and the `replace` option is included in the call.

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.

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.

AMPS disconnect handling gives you the ultimate in control and flexibility regarding how to respond to disconnects. Your application gets to specify exactly what happens when 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.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);
    }
    }
```

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.

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 user. 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

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.

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 `client.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-ex-oof: ??? Notice here that we specified true for the `oofEnabled` parameter. Setting this parameter to `true` causes us to receive *Out-of-Focus* ("OOF") messages for the topic. OOF messages are sent when an entry that was sent to us in the past no longer matches our query. This happens when an entry is removed from the SOW cache via a `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. 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.

6.4. Managing SOW Contents

AMPS allows application 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));
```

Chapter 7. High Availability

The AMPS Client provides an easy way to create highly-available applications using AMPS, via the `HAClient` class. 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.
- Ensure no messages are lost or duplicated after a reconnect or failover.
- (Optional) Persist messages and bookmarks on disk for protection against client failure.

Many of these features require specific configuration settings on your AMPS instance(s). This chapter mentions these features, but you can find full documentation for these settings and server features in the *User Guide*.

7.1. Choosing an HAClient Protection Method

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 protection from disconnection using *Stores*. As the name implies, *stores* hold information about the state of the client. There are two types of store:

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

The AMPS client provides a memory-backed version of each store and a filed-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 protect against 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 protect against client failure and disconnection from AMPS by storing messages and bookmarks on disk. To use this protection method, the `create_file_backed` method requests additional arguments for the two files that will be used for both bookmark storage and message storage.

If these files exist and are non-empty (as they would be after a client application is restarted), the `HAClient` loads their contents and ensures synchronization with the AMPS server once connected. The performance of this option depends heavily on the speed of the device on which these files are placed. When the files do not exist (as they would the first time a client starts on a given system), the `HAClient` creates and initializes the files, and in this case the client does not have a point at which to resume the subscription or messages to republish.

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

In this example, we create two clients, one for ”less-important” messages that uses memory for its store, and one which uses a pair of files for its store:

```
HAClient memoryClient = HAClient::createMemoryBacked(
    "lessImportantMessages");
HAClient diskClient = HAClient::createFileBacked(
    "moreImportantMessages",
    "/mnt/fastDisk/moreImportantMessages.outgoing",
    "/mnt/fastDisk/moreImportantMessages.incoming");
```

Example 7.1. 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.

7.2. 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 disconnect. 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 you can use in applications with simple requirements around choosing which

server to connect with. Or, you can implement `ServerChooser` yourself for more advanced logic, such as choosing a backup server based on your network topology.

In either case, you must provide a `ServerChooser` to `HAClient` to get started, and then invoke `connectAndLogon()` to create the first connection. If no `ServerChooser` is provided, the `HAClient` throws an exception:

```
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 7.2. HAClient logon

Similar to `Client`, `HAClient` remains connected to the server until `disconnect()` is called. Unlike `Client`, `HAClient` automatically attempts to reconnect to your server if it detects a disconnect, and, if that server cannot be connected, fails over to the next server provided by the `ServerChooser`. In this example, the call to `connectAndLogon()` attempts to connect and 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`.

7.3. 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.connectAndLogon();
client.setHeartbeat(3);
...
```

`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.

7.4. 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 these messages asynchronously for maximum performance, but, before exiting or terminating your connection, you should ensure that the server has received all of your messages. The AMPS server occasionally sends persisted acknowledgement messages that indicate messages it has successfully received and persisted. For safety, your application should wait until it has successfully received the final acknowledgement from the AMPS instance. Use the `unpersistedCount()` method in the `Store` to determine how many messages remain unacknowledged by the AMPS instance, as in the following example:

```
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 acknowledged us yet.
while(pub.getPublishStore().unpersistedCount() > 0)
{
    printf(("waiting for final ack from the server...");
    sleep(1000);
}
pub.disconnect();
```

Example 7.3. HA Publisher

In this example, the client sends each message immediately when `publish()` is called, but if AMPS becomes unavailable between the final `publish()` and the `disconnect()`, then the client may not have received an acknowledgement for all of the published messages. It is possible that not every message has been received or persisted by the AMPS server. By waiting until `unpersistedCount()` becomes 0, the application ensures that it has received acknowledgement for every message published. If a disconnect or failover occurs while waiting, `HAClient` automatically reconnects and correlates its internal store with the AMPS server (via the client sequence number returned in the acknowledgement message from the logon), replaying any messages the AMPS server might need in order to be consistent.

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.

If your application crashes or is terminated by an outside force, 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*.

7.5. Considerations for Subscribers

`HAClient` provides two important features for applications that subscribe to one or more topics: resubscription, 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 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 gets published after the subscription is placed.

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 the iterator semantics. That is, once the MessageStream indicates that there are no more elements in the stream, it does not suddenly produce more elements.

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

```
bool still_need_to_process = true;

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

        // check condition on still_need_to_process

        if (still_need_to_process == false) break;
    }
    // end of stream, for a subscribe this means
    // that the connection is likely closed.

}
```

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 book-

markSubscribe(), whenever a disconnection or failover occurs, your application automatically re-subscribes 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(
    "aClient",
    "/logs/aClient.publishLog",
    "/logs/aClient.subscribeLog");

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

AMPS::Command command("subscribe");
command.setTopic("myTopic")
       .setSubscriptionId("MySubId")
       .setBookmark(AMPS::Client::BOOKMARK_RECENT());

std::string commandID =
    client.execute_async(Command("subscribe")
                         .setTopic("myTopic")
                         .setSubscriptionId("MySubId")

 .setBookmark(AMPS::Client::BOOKMARK_RECENT()),
             AMPS::MessageHandler(MyMessageHandler::invoke,(void*)
(&client)));
```

Example 7.4. HAClient Subscription

In this example, the client is a file-backed client, meaning that arriving bookmarks will be stored in a file (`Client.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 7.4 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.

- `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 used before the disconnect, so that no messages are missed.

7.6. 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 8. Advanced 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.

8.1. 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*.

8.2. 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.

8.3. Using execute

Once you've created a message, use the `execute` method to send the message to AMPS. One form of the `execute` method allows you to provide a message handler to process response messages. The `execute_async` method sends the message to AMPS and processes any response on a 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. In this case, AMPS creates a background thread. The call to `execute_async` returns immediately on the main thread, and messages are received in 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.execute_async(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 the command failed.

Notice that the `publish` command does not provide typically return results other than acknowledgement messages. To send a `publish` command, use the `executeAsync()` method with a `NULL` message handler:

```
client.executeAsync(publishCmd, NULL);
```

8.4. 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 is most often used with asynchronous message processing, while passing an empty handler. To use these commands with the synchronous message processing interface, add a `CommandId` to the `Command`

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 8.1. 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.

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.

Publish With CorrelationId

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

Table 8.2. Publish With CorrelationId

Header	Comment
`Topic`	Sets the topic to publish to. The topic specified must be a literal topic name. Regular expression characters in the topic name are not interpreted. Some topics in AMPS, such as views and conflated topics, cannot be published to directly. Instead, a publisher must publish to the underlying topics.
`Data`	The data to publish to the topic. The AMPS client does not interpret, escape, or validate this data: the data is provided to the server verbatim.
`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 8.3. 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.

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.
`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.

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 8.4. 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 8.5. 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 8.6. 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.

Bookmark 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. The `Bookmark` option is only supported for topics that are recorded in an AMPS transaction log.

Table 8.7. 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 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 8.8. 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.

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 8.9. 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.

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 8.10. 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 8.11. 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 8.12. 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

Header	Comment
	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.

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 8.13. 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

Header	Comment
	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 8.14. 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 8.15. 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 8.16. 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 8.17. 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: `1853097931817257202,1040277994020165007522363`

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 8.18. 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 8.19. 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 8.20. Content Filtered SOW and Subscribe

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

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

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 8.21. 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 8.22. 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 8.23. 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 8.24. 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.

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

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 8.25. 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 8.26. Basic Delta Subscription

Header	Comment
`Topic`	Sets the topic to subscribe to. All messages from the topic will be delivered on this subscription. The top-

Header	Comment
	ic 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 8.27. 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 8.28. 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.

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 8.29. 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 8.30. 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. 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 8.31. 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 8.32. 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 8.33. 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 8.34. 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 8.35. 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.

Chapter 9. Utilities

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

9.1. 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

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.

Chapter 10. Advanced Topics

10.1. 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.

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.

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.

Exception	When	Notes
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.
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.